University of Amsterdam
Kruislaan 419, 1098 VA Amsterdam
VU University Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam
The Netherlands

SWI-Prolog 6.0

Reference Manual

Updated for version 6.0.2, March 2012

Jan Wielemaker
`J.Wielemaker@cs.vu.nl`
http://www.swi-prolog.org

SWI-Prolog is a Prolog implementation based on a subset of the WAM (Warren Abstract Machine). SWI-Prolog was developed as an *open* Prolog environment, providing a powerful and bi-directional interface to C in an era when this was unknown to other Prolog implementations. This environment is required to deal with XPCE, an object-oriented GUI system developed at SWI. XPCE is used at SWI for the development of knowledge-intensive graphical applications.

As SWI-Prolog became more popular, a large user community provided requirements that guided its development. Compatibility, portability, scalability, stability and providing a powerful development environment have been the most important requirements. Edinburgh, Quintus, SICStus and the ISO-standard guide the development of the SWI-Prolog primitives.

This document gives an overview of the features, system limits and built-in predicates.

Copyright © 1990–2012, University of Amsterdam

Contents

1 Introduction **10**
- 1.1 SWI-Prolog . . . 10
  - 1.1.1 Books about Prolog . . . 10
- 1.2 Status . . . 11
- 1.3 Compliance to the ISO standard . . . 11
- 1.4 Should you be using SWI-Prolog? . . . 11
- 1.5 The XPCE GUI system for Prolog . . . 12
- 1.6 Release Notes . . . 13
- 1.7 Donate to the SWI-Prolog project . . . 19
- 1.8 Acknowledgements . . . 20

2 Overview **21**
- 2.1 Getting started quickly . . . 21
  - 2.1.1 Starting SWI-Prolog . . . 21
  - 2.1.2 Executing a query . . . 22
- 2.2 The user's initialisation file . . . 22
- 2.3 Initialisation files and goals . . . 23
- 2.4 Command-line options . . . 23
  - 2.4.1 Controlling the stack-sizes . . . 24
  - 2.4.2 Running goals from the commandline . . . 25
  - 2.4.3 Compiler options . . . 25
  - 2.4.4 Maintenance options . . . 26
- 2.5 GNU Emacs Interface . . . 26
- 2.6 Online Help . . . 27
- 2.7 Command-line history . . . 28
- 2.8 Reuse of top-level bindings . . . 28
- 2.9 Overview of the Debugger . . . 29
- 2.10 Compilation . . . 32
  - 2.10.1 During program development . . . 32
  - 2.10.2 For running the result . . . 33
- 2.11 Environment Control (Prolog flags) . . . 36
- 2.12 An overview of hook predicates . . . 46
- 2.13 Automatic loading of libraries . . . 47
- 2.14 Garbage Collection . . . 49
- 2.15 Syntax Notes . . . 49
  - 2.15.1 ISO Syntax Support . . . 49
- 2.16 Rational trees (cyclic terms) . . . 53
- 2.17 Just-in-time clause indexing . . . 54
  - 2.17.1 Future directions . . . 55
  - 2.17.2 Indexing and portability . . . 55

2.18 Wide character support . . . 55
2.18.1 Wide character encodings on streams . . . 55
2.19 System limits . . . 57
2.19.1 Limits on memory areas . . . 57
2.19.2 Other Limits . . . 59
2.19.3 Reserved Names . . . 59
2.20 SWI-Prolog and 64-bit machines . . . 60
2.20.1 Supported platforms . . . 60
2.20.2 Comparing 32- and 64-bits Prolog . . . 60
2.20.3 Choosing between 32- and 64-bits Prolog . . . 60

3 Initialising and Managing a Prolog Project **62**
3.1 The project source files . . . 62
3.1.1 File Names and Locations . . . 62
3.1.2 Project Special Files . . . 63
3.1.3 International source files . . . 64
3.2 Using modules . . . 64
3.3 The test-edit-reload cycle . . . 65
3.3.1 Locating things to edit . . . 65
3.3.2 Editing and incremental compilation . . . 66
3.4 Using the PceEmacs built-in editor . . . 66
3.4.1 Activating PceEmacs . . . 66
3.4.2 Bluffing through PceEmacs . . . 67
3.4.3 Prolog Mode . . . 69
3.5 The Graphical Debugger . . . 71
3.5.1 Invoking the window-based debugger . . . 71
3.6 The Prolog Navigator . . . 72
3.7 Cross referencer . . . 72
3.8 Accessing the IDE from your program . . . 74
3.9 Summary of the IDE . . . 75

4 Built-in predicates **76**
4.1 Notation of Predicate Descriptions . . . 76
4.2 Character representation . . . 76
4.3 Loading Prolog source files . . . 77
4.3.1 Conditional compilation and program transformation . . . 86
4.3.2 Loading files, active code and threads . . . 89
4.3.3 Quick load files . . . 91
4.4 Editor Interface . . . 91
4.4.1 Customizing the editor interface . . . 92
4.5 List the program, predicates or clauses . . . 93
4.6 Verify Type of a Term . . . 94
4.7 Comparison and Unification of Terms . . . 95
4.7.1 Standard Order of Terms . . . 96
4.7.2 Special unification and comparison predicates . . . 97
4.8 Control Predicates . . . 99
4.9 Meta-Call Predicates . . . 100

4.10 ISO compliant Exception handling . . . 104
  4.10.1 Debugging and exceptions . . . 104
  4.10.2 The exception term . . . 105
  4.10.3 Printing messages . . . 105
4.11 Handling signals . . . 108
  4.11.1 Notes on signal handling . . . 109
4.12 DCG Grammar rules . . . 110
4.13 Database . . . 111
  4.13.1 Update view . . . 114
  4.13.2 Indexing databases . . . 114
4.14 Declaring predicates properties . . . 115
4.15 Examining the program . . . 116
4.16 Input and output . . . 121
  4.16.1 ISO Input and Output Streams . . . 121
  4.16.2 Edinburgh-style I/O . . . 127
  4.16.3 Switching Between Edinburgh and ISO I/O . . . 129
  4.16.4 Write onto atoms, code-lists, etc. . . . 130
4.17 Status of streams . . . 131
4.18 Primitive character I/O . . . 132
4.19 Term reading and writing . . . 135
4.20 Analysing and Constructing Terms . . . 141
  4.20.1 Non-logical operations on terms . . . 142
4.21 Analysing and Constructing Atoms . . . 144
4.22 Character properties . . . 147
  4.22.1 Case conversion . . . 148
  4.22.2 White space normalization . . . 149
  4.22.3 Language specific comparison . . . 149
4.23 Representing text in strings . . . 149
4.24 Operators . . . 150
4.25 Character Conversion . . . 152
4.26 Arithmetic . . . 152
  4.26.1 Special purpose integer arithmetic . . . 152
  4.26.2 General purpose arithmetic . . . 153
4.27 Misc arithmetic support predicates . . . 161
4.28 Built-in list operations . . . 161
4.29 Finding all Solutions to a Goal . . . 163
4.30 Forall . . . 164
4.31 Formatted Write . . . 164
  4.31.1 Writef . . . 164
  4.31.2 Format . . . 166
  4.31.3 Programming Format . . . 168
4.32 Terminal Control . . . 169
4.33 Operating System Interaction . . . 169
  4.33.1 Dealing with time and date . . . 172
  4.33.2 Controlling the `swipl-win.exe` console window . . . 177
4.34 File System Interaction . . . 178
4.35 User Top-level Manipulation . . . 182

4.36 Creating a Protocol of the User Interaction . . . 183
4.37 Debugging and Tracing Programs . . . 183
4.38 Obtaining Runtime Statistics . . . 186
4.39 Execution profiling . . . 186
  4.39.1 Profiling predicates . . . 188
  4.39.2 Visualizing profiling data . . . 188
  4.39.3 Information gathering . . . 189
4.40 Memory Management . . . 190
4.41 Windows DDE interface . . . 192
  4.41.1 DDE client interface . . . 192
  4.41.2 DDE server mode . . . 193
4.42 Miscellaneous . . . 194

5 Modules 196
5.1 Why Use Modules? . . . 196
5.2 Defining a Module . . . 196
5.3 Importing Predicates into a Module . . . 197
5.4 Defining a meta-predicate . . . 198
5.5 Overruling Module Boundaries . . . 200
5.6 Interacting with modules from the toplevel . . . 200
5.7 Composing modules from other modules . . . 201
5.8 Operators and modules . . . 201
5.9 Dynamic importing using import modules . . . 202
5.10 Reserved Modules and using the ‘user’ module . . . 203
5.11 An alternative import/export interface . . . 203
5.12 Dynamic Modules . . . 203
5.13 Transparent predicates: definition and context module . . . 204
5.14 Module properties . . . 205
5.15 Compatibility of the Module System . . . 206

6 Special Variables and Coroutining 208
6.1 Attributed variables . . . 208
  6.1.1 Attribute manipulation predicates . . . 210
  6.1.2 Attributed variable hooks . . . 210
  6.1.3 Operations on terms with attributed variables . . . 211
  6.1.4 Special purpose predicates for attributes . . . 211
6.2 Coroutining . . . 212
6.3 Global variables . . . 213
  6.3.1 Compatibility of SWI-Prolog Global Variables . . . 214

7 CHR: Constraint Handling Rules 215
7.1 Introduction . . . 215
7.2 Syntax and Semantics . . . 215
  7.2.1 Syntax . . . 215
  7.2.2 Semantics . . . 217
7.3 CHR in SWI-Prolog Programs . . . 218
  7.3.1 Embedding in Prolog Programs . . . 218

- 7.3.2 Constraint declaration . . . 219
- 7.3.3 Compilation . . . 222
- 7.4 Debugging . . . 222
  - 7.4.1 Ports . . . 222
  - 7.4.2 Tracing . . . 223
  - 7.4.3 CHR Debugging Predicates . . . 224
- 7.5 Examples . . . 225
- 7.6 Backwards Compatibility . . . 226
  - 7.6.1 The Old SICStus CHR implemenation . . . 226
  - 7.6.2 The Old ECLiPSe CHR implemenation . . . 227
- 7.7 Programming Tips and Tricks . . . 227
- 7.8 Compiler Errors and Warnings . . . 228
  - 7.8.1 CHR Compiler Errors . . . 228

8 Multi-threaded applications 230

- 8.1 Creating and destroying Prolog threads . . . 230
- 8.2 Monitoring threads . . . 233
- 8.3 Thread communication . . . 234
  - 8.3.1 Message queues . . . 234
  - 8.3.2 Signalling threads . . . 237
  - 8.3.3 Threads and dynamic predicates . . . 238
- 8.4 Thread synchronisation . . . 238
- 8.5 Thread support library(threadutil) . . . 240
  - 8.5.1 Debugging threads . . . 240
  - 8.5.2 Profiling threads . . . 241
- 8.6 Unbounded thread creation . . . 242
- 8.7 Multi-threaded mixed C and Prolog applications . . . 242
  - 8.7.1 A Prolog thread for each native thread (one-to-one) . . . 242
  - 8.7.2 Pooling Prolog engines (many-to-many) . . . 243
- 8.8 Multithreading and the XPCE graphics system . . . 244

9 Foreign Language Interface 246

- 9.1 Overview of the Interface . . . 246
- 9.2 Linking Foreign Modules . . . 246
  - 9.2.1 What linking is provided? . . . 247
  - 9.2.2 What kind of loading should I be using? . . . 247
  - 9.2.3 library(shlib): Utility library for loading foreign objects (DLLs, shared objects) 247
  - 9.2.4 Low-level operations on shared libraries . . . 249
  - 9.2.5 Static Linking . . . 250
- 9.3 Interface Data types . . . 251
  - 9.3.1 Type `term_t`: a reference to a Prolog term . . . 251
  - 9.3.2 Other foreign interface types . . . 252
- 9.4 The Foreign Include File . . . 253
  - 9.4.1 Argument Passing and Control . . . 253
  - 9.4.2 Atoms and functors . . . 256
  - 9.4.3 Analysing Terms via the Foreign Interface . . . 257
  - 9.4.4 Constructing Terms . . . 264

9.4.5 Unifying data . . . 267
9.4.6 Convenient functions to generate Prolog exceptions . . . 273
9.4.7 BLOBS: Using atoms to store arbitrary binary data . . . 275
9.4.8 Exchanging GMP numbers . . . 278
9.4.9 Calling Prolog from C . . . 279
9.4.10 Discarding Data . . . 281
9.4.11 Foreign Code and Modules . . . 282
9.4.12 Prolog exceptions in foreign code . . . 283
9.4.13 Catching Signals (Software Interrupts) . . . 285
9.4.14 Miscellaneous . . . 286
9.4.15 Errors and warnings . . . 289
9.4.16 Environment Control from Foreign Code . . . 289
9.4.17 Querying Prolog . . . 289
9.4.18 Registering Foreign Predicates . . . 289
9.4.19 Foreign Code Hooks . . . 293
9.4.20 Storing foreign data . . . 294
9.4.21 Embedding SWI-Prolog in other applications . . . 297
9.5 Linking embedded applications using swipl-ld . . . 301
9.5.1 A simple example . . . 303
9.6 The Prolog 'home' directory . . . 303
9.7 Example of Using the Foreign Interface . . . 305
9.8 Notes on Using Foreign Code . . . 307
9.8.1 Memory Allocation . . . 307
9.8.2 Compatibility between Prolog versions . . . 307
9.8.3 Debugging and profiling foreign code (valgrind) . . . 308
9.8.4 Name Conflicts in C modules . . . 308
9.8.5 Compatibility of the Foreign Interface . . . 308

10 Generating Runtime Applications 310
10.1 Limitations of qsave_program . . . 312
10.2 Runtimes and Foreign Code . . . 312
10.3 Using program resources . . . 313
10.3.1 Predicate Definitions . . . 314
10.3.2 The `swipl-rc` program . . . 315
10.4 Finding Application files . . . 315
10.4.1 Passing a path to the application . . . 316

A The SWI-Prolog library 317
A.1 library(aggregate): Aggregation operators on backtrackable predicates . . . 317
A.2 library(apply): Apply predicates on a list . . . 320
A.3 `assoc`: Association lists . . . 320
A.4 `broadcast`: Broadcast and receive event notifications . . . 322
A.5 library(charsio): I/O on Lists of Character Codes . . . 324
A.6 `check`: Elementary completeness checks . . . 325
A.7 library(clpfd): Constraint Logic Programming over Finite Domains . . . 326
A.8 `clpqr`: Constraint Logic Programming over Rationals and Reals . . . 340
A.8.1 Solver predicates . . . 341

A.8.2 Syntax of the predicate arguments 342
A.8.3 Use of unification 342
A.8.4 Non-linear constraints 343
A.8.5 Status and known problems 343
A.9 library(csv): Process CSV (Comma-Separated Values) data 344
A.10 library(debug): Print debug messages and test assertions 345
A.11 `gensym`: Generate unique identifiers 347
A.12 library(lists): List Manipulation 347
A.13 `nb_set`: Non-backtrackable set 352
A.14 `www_browser`: Activating your Web-browser 353
A.15 library(option): Option list processing 353
A.16 library(optparse): command line parsing 355
A.16.1 Notes and tips 359
A.17 library(ordsets): Ordered set manipulation 361
A.18 library(pairs): Operations on key-value lists 363
A.19 `pio`: Pure I/O 364
A.19.1 library(pure_input): Pure Input from files 364
A.20 `predicate_options`: Declare option-processing of predicates 365
A.20.1 The strength and weakness of predicate options 365
A.20.2 Options as arguments or environment? 366
A.20.3 Improving on the current situation 366
A.21 `prolog_xref`: Cross-reference data collection library 369
A.21.1 Extending the library 370
A.22 library(random): Random numbers 370
A.23 `readutil`: Reading lines, streams and files 372
A.24 `record`: Access named fields in a term 373
A.25 `registry`: Manipulating the Windows registry 375
A.26 `simplex`: Solve linear programming problems 376
A.26.1 Example 1 378
A.26.2 Example 2 378
A.26.3 Example 3 379
A.27 library(thread_pool): Resource bounded thread management 380
A.28 `ugraphs`: Unweighted Graphs 382
A.29 library(url): Analysing and constructing URL 385
A.30 library(varnumbers): Utilities for numbered terms 387

B Hackers corner **389**
B.1 Examining the Environment Stack 389
B.2 Intercepting the Tracer 391
B.3 Adding context to errors: prolog_exception_hook 393
B.4 Hooks using the exception predicate 393
B.5 Hooks for integrating libraries 394
B.6 Hooks for loading files 395
B.7 Readline Interaction 395

C Compatibility with other Prolog dialects **397**
C.1 Some considerations for writing portable code 398

D Glossary of Terms 401

E SWI-Prolog License Conditions and Tools 407
- E.1 The SWI-Prolog kernel and foreign libraries . . . 407
  - E.1.1 The SWI-Prolog Prolog libraries . . . 407
- E.2 Contributing to the SWI-Prolog project . . . 408
- E.3 Software support to keep track of license conditions . . . 408
- E.4 License conditions inherited from used code . . . 409
  - E.4.1 Cryptographic routines . . . 409

F Summary 411
- F.1 Predicates . . . 411
- F.2 Library predicates . . . 425
  - F.2.1 `aggregate` . . . 425
  - F.2.2 `apply` . . . 425
  - F.2.3 `assoc` . . . 425
  - F.2.4 `broadcast` . . . 425
  - F.2.5 `charsio` . . . 426
  - F.2.6 `check` . . . 426
  - F.2.7 `csv` . . . 426
  - F.2.8 `lists` . . . 426
  - F.2.9 `debug` . . . 427
  - F.2.10 `option` . . . 427
  - F.2.11 `optparse` . . . 428
  - F.2.12 `ordsets` . . . 428
  - F.2.13 `predicate_options` . . . 428
  - F.2.14 `prologxref` . . . 429
  - F.2.15 `pairs` . . . 429
  - F.2.16 `pio` . . . 429
  - F.2.17 `random` . . . 429
  - F.2.18 `readutil` . . . 430
  - F.2.19 `record` . . . 430
  - F.2.20 `registry` . . . 430
  - F.2.21 `ugraphs` . . . 430
  - F.2.22 `url` . . . 431
  - F.2.23 `www_browser` . . . 431
  - F.2.24 `clp/clpfd` . . . 431
  - F.2.25 `clpqr` . . . 432
  - F.2.26 `clp/simplex` . . . 432
  - F.2.27 `thread_pool` . . . 432
  - F.2.28 `varnumbers` . . . 433
- F.3 Arithmetic Functions . . . 434
- F.4 Operators . . . 436

Introduction 1

1.1 SWI-Prolog

SWI-Prolog started back in 1986 with the requirement for a Prolog that could handle recursive interaction with the C-language: Prolog calling C and C calling Prolog recursively. Those days Prolog systems were very aware of its environment and we needed such a system to support interactive applications. Since then, SWI-Prolog's development has been guided by requests from the user community, especially focussing on (in arbitrary order) interaction with the environment, scalability, (I/O) performance, standard compliance, teaching and the program development environment.

SWI-Prolog is based on a very simple Prolog virtual machine called ZIP [Bowen *et al.*, 1983, Neumerkel, 1993] which defines only 7 instructions. Prolog can easily be compiled into this language and the abstract machine code is easily decompiled back into Prolog. As it is also possible to wire a standard 4-port debugger in the virtual machine there is no need for a distinction between compiled and interpreted code. Besides simplifying the design of the Prolog system itself this approach has advantages for program development: the compiler is simple and fast, the user does not have to decide in advance whether debugging is required and the system only runs slightly slower when in debug mode. The price we have to pay is some performance degradation (taking out the debugger from the VM interpreter improves performance by about 20%) and somewhat additional memory usage to help the decompiler and debugger.

SWI-Prolog extends the minimal set of instructions described in [Bowen *et al.*, 1983] to improve performance. While extending this set care has been taken to maintain the advantages of decompilation and tracing of compiled code. The extensions include specialised instructions for unification, predicate invocation, some frequently used built-in predicates, arithmetic, and control (`;/2`, `|/2`), if-then (`->/2`) and negation-by-failure (`\+/1`).

1.1.1 Books about Prolog

This manual does not describe the full syntax and semantics of Prolog, nor how one should write a program in Prolog. These subjects have been described extensively in the literature. See [Bratko, 1986], [Sterling & Shapiro, 1986], and [Clocksin & Melish, 1987]. For more advanced Prolog material see [O'Keefe, 1990]. Syntax and standard operator declarations conform to the 'Edinburgh standard'. Most built-in predicates are compatible with those described in [Clocksin & Melish, 1987]. SWI-Prolog also offers a number of primitive predicates compatible with Quintus Prolog[1] [Qui, 1997] and BIM_Prolog[2] [BIM, 1989].

ISO compliant predicates are based on "Prolog: The Standard", [Deransart *et al.*, 1996], validated using [Hodgson, 1998].

---

[1] Quintus is a trademark of Quintus Computer Systems Inc., USA

[2] BIM is a trademark of BIM sa/nv., Belgium

1.2 Status

This manual describes version 6.0 of SWI-Prolog. SWI-Prolog has been used now for many years. The application range includes Prolog course material, meta-interpreters, simulation of parallel Prolog, learning systems, natural language processing, complex interactive systems, web-server and web-server components. Although in our experience rather obvious and critical bugs can remain unnoticed for a remarkably long period, we assume the basic Prolog system is fairly stable. Bugs can be expected in infrequently used built-in predicates.

Some bugs are known to the author. They are described as footnotes in this manual.

1.3 Compliance to the ISO standard

SWI-Prolog 3.3.0 implements all predicates described in "Prolog: The Standard" [Deransart *et al.*, 1996].

Exceptions and warnings are still weak. Some SWI-Prolog predicates silently fail on conditions where the ISO specification requires an exception (`functor/3` for example). Some predicates print warnings rather than raising an exception. All predicates where exceptions may be caused due to a correct program operating in an imperfect world (I/O, arithmetic, resource overflows) should behave according to the ISO standard. In other words: SWI-Prolog should be able to execute any program conforming to [Deransart *et al.*, 1996] that does not rely on exceptions generated by errors in the program.

1.4 Should you be using SWI-Prolog?

There are a number of reasons why you better choose a commercial Prolog system, or another academic product:

- *SWI-Prolog is not supported*
  Although I usually fix bugs shortly after a bug report arrives, I cannot promise anything. Now that the sources are provided, you can always dig into them yourself.

- *Memory requirements and performance are your first concerns*
  A number of commercial compilers are more keen on memory and performance than SWI-Prolog. I do not wish to sacrifice some of the nice features of the system, nor its portability to compete on raw performance.

- *You need features not offered by SWI-Prolog*
  In this case you may wish to give me suggestions for extensions. If you have great plans, please contact me (you might have to implement them yourself however).

On the other hand, SWI-Prolog offers some nice facilities:

- *Nice environment*
  This includes 'Do What I Mean', automatic completion of atom names, history mechanism and a tracer that operates on single key-strokes. Interfaces to some standard editors are provided (and can be extended), as well as a facility to maintain programs (see `make/0`).

- *Very fast compiler*
  Even very large applications can be loaded in seconds on most machines. If this is not enough, there is a Quick Load Format that is slightly more compact and loading is almost always I/O bound.

- *Transparent compiled code*
  SWI-Prolog compiled code can be treated just as interpreted code: you can list it, trace it, etc. This implies you do not have to decide beforehand whether a module should be loaded for debugging or not. Also, performance is much better than the performance of most interpreters.

- *Profiling*
  SWI-Prolog offers tools for performance analysis, which can be very useful to optimise programs. Unless you are very familiar with Prolog and Prolog performance considerations this might be more helpful than a better compiler without these facilities.

- *Flexibility*
  SWI-Prolog can easily be integrated with C, supporting non-determinism in Prolog calling C as well as C calling Prolog (see section 9). It can also be *embedded* in external programs (see section 9.5). System predicates can be redefined locally to provide compatibility with other Prolog systems.

- *Integration with XPCE*
  SWI-Prolog offers a tight integration to the Object Oriented Package for User Interface Development, called XPCE [Anjewierden & Wielemaker, 1989]. XPCE allows you to implement graphical user interfaces that are source-code compatible over Unix/X11, Windows and MacOS X using X11.

1.5 The XPCE GUI system for Prolog

The XPCE GUI system for dynamically typed languages has been with SWI-Prolog for a long time. It is developed by Anjo Anjewierden and Jan Wielemaker from the department of SWI, University of Amsterdam. It aims at a high-productive development environment for graphical applications based on Prolog.

Object oriented technology has proven to be a suitable model for implementing GUIs, which typically deal with things Prolog is not very good at: event-driven control and global state. With XPCE, we designed a system that has similar characteristics that make Prolog such a powerful tool: dynamic typing, meta-programming and dynamic modification of the running system.

XPCE is an object-system written in the C-language. It provides for the implementation of methods in multiple languages. New XPCE classes may be defined from Prolog using a simple, natural syntax. The body of the method is executed by Prolog itself, providing a natural interface between the two systems. Below is a very simple class definition.

```
:- pce_begin_class(prolog_lister, frame,
                   "List Prolog predicates").

initialise(Self) :->
        "As the C++ constructor"::
        send_super(Self, initialise, 'Prolog Lister'),
        send(Self, append, new(D, dialog)),
        send(D, append,
             text_item(predicate, message(Self, list, @arg1))),
        send(new(view), below, D).

list(Self, From:name) :->
        "List predicates from specification"::
        (   catch(term_to_atom(Term, From), _, fail)
        ->  get(Self, member, view, V),
            current_output(Old),
            pce_open(V, write, Fd),
            set_output(Fd),
            listing(Term),
            close(Fd),
            set_output(Old)
        ;   send(Self, report, error, 'Syntax error')
        ).

:- pce_end_class.

test :- send(new(prolog_lister), open).
```

Its 165 built-in classes deal with the meta-environment, data-representation and—of course—graphics. The graphics classes concentrate on direct-manipulation of diagrammatic representations.

Availability. XPCE runs on most Unix^{tm} platforms, Windows 95/98/ME, Windows NT/2000/XP and MacOS X (using X11). In the past, versions for Quintus- and SICStus Prolog as well as some Lisp dialects have existed. After discontinuing active Lisp development at SWI the Lisp versions have died. Active development on the Quintus and SICStus versions has been stopped due to lack of standardisation in the Prolog community. If adequate standards emerge we are happy to actively support other Prolog implementations.

Info. further information is available from `http://www.swi-prolog.org/packages/xpce/` or by E-mail to `info@www.swi-prolog.org`.

1.6 Release Notes

Collected release-notes. This section only contains some highlights. Smaller changes to especially older releases have been removed. For a complete log, see the file `ChangeLog` from the distribution.

Version 1.8 Release Notes

Version 1.8 offers a stack-shifter to provide dynamically expanding stacks on machines that do not offer operating-system support for implementing dynamic stacks.

Version 1.9 Release Notes

Version 1.9 offers better portability including an MS-Windows 3.1 version. Changes to the Prolog system include:

- *Redefinition of system predicates*
  Redefinition of system predicates was allowed silently in older versions. Version 1.9 only allows it if the new definition is headed by a :- `redefine_system_predicate/1` directive.

- *Top-level 'answer' reuse*
  The top-level maintains a table of bindings returned by top-level goals and allows for reuse of these bindings by prefixing the variables with the $ sign. See section 2.8.

- *Better source code administration*
  Allows for proper updating of multifile predicates and finding the sources of individual clauses.

Version 2.0 Release Notes

New features offered:

- *32-bit Virtual Machine*
  Removes various limits and improves performance.

- *Inline foreign functions*
  'Simple' foreign predicates no longer build a Prolog stack-frame, but are directly called from the VM. Notably provides a speedup for the test predicates such as `var/1`, etc.

- *Various compatibility improvements*

- *Stream based I/O library*
  All SWI-Prolog's I/O is now handled by the stream-package defined in the foreign include file `SWI-Stream.h`. Physical I/O of Prolog streams may be redefined through the foreign language interface, facilitating much simpler integration in window environments.

Version 2.5 Release Notes

Version 2.5 is an intermediate release on the path from 2.1 to 3.0. All changes are to the foreign-language interface, both to user- and system-predicates implemented in the C-language. The aim is twofold. First of all to make garbage-collection and stack-expansion (stack-shifts) possible while foreign code is active without the C-programmer having to worry about locking and unlocking C-variables pointing to Prolog terms. The new approach is closely compatible with the Quintus and SICStus Prolog foreign interface using the `+term` argument specification (see their respective manuals). This allows for writing foreign interfaces that are easily portable over these three Prolog platforms.

Apart from various bug fixes listed in the ChangeLog file, these are the main changes since 2.1.0:

- *ISO compatibility*
  Many ISO compatibility features have been added: `open/4`, arithmetic functions, syntax, etc.

- *Win32*
  Many fixes for the Win32 (NT, '95 and win32s) platforms. Notably many problems related to pathnames and a problem in the garbage collector.

- *Performance*
  Many changes to the clause indexing system: added hash-tables, lazy computation of the index information, etc.

- *Portable saved-states*
  The predicate `qsave_program/[1,2]` allows for the creation of machine independent saved-states that load very quickly.

Version 2.6 Release Notes

Version 2.6 provides a stable implementation of the features added in the 2.5.x releases, but at the same time implements a number of new features that may have impact on the system stability.

- *32-bit integer and double float arithmetic*
  The biggest change is the support for full 32-bit signed integers and machin-nformac
  prcirsion floatrsThe intrnal reprsmentation the
  intrfg on the arithmetic uncatiosy hsy changnd for tis(.)]TJ0 g 0 G/F22 10.9091 Tf -10.909 -23.96 Td [( )]TJ/F41 1
  TheWin32 vehsiogn signised.STheenelnwimplemenWin32
  that may inppldicatioss.TP providdn, oey for widoe-beasnd
  and oey on ts a Win32 cnsoble ppldicatio(.)]TJ0 g 0 G/F22 10.9091 Tf -10.909 -23.96 Td [( )]TJ/F41 10.9091 Tf 1

Version 2.9 Release Notes

Version 2.9 is the next step towards version 3.0, improving ISO compliance and introducing ISO compliant exception handling. New are `catch/3`, `throw/1`, `abolish/1`, `write_term/[2,3]`, `write_canonical/[1,2]` and the C-functions `PL_exception()` and `PL_throw()`. The predicates `display/[1,2]` and `displayq/[1,2]` have been moved to `backcomp`, so old code referring to them will autoload them.

- `!/0`, `call/1`
  The cut now behaves according to the ISO standard. This implies it works in compound goals passed to `call/1` and is local to the *condition* part of if-then-else as well as the argument of `\+/1`.

- *atom_chars/2*
  This predicate is now ISO compliant and thus generates a list of one-character atoms. The behaviour of the old predicate is available in the —also ISO compliant— `atom_codes/2` predicate. Safest repair is a replacement of all `atom_chars` into `atom_codes`. If you do not want to change any source-code, you might want to use

  ```
  user:goal_expansion(atom_chars(A,B), atom_codes(A,B)).
  ```

- *number_chars/2*
  Same applies for `number_chars/2` and `number_codes/2`.

- `feature/2`, `set_feature/2`
  These are replaced by the ISO compliant `current_prolog_flag/2` and `set_prolog_flag/2`. The library `backcomp` provides definitions for these predicates, so no source **must** be updated.

- *Accessing command-line arguments*
  This used to be provided by the undocumented '$argv'/1 and Quintus compatible library `unix/1`. Now there is also documented `current_prolog_flag`(*argv, Argv*).

- *dup_stream/2*
  Has been deleted. New stream-aliases can deal with most of the problems for which `dup_stream/2` was designed and `dup/2` from the *clib* package can with most others.

- *op/3*
  Operators are now **local to modules**. This implies any modification of the operator-table does not influence other modules. This is consistent with the proposed ISO behaviour and a necessity to have any usable handling of operators in a multi-threaded environment.

- *set_prolog_flag(character_escapes, Bool)*
  This Prolog flag is now an interface to changing attributes on the current source-module, effectively making this flag module-local as well. This is required for consistent handling of sources written with ISO (obligatory) character-escape sequences together with old Edinburgh code.

- *current_stream/3 and stream_position*
  These predicates have been moved to `quintus`.

Version 3.4 Release Notes

The 3.4 release is a consolidation release. It consolidates the improvements and standard conformance of the 3.3 releases. This version is closely compatible with the 3.3 version except for one important change:

- *Argument order in* `select/3`
  The list-processing predicate `select/3` somehow got into a very early version of SWI-Prolog with the wrong argument order. This has been fixed in 3.4.0. The correct order is select(?Elem, ?List, ?Rest).

  As `select/3` has no error conditions, runtime checking cannot be done. To simplify debugging, the library module `checkselect` will print references to `select/3` in your source code and install a version of select that enters the debugger if select is called and the second argument is not a list.

  This library can be loaded explicitly or by calling `check_old_select/0`.

Version 4.0 Release Notes

As of version 4.0 the standard distribution of SWI-Prolog is bundled with a number of its popular extension packages, among which the now open source XPCE GUI toolkit (see section 1.5). No significant changes have been made to the basic SWI-Prolog engine.

Some useful tricks in the integrated environment:

- *Register the GUI tracer*
  Using a call to `guitracer/0`, hooks are installed that replace the normal command-line driven tracer with a graphical front-end.

- *Register PceEmacs for editing files*
  From your initialisation file. you can load `emacs/swi_prolog` that cause `edit/1` to use the built-in PceEmacs editor.

Version 5.0 Release Notes

Version 5.0 marks a breakpoint in the philosophy, where SWI-Prolog moves from a dual GPL/proprietary to a uniform LGPL (Lesser GNU Public Licence) schema, providing a widely usable Free Source Prolog implementation.

On the technical site the development environment, consisting of source-level debugger, integrated editor and various analysis and navigation tools progress steadily towards a mature set of tools.

Many portability issues have been improved, including a port to MacOS X (Darwin).

For details, please visit the new website at `http://www.swi-prolog.org`

Version 5.1 Release Notes

Version 5.1 is a beta-series introducing portable multi-threading. See chapter 8. In addition it introduces many new facilities to support server applications, such as the new `rlimit` library to limit system resources and the possibility to set timeouts on input streams.

Version 5.2 Release Notes

Version 5.2 consolidates the 5.1.x beta series that introduced threading and many related modifications to the kernel.

Version 5.3 Release Notes

Version 5.3.x is a development series for adding coroutining, constraints, global variables, cyclic terms (infinite trees) and other goodies to the kernel. The package JPL, providing a bidirectional Java/Prolog interface is added to the common source-tree and common binary packages.

Version 5.4 Release Notes

Version 5.4 consolidates the 5.3.x beta series.

Version 5.5 Release Notes

Version 5.5.x provides support for *wide characters* with UTF-8 and UNICODE I/O (section 2.18.1). On both 32 and 64-bit hardware Prolog integers are now at minimum 64-bit integers. If available, SWI-Prolog arithmetic uses the GNU GMP library to provide *unbounded* integer arithmetic as well as rational arithmetic. Adding GMP support is sponsored by Scientific Software and Systems Limited, `www.sss.co.nz`. This version also incorporates clp(r) by Christian Holzbaur, brought to SWI-Prolog by Tom Schrijvers and Leslie De Koninck (section A.8).

Version 5.6 Release Notes

Version 5.6 consolidates the 5.5.x beta series.

Version 5.7 Release Notes

The aim of the 5.7 series is to clean up much of the system. Notably, the virtual machine has a much simpler setup that makes it much easier to add new instructions. This facility has been exploited to enhance performance and provide proper support for the `meta_predicate/1` directive for enhanced portability.

Version 5.10 Release Notes

The 5.9 series has enhanced SWI-Prolog in terms of memory management, scalability and robustness. Notably, threads are much cheaper and now limited in count only by the OS. Database and stream-handles have become safe. Compatibility to YAP and SISCtus has been improved.

1.7 Donate to the SWI-Prolog project

If you are happy with SWI-Prolog and you care that it stays around longer while it becomes faster, more stable and with more features, you should consider donating to the SWI-Prolog foundation. Please visit the page below.

```
http://www.swi-prolog.org/donate.html
```

1.8 Acknowledgements

Overview 2

2.1 Getting started quickly

2.1.1 Starting SWI-Prolog

Starting SWI-Prolog on Unix

By default, SWI-Prolog is installed as 'swipl'. The command-line arguments of SWI-Prolog itself and its utility programs are documented using standard Unix `man` pages. SWI-Prolog is normally operated as an interactive application simply by starting the program:

```
machine% swipl
Welcome to SWI-Prolog (Version \plversion)
...

1 ?-
```

After starting Prolog, one normally loads a program into it using `consult/1`, which — for historical reasons — may be abbreviated by putting the name of the program file between square brackets. The following goal loads the file `likes.pl` containing clauses for the predicates `likes/2`:

```
?- [likes].
% likes compiled, 0.00 sec, 596 bytes.

Yes
?-
```

After this point, Unix and Windows users unite, so if you are using Unix please continue at section 2.1.2.

Starting SWI-Prolog on Windows

After SWI-Prolog has been installed on a Windows system, the following important new things are available to the user:

- A folder (called *directory* in the remainder of this document) called `pl` containing the executables, libraries, etc. of the system. No files are installed outside this directory.
- A program `swipl-win.exe`, providing a window for interaction with Prolog. The program `swipl.exe` is a version of SWI-Prolog that runs in a DOS-box.

- The file-extension `.pl` is associated with the program `swipl-win.exe`. Opening a `.pl` file will cause `swipl-win.exe` to start, change directory to the directory in which the file-to-open resides and load this file.

The normal way to start with the `likes.pl` file mentioned in section 2.1.1 is by simply double-clicking this file in the Windows explorer.

2.1.2 Executing a query

After loading a program, one can ask Prolog queries about the program. The query below asks Prolog what food 'sam' likes. The system responds with `X =` ⟨*value*⟩ if it can prove the goal for a certain *X*. The user can type the semi-colon (;)[1] if (s)he wants another solution, or RETURN if (s)he is satisfied, after which Prolog will say **Yes**. If Prolog answers **No**, it indicates it cannot find any (more) answers to the query. Finally, Prolog can answer using an error message to indicate the query or program contains an error.

```
?- likes(sam, X).

X = dahl ;

X = tandoori ;

...

X = chips ;

No
?-
```

2.2 The user's initialisation file

After the necessary system initialisation the system consults (see `consult/1`) the user's startup file. The base-name of this file follows conventions of the operating system. On MS-Windows, it is the file `pl.ini` and on Unix systems `.plrc`. The file is searched using the `file_search_path/2` clauses for `user_profile`. The table below shows the default value for this search-path. The phrase ⟨*appdata*⟩ refers to the Windows CSIDL name for the folder. The actual name depends on the Windows language. English versions typically use `ApplicationData`. See also `win_folder/2`

	Unix	**Windows**
local	.	.
home	~	⟨*appdata*⟩/SWI-Prolog

After the first startup file is found it is loaded and Prolog stops looking for further startup files. The name of the startup file can be changed with the '`-f` *`file`*' option. If *File* denotes an absolute path,

---

[1] On most installations, single-character commands are executed without waiting for the RETURN key.

this file is loaded, otherwise the file is searched for using the same conventions as for the default startup file. Finally, if *file* is `none`, no file is loaded.

See also the `-s` (script) and `-F` (system-wide initialisation) in section 2.4 and section 2.3.

2.3 Initialisation files and goals

Using command-line arguments (see section 2.4), SWI-Prolog can be forced to load files and execute queries for initialisation purposes or non-interactive operation. The most commonly used options are `-f` *`file`* or `-s` *`file`* to make Prolog load a file, `-g` *`goal`* to define an initialisation goal and `-t` *`goal`* to define the *top-level goal*. The following is a typical example for starting an application directly from the command-line.

```
machine% swipl -s load.pl -g go -t halt
```

It tells SWI-Prolog to load `load.pl`, start the application using the *entry-point* `go/0` and —instead of entering the interactive top-level— exit after completing `go/0`. The `-q` may be used to suppress all informational messages.

In MS-Windows, the same can be achieved using a short-cut with appropriately defined command-line arguments. A typically seen alternative is to write a file `run.pl` with content as illustrated below. Double-clicking `run.pl` will start the application.

```
:- [load].                    % load program
:- go.                        % run it
:- halt.                      % and exit
```

Section 2.10.2 discusses further scripting options and chapter 10 discusses the generation of runtime executables. Runtime executables are a means to deliver executables that do not require the Prolog system.

2.4 Command-line options

The full set of command-line options is given below:

`--arch`
When given as the only option, it prints the architecture identifier (see Prolog flag `arch`) and exits. See also `-dump-runtime-variables`. Also available as `-arch`.

`--dump-runtime-variables`
When given as the only option, it prints a sequence of variable settings that can be used in shell-scripts to deal with Prolog parameters. This feature is also used by `swipl-ld` (see section 9.5). Below is a typical example of using this feature. Also available as `-dump-runtime-variables`.

```
eval `swipl --dump-runtime-variables`
cc -I$PLBASE/include -L$PLBASE/lib/$PLARCH ...
```

The option can be followed by `=sh` to dump in POSIX shell format (default) or `cmd` to dump in MS-Windows `cmd.exe` compatible format.

--help
When given as the only option, it summarises the most important options. Also available as `-h` and `-help`.

--home=DIR
Use DIR as home directory. See section 9.6 for details.

--quiet
Set the Prolog flag `verbose` to `silent`, suppressing informational and banner messages. Also available as `-q`.

--nodebug
Disable debugging. See the `current_prolog_flag/2` flag `generate_debug_info` for details.

--nosignals
Inhibit any signal handling by Prolog, a property that is sometimes desirable for embedded applications. This option sets the flag `signals` to `false`. See section 9.4.21 for details.

-tty
Unix only. Switches controlling the terminal for allowing single-character commands to the tracer and `get_single_char/1`. By default manipulating the terminal is enabled unless the system detects it is not connected to a terminal or it is running as a GNU-Emacs inferior process. This flag is sometimes required for smooth interaction with other applications.

--version
When given as the only option, it summarises the version and the architecture identifier. Also available as `-v`.

--win_app
This option is available only in `swipl-win.exe` and is used for the start-menu item. If causes `plwin` to start in the folder `...\My Documents\Prolog` or local equivalent thereof (see `win_folder/2`). The `Prolog` subdirectory is created if it does not exist.

--
Stops scanning for more arguments, so you can pass arguments for your application after this one. See `current_prolog_flag/2` using the flag `argv` for obtaining the command-line arguments.

2.4.1 Controlling the stack-sizes

As of SWI-Prolog 5.9.8, the default limit for the stack-sizes is 128Mb on 32-bit and 256Mb on 64-bit hardware. The 128Mb limit on 32-bit system is the highest possible value and this option can thus only be used to lower the limit. On 64-bit systems, the limit can both be reduced and enlarged. See section 2.19. Here are two examples, the first reducing the local stack limit to catch unbounded recursion really quickly and the second using a really big (32Gb) global limit on a 64-bit machine:

```
$ swipl -L8m
$ swipl -G32g
```

-G*size[kmg]*

Limit for the global stack (sometimes also called *term-stack* or *heap*). This is where compound terms and large numbers live.

-L*size[kmg]*

Limit for the local stack ((sometimes also called *environment-stack*). This is where environments and choice-points live.

-T*size[kmg]*

Limit for the trail stack. This is where we keep track of assignments, so we can rollback on backtracking or exceptions.

2.4.2 Running goals from the commandline

-g *goal*

Goal is executed just before entering the top level. Default is a predicate which prints the welcome message. The welcome message can be suppressed with `--quiet`, but also with `-g true`. *goal* can be a complex term. In this case quotes are normally needed to protect it from being expanded by the shell. A safe way to run a goal non-interactively is here:

```
% swipl <options> -g go,halt -t 'halt(1)'
```

-t *goal*

Use *goal* as interactive top-level instead of the default goal `prolog/0`. *goal* can be a complex term. If the top-level goal succeeds SWI-Prolog exits with status 0. If it fails the exit status is 1. If the toplevel raises an exception, this is printed as an uncaught error and the toplevel is restarted. This flag also determines the goal started by `break/0` and `abort/0`. If you want to stop the user from entering interactive mode start the application with ‘`-g goal`’ and give ‘halt’ as top-level.

2.4.3 Compiler options

-c *file . . .*

Compile files into an ‘intermediate code file’. See section 2.10.

-o *output*

Used in combination with `-c` or `-b` to determine output file for compilation.

-O

Optimised compilation. See `current_prolog_flag/2` flag `optimise` for details.

-s *file*

Use *file* as a script-file. The script file is loaded after the initialisation file specified with the `-f file` option. Unlike `-f file`, using `-s` does not stop Prolog from loading the personal initialisation file.

-f *file*

Use *file* as initialisation file instead of the default `.plrc` (Unix) or `pl.ini` (Windows). '`-f none`' stops SWI-Prolog from searching for a startup file. This option can be used as an alternative to `-s file` that stops Prolog from loading the personal initialisation file. See also section 2.2.

-F *script*

Selects a startup-script from the SWI-Prolog home directory. The script-file is named ⟨*script*⟩`.rc`. The default *script* name is deduced from the executable, taking the leading alphanumerical characters (letters, digits and underscore) from the program-name. `-F none` stops looking for a script. Intended for simple management of slightly different versions. One could for example write a script `iso.rc` and then select ISO compatibility mode using `pl -F iso` or make a link from `iso-pl` to `pl`.

-x *bootfile*

Boot from *bootfile* instead of the system's default boot file. A bootfile is a file resulting from a Prolog compilation using the `-b` or `-c` option or a program saved using `qsave_program/[1,2]`.

-p *alias=path1[:path2 . . .]*

Define a path alias for file_search_path. *alias* is the name of the alias, *path1* ... is a list of values for the alias. On Windows the list-separator is `;`. On other systems it is `:`. A value is either a term of the form alias(value) or pathname. The computed aliases are added to `file_search_path/2` using `asserta/1`, so they precede predefined values for the alias. See `file_search_path/2` for details on using this file-location mechanism.

2.4.4 Maintenance options

The following options are for system maintenance. They are given for reference only.

-b *initfile* . . . `-c` *file* . . .

Boot compilation. *initfile* . . . are compiled by the C-written bootstrap compiler, *file* . . . by the normal Prolog compiler. System maintenance only.

-d *level*

Set debug level to *level*. Only has effect if the system is compiled with the `-DO_DEBUG` flag. System maintenance only.

2.5 GNU Emacs Interface

The default Prolog mode for GNU-Emacs can be activated by adding the following rules to your Emacs initialisation file:

```
(setq auto-mode-alist
      (append
       '(("\\.pl" . prolog-mode))
       auto-mode-alist))
(setq prolog-program-name "swipl")
```

```
(setq prolog-consult-string "[user].\n")
;If you want this.  Indentation is either poor or I don't use
;it as intended.
;(setq prolog-indent-width 8)
```

Unfortunately the default Prolog mode of GNU-Emacs is not very good. There are several alternatives though:

- `http://turing.ubishops.ca/home/bruda/emacs-prolog/`
- `http://stud4.tuwien.ac.at/ e0225855/ediprolog/ediprolog.html`
- `http://stud4.tuwien.ac.at/ e0225855/pceprolog/pceprolog.html`
- `http://stud4.tuwien.ac.at/ e0225855/etrace/etrace.html`

2.6 Online Help

Online help provides a fast lookup and browsing facility to this manual. The online manual can show predicate definitions as well as entire sections of the manual.

The online help is displayed from the file `'MANUAL'`. The file `helpidx` provides an index into this file. `'MANUAL'` is created from the LaTeX sources with a modified version of `dvitty`, using overstrike for printing bold text and underlining for rendering italic text. XPCE is shipped with `swi_help`, presenting the information from the online help in a hypertext window. The Prolog flag `write_help_with_overstrike` controls whether or not `help/1` writes its output using overstrike to realise bold and underlined output or not. If this Prolog flag is not set it is initialised by the help library to `true` if the `TERM` variable equals `xterm` and `false` otherwise. If this default does not satisfy you, add the following line to your personal startup file (see section 2.2):

```
:- set_prolog_flag(write_help_with_overstrike, true).
```

help

Equivalent to `help(help/1)`.

help(+*What*)

Show specified part of the manual. *What* is one of:

⟨*Name*⟩/⟨*Arity*⟩	Give help on specified predicate
⟨*Name*⟩	Give help on named predicate with any arity or C interface function with that name
⟨*Section*⟩	Display specified section. Section numbers are dash-separated numbers: `2-3` refers to section 2.3 of the manual. Section numbers are obtained using `apropos/1`.

Examples:

`?- help(assert).`	Give help on predicate assert
`?- help(3-4).`	Display section 3.4 of the manual
`?- help('PL_retry').`	Give help on interface function `PL_retry()`

`!!.`	Repeat last query
`!nr.`	Repeat query numbered ⟨*nr*⟩
`!str.`	Repeat last query starting with ⟨*str*⟩
`h.`	Show history of commands
`!h.`	Show this list

Table 2.1: History commands

See also `apropos/1` and the SWI-Prolog home page at `http://www.swi-prolog.org`, which provides a FAQ, an HTML version of the manual for online browsing, and HTML and PDF versions for downloading.

apropos(+*Pattern***)**

Display all predicates, functions and sections that have *Pattern* in their name or summary description. Lowercase letters in *Pattern* also match a corresponding uppercase letter. Example:

`?- apropos(file).`	Display predicates, functions and sections that have 'file' (or 'File', etc.) in their summary description.

explain(+*ToExplain***)**

Give an explanation on the given 'object'. The argument may be any Prolog data object. If the argument is an atom, a term of the form *Name/Arity* or a term of the form *Module:Name/Arity*, `explain/1` describes the predicate as well as possible references to it. See also `gxref/0`.

explain(+*ToExplain, -Explanation***)**

Unify *Explanation* with an explanation for *ToExplain*. Backtracking yields further explanations.

2.7 Command-line history

SWI-Prolog offers a query substitution mechanism called 'history'. The availability of this feature is controlled by `set_prolog_flag/2`, using the `history` Prolog flag. By default, history is available if the Prolog flag `readline` is `false`. To enable this feature, remembering the last 50 commands, put the following into your startup file (see section 2.2):

```
:- set_prolog_flag(history, 50).
```

The history system allows the user to compose new queries from those typed before and remembered by the system. The available history commands are shown in table 2.1. History expansion is not done if these sequences appear in quoted atoms or strings.

2.8 Reuse of top-level bindings

Bindings resulting from the successful execution of a top-level goal are asserted in a database. These values may be reused in further top-level queries as $Var. Only the latest binding is available. Example:

Note that variables may be set by executing `=/2`:

```
1 ?- maplist(plus(1), "hello", X).

X = [105,102,109,109,112]

Yes
2 ?- format(’˜s˜n’, [$X]).
ifmmp

Yes
3 ?-
```

Figure 2.1: Reusing top-level bindings

```
6 ?- X = statistics.

X = statistics

Yes
7 ?- $X.
28.00 seconds cpu time for 183,128 inferences
4,016 atoms, 1,904 functors, 2,042 predicates, 52 modules
55,915 byte codes; 11,239 external references

                      Limit    Allocated       In use
Heap         :                                624,820 Bytes
Local  stack :    2,048,000        8,192          404 Bytes
Global stack :    4,096,000       16,384          968 Bytes
Trail  stack :    4,096,000        8,192          432 Bytes

Yes
8 ?-
```

2.9 Overview of the Debugger

SWI-Prolog has a 6-port tracer, extending the standard 4-port Byrd box model tracer [Byrd, 1980, Clocksin & Melish, 1987] with two additional ports. The optional *unify* port allows the user to inspect the result after unification of the head. The *exception* port shows exceptions raised by `throw/1` or one of the built-in predicates. See section 4.10.

The standard ports are called `call`, `exit`, `redo`, `fail` and `unify`. The tracer is started by the `trace/0` command, when a spy point is reached and the system is in debugging mode (see `spy/1` and `debug/0`) or when an exception is raised.

The interactive top-level goal `trace/0` means "trace the next query". The tracer shows the port, displaying the port name, the current depth of the recursion and the goal. The goal

```
min_numlist([H|T], Min) :-
        min_numlist(T, H, Min).

min_numlist([], Min, Min).
min_numlist([H|T], Min0, Min) :-
        Min1 is min(H, Min0),
        min_numlist(T, Min1, Min).
```

```
1 ?- visible(+all), leash(-exit).
true.

2 ?- trace, min_numlist([3, 2], X).
   Call: (7) min_numlist([3, 2], _G0) ? creep
   Unify: (7) min_numlist([3, 2], _G0)
   Call: (8) min_numlist([2], 3, _G0) ? creep
   Unify: (8) min_numlist([2], 3, _G0)
^  Call: (9) _G54 is min(2, 3) ? creep
^  Exit: (9) 2 is min(2, 3)
   Call: (9) min_numlist([], 2, _G0) ? creep
   Unify: (9) min_numlist([], 2, 2)
   Exit: (9) min_numlist([], 2, 2)
   Exit: (8) min_numlist([2], 3, 2)
   Exit: (7) min_numlist([3, 2], 2)
X = 2.
```

Figure 2.2: Example trace of the program above showing all ports. The lines marked ^ indicate calls to *transparent* predicates. See section 5.

is printed using the Prolog predicate `write_term/2`. The style is defined by the Prolog flag `debugger_print_options` and can be modified using this flag or using the `w`, `p` and `d` commands of the tracer.

On *leashed ports* (set with the predicate `leash/1`, default are `call`, `exit`, `redo` and `fail`) the user is prompted for an action. All actions are single character commands which are executed **without** waiting for a return, unless the command-line option `-tty` is active. Tracer options:

`+` **(Spy)**
Set a spy point (see `spy/1`) on the current predicate.

`-` **(No spy)**
Remove the spy point (see `nospy/1`) from the current predicate.

`/` **(Find)**
Search for a port. After the '/', the user can enter a line to specify the port to search for. This line consists of a set of letters indicating the port type, followed by an optional term, that should unify with the goal run by the port. If no term is specified it is taken as a variable, searching for

any port of the specified type. If an atom is given, any goal whose functor has a name equal to that atom matches. Examples:

`/f`	Search for any fail port
`/fe solve`	Search for a fail or exit port of any goal with name `solve`
`/c solve(a, _)`	Search for a call to solve/2 whose first argument is a variable or the atom `a`
`/a member(_, _)`	Search for any port on `member/2`. This is equivalent to setting a spy point on `member/2`.

`.` **(Repeat find)**
Repeat the last find command (see '/').

`A` **(Alternatives)**
Show all goals that have alternatives.

`C` **(Context)**
Toggle 'Show Context'. If `on`, the context module of the goal is displayed between square brackets (see section 5). Default is `off`.

`L` **(Listing)**
List the current predicate with `listing/1`.

`a` **(Abort)**
Abort Prolog execution (see `abort/0`).

`b` **(Break)**
Enter a Prolog break environment (see `break/0`).

`c` **(Creep)**
Continue execution, stop at next port. (Also return, space).

`d` **(Display)**
Set the `max_depth`(*Depth*) option of `debugger_print_options`, limiting the depth to which terms are printed. See also the `w` and `p` options.

`e` **(Exit)**
Terminate Prolog (see `halt/0`).

`f` **(Fail)**
Force failure of the current goal.

`g` **(Goals)**
Show the list of parent goals (the execution stack). Note that due to tail recursion optimization a number of parent goals might not exist any more.

`h` **(Help)**
Show available options (also '?').

`i` **(Ignore)**
Ignore the current goal, pretending it succeeded.

`l` **(Leap)**
Continue execution, stop at next spy point.

`n` **(No debug)**
Continue execution in ‘no debug’ mode.

`p` **(Print)**
Set the Prolog flag `debugger_print_options` to `[quoted(true), portray(true), max_depth(10), priority(699)]`. This is the default.

`r` **(Retry)**
Undo all actions (except for database and i/o actions) back to the call port of the current goal and resume execution at the call port.

`s` **(Skip)**
Continue execution, stop at the next port of **this** goal (thus skipping all calls to children of this goal).

`u` **(Up)**
Continue execution, stop at the next port of **the parent** goal (thus skipping this goal and all calls to children of this goal). This option is useful to stop tracing a failure driven loop.

`w` **(Write)**
Set the Prolog flag `debugger_print_options` to `[quoted(true), attributes(write), priority(699)]`, bypassing `portray/1`, etc.

The ideal 4 port Byrd box model [Byrd, 1980] as described in many Prolog books [Clocksin & Melish, 1987] is not visible in many Prolog implementations because code optimisation removes part of the choice- and exit-points. Backtrack points are not shown if either the goal succeeded deterministically or its alternatives were removed using the cut. When running in debug mode (`debug/0`) choice points are only destroyed when removed by the cut. In debug mode, last call optimisation is switched off.[2]

Reference information to all predicates available for manipulating the debugger is in section 4.37.

2.10 Compilation

2.10.1 During program development

During program development, programs are normally loaded using `consult/1`, or the list abbreviation. It is common practice to organise a project as a collection of source files and a *load-file*, a Prolog file containing only `use_module/[1,2]` or `ensure_loaded/1` directives, possibly with a definition of the *entry-point* of the program, the predicate that is normally used to start the program. This file is often called `load.pl`. If the entry-point is called *go*, a typical session starts as:

```
% swipl
<banner>
```

---

[2] This implies the system can run out of local stack in debug mode, while no problems arise when running in non-debug mode.

```
1 ?- [load].
<compilation messages>

Yes
2 ?- go.
<program interaction>
```

When using Windows, the user may open `load.pl` from the Windows explorer, which will cause `swipl-win.exe` to be started in the directory holding `load.pl`. Prolog loads `load.pl` before entering the top-level.

2.10.2 For running the result

There are various options if you want to make your program ready for real usage. The best choice depends on whether the program is to be used only on machines holding the SWI-Prolog development system, the size of the program and the operating system (Unix vs. Windows).

Using PrologScript

New in version 4.0.5 is the possibility to use a Prolog source file directly as a Unix script-file. The same mechanism is useful to specify additional parameters for running a Prolog file on Windows.

If the first letter of a Prolog file is `#`, the first line is treated as comment.[3] To create a Prolog script, make the first line start like this:

`#!/path/to/pl` ⟨*options*⟩ `-s`

Prolog recognises this starting sequence and causes the interpreter to receive the following argument-list:

`/path/to/pl` ⟨*options*⟩ `-s` ⟨*script*⟩ `--` ⟨*ScriptArguments*⟩

Instead of `-s`, the user may use `-f` to stop Prolog from looking for a personal initialisation file.

Here is a simple script doing expression evaluation:

```
#!/usr/bin/pl -q -t main -f

eval :-
        current_prolog_flag(argv, Argv),
        append(_, [--|Args], Argv),
        concat_atom(Args, ' ', SingleArg),
        term_to_atom(Term, SingleArg),
        Val is Term,
        format('~w~n', [Val]).

main :-
```

---

[3] The `#`-sign can be the legal start of a normal Prolog clause. In the unlikely case this is required, leave the first line blank or add a header-comment.

```
        catch(eval, E, (print_message(error, E), fail)),
        halt.
main :-
        halt(1).
```

And here are two example runs:

```
% eval 1+2
3
% eval foo
ERROR: Arithmetic: `foo/0' is not a function
%
```

The Windows version supports the `#!` construct too, but here it serves a rather different role. The Windows shell already allows the user to start Prolog source files directly through the Windows file-type association. Windows however makes it rather complicated to provide additional parameters, such as the required stack-size for an individual Prolog file. The `#!` line provides for this, providing a more flexible approach than changing the global defaults. The following starts Prolog with unlimited stack-size on the given source file:

```
#!/usr/bin/pl -L0 -T0 -G0 -s

....
```

Note the use of `/usr/bin/pl`, which specifies the interpreter. This argument is ignored in the Windows version, but required to ensure best cross-platform compatibility.

Creating a shell-script

With the introduction of *PrologScript* (see section 2.10.2), using shell-scripts as explained in this section has become redundant for most applications.

Especially on Unix systems and not-too-large applications, writing a shell-script that simply loads your application and calls the entry-point is often a good choice. A skeleton for the script is given below, followed by the Prolog code to obtain the program arguments.

```
#!/bin/sh

base=<absolute-path-to-source>
PL=pl

exec $PL -f none -g "load_files(['$base/load'],[silent(true)])" \
         -t go -- $*
```

```
go :-
        current_prolog_flag(argv, Arguments),
        append(_SytemArgs, [--|Args], Arguments), !,
        go(Args).

go(Args) :-
        ...
```

On Windows systems, similar behaviour can be achieved by creating a shortcut to Prolog, passing the proper options or writing a `.bat` file.

Creating a saved-state

For larger programs, as well as for programs that are required to run on systems that do not have the SWI-Prolog development system installed, creating a saved state is the best solution. A saved state is created using `qsave_program/[1,2]` or using the linker swipl-ld(1). A saved state is a file containing machine-independent intermediate code in a format dedicated for fast loading. Optionally, the emulator may be integrated in the saved state, creating a single-file, but machine-dependent, executable. This process is described in chapter 10.

Compilation using the -c command-line option

This mechanism loads a series of Prolog source files and then creates a saved-state as `qsave_program/2` does. The command syntax is:

```
% swipl [option ...] [-o output] -c file ...
```

The *options* argument are options to `qsave_program/2` written in the format below. The option-names and their values are described with `qsave_program/2`.

−−*option-name*=*option-value*

For example, to create a stand-alone executable that starts by executing `main/0` and for which the source is loaded through `load.pl`, use the command

```
% swipl --goal=main --stand_alone=true -o myprog -c load.pl
```

This performs exactly the same as executing

```
% swipl
<banner>

?- [load].
?- qsave_program(myprog,
                 [ goal(main),
                   stand_alone(true)
                 ]).
?- halt.
```

2.11 Environment Control (Prolog flags)

The predicates `current_prolog_flag/2` and `set_prolog_flag/2` allow the user to examine and modify the execution environment. It provides access to whether optional features are available on this version, operating system, foreign-code environment, command-line arguments, version, as well as runtime flags to control the runtime behaviour of certain predicates to achieve compatibility with other Prolog environments.

current_prolog_flag(*?Key, -Value***)** *[ISO]*

The predicate `current_prolog_flag/2` defines an interface to installation features: options compiled in, version, home, etc. With both arguments unbound, it will generate all defined Prolog flags. With 'Key' instantiated, it unifies the value of the Prolog flag. Flag values are typed. Flags marked as `bool` can have the values `true` and `false`. Some Prolog flags are not defined in all versions, which is normally indicated in the documentation below as *"if present and true"*. A Boolean Prolog flag is true iff the Prolog flag is present **and** the *Value* is the atom `true`. Tests for such flags should be written as below.

```
        (   current_prolog_flag(windows, true)
        ->  <Do MS-Windows things>
        ;   <Do normal things>
        )
```

access_level *(atom, changeable)*

This flag defines a normal 'user' view (`user`, default) or a 'system' view. In system view all system code is fully accessible as if it was normal user code. In user view, certain operations are not permitted and some details are kept invisible. We leave the exact consequences undefined, but for example system code can be traced using system access and system predicates can be redefined.

address_bits *(integer)*

Address-size of the hosting machine. Typically 32 or 64. Except for the maximum stack limit, this has few implications to the user. See also the Prolog flag `arch`.

agc_margin *(integer, changeable)*

If this amount of atoms has been created since the last atom garbage collection, perform atom garbage collection at the first opportunity. Initial value is 10,000. May be changed. A value of 0 (zero) disables atom garbage collection. See also `PL_register_atom()`.

allow_variable

argv *(list)*
List is a list of atoms representing the command-line arguments used to invoke SWI-Prolog. Please note that **all** arguments are included in the list returned.

arch *(atom)*
Identifier for the hardware and operating system SWI-Prolog is running on. Used to select foreign files for the right architecture. See also section 9.2.3 and `file_search_path/2`.

associate *(atom, changeable)*
On Windows systems, this is set to the filename extension (`pl` (default) or `pro` (can be selected in the installer)) associated with `swipl-win.exe`.

autoload *(bool, changeable)*
If `true` (default) autoloading of library functions is enabled.

backquoted_string *(bool, changeable)*
If `true` (default `false`), read translates text between backquotes into a string object (see section 4.23). This flag is mainly for compatibility with LPA Prolog.

bounded *(bool)*
ISO Prolog flag. If `true`, integer representation is bound by `min_integer` and `max_integer`. If `false` integers can be arbitrarily large and the `min_integer` and `max_integer` are not present. See section 4.26.2.

break_level *(integer)*
Current break-level. The initial toplevel (started with `-t`) has value 0. See `break/0`. This flag is absent from threads that are not running a toplevel loop.

c_cc *(atom)*
Name of the C-compiler used to compile SWI-Prolog. Normally either gcc or cc. See section 9.5.

c_ldflags *(atom)*
Special linker flags passed to link SWI-Prolog. See section 9.5.

c_libs *(atom)*
Libraries passed to the C-linker when SWI-Prolog was linked. May be used to determine the libraries needed to create statically linked extensions for SWI-Prolog. See section 9.5.

char_conversion *(bool, changeable)*
Determines whether character-conversion takes place while reading terms. See also `char_conversion/2`.

character_escapes *(bool, changeable)*
If `true` (default), `read/1` interprets \ escape sequences in quoted atoms and strings. May be changed. This flag is local to the module in which it is changed.

compile_meta_arguments *(atom, changeable)*
Experimental flag that controls compilation of arguments passed to meta-calls marked '0' or '^' (see `meta_predicate/1`). Supported values are:

false
(default). Meta-arguments are passed verbatim.

control
Compile meta-arguments that contain control-structures ((A,B), (A;B), (A-¿B;C),

etc.). If not compiled at compile-time, such arguments are compiled to a temporary clause before execution. Using this option enhances performance of processing complex meta-goals that are known at compile-time.

true

Also compile references to normal user-predicates. This harms performance (a little), but enhances the power of poor-mens consistency check used by `make/0` and implemented by `list_undefined/0`.

always

Always create an intermediate clause, even for system predicates. This prepares for a replacing the normal head of the generated predicate with a special reference (similar to database-references as used by e.g., `assert/2`) that provides direct access to the executable code, thus avoiding runtime lookup of predicates for meta-calling.

compiled_at *(atom)*

Describes when the system has been compiled. Only available if the C-compiler used to compile SWI-Prolog provides the __DATE__ and __TIME__ macros.

console_menu *(bool)*

Set to `true` in `swipl-win.exe` to indicate the console supports menus. See also section 4.33.2.

cpu_count *(integer, changeable)*

Number of physical CPUs in the system. Unfortunately there is no standard to get this number, so on most operating systems this flag is not available. It is marked read-write both to allow obtaining this value later and to allow pretending the system has more or less processors. See also `thread_setconcurrency/2` and the library `thread`. Currently this flag is supported in Windows and Linux if `/proc` is enabled. If you can provide us with a C-code fragment getting the number for a specific OS, please submit an enhancement report at `http://gollem.science.uva.nl/bugzilla/`

dde *(bool)*

Set to `true` if this instance of Prolog supports DDE as described in section 4.41.

debug *(bool, changeable)*

Switch debugging mode on/off. If debug mode is activated the system traps encountered spy-points (see `spy/1`) and trace-points (see `trace/1`). In addition, last-call optimisation is disabled and the system is more conservative in destroying choice points to simplify debugging.

Disabling these optimisations can cause the system to run out of memory on programs that behave correctly if debug mode is off.

debug_on_error *(bool, changeable)*

If `true`, start the tracer after an error is detected. Otherwise just continue execution. The goal that raised the error will normally fail. See also `fileerrors/2` and the Prolog flag `report_error`. May be changed. Default is `true`, except for the runtime version.

debugger_print_options *(term, changeable)*

This argument is given as option-list to `write_term/2` for printing goals by the debugger. Modified by the ‘w’, ‘p’ and ‘⟨*N*⟩ d’ commands of the debugger. Default is `[quoted(true), portray(true), max_depth(10), attributes(portray)]`.

debugger_show_context *(bool, changeable)*
If `true`, show the context module while printing a stack-frame in the tracer. Normally controlled using the 'C' option of the tracer.

dialect *(atom)*
Fixed to 600(:-cer)55.Tf .Tf .Tfrg 5.Tf .Tf .TfRG10(w)]TJETq521.9 591.2 Tf670.386 cm[]0 199 J 0.398 0 -9.

gui *(bool)*
Set to `true` if XPCE is around and can be used for graphics.

history *(integer, changeable)*
If *integer* > 0, support Unix `csh(1)` like history as described in section 2.7. Otherwise, only support reusing commands through the command-line editor. The default is to set this Prolog flag to 0 if a command-line editor is provided (see Prolog flag `readline`) and 15 otherwise.

home *(atom)*
SWI-Prolog's notion of the home-directory. SWI-Prolog uses its home directory to find its startup file as ⟨*home*⟩`/boot32.prc` (32-bit machines) or ⟨*home*⟩`/boot64.prc` (64-bit machines) and to find its library as ⟨*home*⟩`/library`.

hwnd *(integer)*
In `swipl-win.exe`, this refers to the MS-Windows window-handle of the console window.

integer_rounding_function *(down,toward_zero)*
ISO Prolog flag describing rounding by `//` and `rem` arithmetic functions. Value depends on the C-compiler used.

iso *(bool, changeable)*
Include some weird ISO compatibility that is incompatible with normal SWI-Prolog behaviour. Currently it has the following effect:

- The `//2` (float division) *always* return a float, even if applied to integers that can be divided.
- In the standard order of terms (see section 4.7.1), all floats are before all integers.
- `atom_length/2` yields a type error if the first argument is a number.
- `clause/[2,3]` raises a permission error when accessing static predicates.
- `abolish/[1,2]` raises a permission error when accessing static predicates.
- Syntax is closer to the ISO standard.
  - Unquoted commas and bars appearing as atoms are not allowed. Instead of `f(,,a)` now write `f(',',a)`. Unquoted commas can only be used to separate arguments in functional notation and list notation, and as a conjunction operator. Unquoted bars can only appear within lists to separate head and tail like `[Head|Tail]`, and as infix operator for alternation in grammar rules like `a --> b | c.`
  - Within functional notation and list notation terms must have priority below 1000. That means that rules and control constructs appearing as arguments need bracketing. A term like `[a :- b, c].` must now be disambiguated to mean `[(a :- b), c].` or `[(a :- b, c)].`
  - Operators appearing as operands must be bracketed. Instead of `X == -, true.` write `X == (-), true.` Currently, this is not entirely enforced.

large_files *(bool)*
If present and `true`, SWI-Prolog has been compiled with *large file support* (LFS) and is capable to access files larger than 2GB on 32-bit hardware. Large file-support is default on installations built using `configure` that support it and may be switched off using the configure option `--disable-largefile`.

max_arity *(unbounded)*
ISO Prolog flag describing there is no maximum arity to compound terms.

max_integer *(integer)*
Maximum integer value if integers are *bounded*. See also the flag `bounded` and section 4.26.2.

max_tagged_integer *(integer)*
Maximum integer value represented as a 'tagged' value. Tagged integers require 1 word storage. Larger integers are represented as 'indirect data' and require significantly more space.

min_integer *(integer)*
Minimum integer value if integers are *bounded*. See also the flag `bounded` and section 4.26.2.

min_tagged_integer *(integer)*
Start of the tagged-integer value range.

occurs_check *(atom, changeable)*
This flag controls unification that creates an infinite tree (also called *cyclic term*) and can have three values. Using `false` (default), unification succeeds, creating an infinite tree. Using `true`, unification behaves as `unify_with_occurs_check/2`, failing silently. Using `error`, an attempt to create a cyclic term results in an `occurs_check` exception. The latter is intended for debugging unintentional creations of cyclic terms. Note that this flag is a global flag modifying fundamental behaviour of Prolog. Changing the flag from its default may cause libraries to stop functioning properly.

open_shared_object *(bool)*
If true, `open_shared_object/2` and friends are implemented, providing access to shared libraries (`.so` files) or dynamic link libraries (`.DLL` files).

optimise *(bool, changeable)*
If `true`, compile in optimised mode. The initial value is `true` if Prolog was started with the `-O` command-line option.

Currently optimise compilation implies compilation of arithmetic, and deletion of redundant `true/0` that may result from `expand_goal/2`.

Later versions might imply various other optimisations such as integrating small predicates into their callers, eliminating constant expressions and other predictable constructs. Source code optimisation is never applied to predicates that are declared dynamic (see `dynamic/1`).

pid *(int)*
Process identifier of the running Prolog process. Existence of this flag is implementation defined.

pipe *(bool, changeable)*
If true, `open(pipe(command), mode, Stream)`, etc. are supported. Can be changed to disable the use of pipes in applications testing this feature. Not recommended.

preprocessor *(atom, changeable)*
Current preprocessor. See `preprocessor/2`. The reserved atom =none= (default) implies no preprocessor.

prompt_alternatives_on *(atom, changeable)*
Determines prompting for alternatives in the Prolog toplevel. Default is `determinism`, which implies the system prompts for alternatives if the goal succeeded while leaving choicepoints. Many classical Prolog systems behave as `groundness`: they prompt for alternatives if and only if the query contains variables.

qcompile *(atom, changeable)*
This option provides the default for the `qcompile`(+*Atom*) option of `load_files/2`.

readline *(bool)*
If true, SWI-Prolog is linked with the readline library. This is done by default if you have this library installed on your system. It is also true for the Win32 swipl-win.exe version of SWI-Prolog, which realises a subset of the readline functionality.

resource_database *(atom)*
Set to the absolute-filename of the attached state. Typically this is the file `boot32.prc`, the file specified with `-x` or the running executable. See also `resource/3`.

report_error *(bool, changeable)*
If `true`, print error messages, otherwise suppress them. May be changed. See also the `debug_on_error` Prolog flag. Default is `true`, except for the runtime version.

runtime *(bool)*
If present and `true`, SWI-Prolog is compiled with -DO_RUNTIME, disabling various useful development features (currently the tracer and profiler).

saved_program *(bool)*
If present and `true`, Prolog has been started from a state saved with `qsave_program/[1,2]`.

shared_object_extension *(atom)*
Extension used by the operating system for shared objects. `.so` for most Unix systems and `.dll` for Windows. Used for locating files using the `file_type` `executable`. See also `absolute_file_name/3`.

shared_object_search_path *(atom)*
Name of the environment variable used by the system to search for shared objects.

signals *(bool)*
Determine whether Prolog is handling signals (software interrupts). This flag is `false` if the hosting OS does not support signal handling or the command-line option `-nosignals` is active. See section 9.4.21 for details.

stream_type_check *(atom, changeable)*
Defines whether and how strictly the system validates that byte IO should not be applied to text-streams and text IO should not be applied to binary streams. Values are `false` (no checking), `true` (full checking) and `loose`. Using checking mode `loose` (default), the system accepts byte I/O from text stream that use ISO Latin-1 encoding and accepts writing text to binary streams.

system_thread_id *(int)*
Available in multi-threaded version (see section 8) where the operating system provides system-wide integer thread identifiers. The integer is the thread-identifier used by the operating system for the calling thread. See also `thread_self/1`.

last_call_optimisation *(bool, changeable)*
Determines whether or not last-call optimisation is enabled. Normally the value of this flag is equal to the `debug` flag. As programs may run out of stack if last-call optimisation is omitted, it is sometimes necessary to enable it during debugging.

timezone *(integer)*
Offset in seconds west of GMT of the current time-zone. Set at initialization time from the `timezone` variable associated with the POSIX tzset() function. See also `convert_time/2`.

toplevel_print_anon *(bool, changeable)*
If `true`, top-level variables starting with an underscore (_) are printed normally. If `false` they are hidden. This may be used to hide bindings in complex queries from the top-level.

toplevel_print_factorized *(bool, changeable)*
If `true` (default `false`) show the internal sharing of subterms in the answer substitution. The example below reveals internal sharing of leaf-nodes in *red-black trees* as implemented by the `rbtrees` predicate `rb_new/1`:

```
?- set_prolog_flag(toplevel_print_factorized, true).
?- rb_new(X).
X = t(_S1, _S1), % where
    _S1 = black(’’, _G387, _G388, ’’).
```

If this flag is `false`, the `% where` notation is still used to indicate cycles as illustrated below. This example also shows that the implementation reveals the internal cycle length, and *not* the minimal cycle length. Cycles of different length are indistinguishable in Prolog (as illustrated by `S == R`).

```
?- S = s(S), R = s(s(R)), S == R.
S = s(S),
R = s(s(R)).
```

toplevel_print_options *(term, changeable)*
This argument is given as option-list to `write_term/2` for printing results of queries. Default is `[quoted(true), portray(true), max_depth(10), attributes(portray)]`.

toplevel_prompt *(atom, changeable)*
Define the prompt that is used by the interactive toplevel. The following ~ (tilde) sequences are replaced:

~m	*Type in* module if not `user` (see `module/1`)
~l	*Break level* if not 0 (see `break/0`)
~d	*Debugging state* if not normal execution (see `debug/0`, `trace/0`)
~!	*History event* if history is enabled (see flag `history`)

toplevel_var_size *(int, changeable)*
Maximum size counted in literals of a term returned as a binding for a variable in a top-level query that is saved for re-use using the `$` variable reference. See section 2.8.

trace_gc *(bool, changeable)*
If true (false is the default), garbage collections and stack-shifts will be reported on the

terminal. May be changed. Values are reported in bytes as $G+T$, where G is the global stack value and T the trail stack value. ‘Gained’ describes the number of bytes reclaimed. ‘used’ the number of bytes on the stack after GC and ‘free’ the number of bytes allocated, but not in use. Below is an example output.

```
% GC: gained 236,416+163,424 in 0.00 sec; used 13,448+5,808; free 72,568+
```

tty_control *(bool, changeable)*
Determines whether the terminal is switched to raw mode for `get_single_char/1`, which also reads the user-actions for the trace. May be set. See also the `+/-tty` command-line option.

unix *(bool)*
If present and `true`, the operating system is some version of Unix. Defined if the C-compiler used to compile this version of SWI-Prolog either defines `__unix__` or `unix`. On other systems this flag is not available.

unknown *(fail,warning,error, changeable)*
Determines the behaviour if an undefined procedure is encountered. If `fail`, the predicate fails silently. If `warn`, a warning is printed, and execution continues as if the predicate was not defined and if `error` (default), an `existence_error` exception is raised. This flag is local to each module and inherited from the module’s *import-module*. Using default setup, this implies that normal modules inherit the flag from `user`, which in turn inherits the value `error` from `system`. The user may change the flag for module `user` to change the default for all application modules or for a specific module. It is strongly advised to keep the `error` default and use `dynamic/1` and/or `multifile/1` to specify possible non-existence of a predicate.

user_flags *(Atom, changeable)*
Define the behaviour of `set_prolog_flag/2` if the flag is not known. Values are `silent`, `warning` and `error`. The first two create the flag on-the-fly, where `warning` prints a message. The value `error` is consistent with ISO: it raises an existence error and does not create the flag. See also `create_prolog_flag/3`. The default is `silent`, but future versions may change that. Developers are encouraged to use another value and ensure proper use of `create_prolog_flag/3` to create flags for their library.

verbose *(Atom, changeable)*
This flag is used by `print_message/2`. If its value is `silent`, messages of type `informational` and `banner` are suppressed. The `-q` switches the value from the initial `normal` to `silent`.

verbose_autoload *(bool, changeable)*
If `true` the normal consult message will be printed if a library is autoloaded. By default this message is suppressed. Intended to be used for debugging purposes.

verbose_load *(bool, changeable)*
If `false` normal consult messages will be suppressed. Default is `true`. The value of this flag is normally controlled by the option `silent`(*Bool*) provided by `load_files/2`.

verbose_file_search *(bool, changeable)*
If `true` (default `false`), print messages indicating the progress of

`absolute_file_name/[2,3]` in locating files. Intended for debugging complicated file-search paths. See also `file_search_path/2`.

version *(integer)*
The version identifier is an integer with value:

$$10000 \times \mathit{Major} + 100 \times \mathit{Minor} + \mathit{Patch}$$

Note that in releases up to 2.7.10 this Prolog flag yielded an atom holding the three numbers separated by dots. The current representation is much easier for implementing version-conditional statements.

version_data *(swi(Major, Minor, Patch, Extra))*
Part of the dialect compatibility layer; see also the Prolog flag `dialect` and section C. *Extra* provides platform-specific version information. Currently it is simply unified to `[]`.

version_git *(atom)*
Available if created from a git repository. See `git-describe` for details.

windows *(bool)*
If present and `true`, the operating system is an implementation of Microsoft Windows (NT/2000/XP, etc.). This flag is only available on MS-Windows based versions.

write_attributes *(atom, changeable)*
Defines how `write/1` and friends write attributed variables. The option values are described with the `attributes` option of `write_term/3`. Default is `ignore`.

write_help_with_overstrike *(bool)*
Internal flag used by `help/1` when writing to a terminal. If present and `true` it prints bold and underlined text using *overstrike*.

xpce *(bool)*
Available and set to `true` if the XPCE graphics system is loaded.

xpce_version *(atom)*
Available and set to the version of the loaded XPCE system.

set_prolog_flag(*:Key, +Value***)** *[ISO]*
Define a new Prolog flag or change its value. *Key* is an atom. If the flag is a system-defined flag that is not marked *changeable* above, an attempt to modify the flag yields a `permission_error`. If the provided *Value* does not match the type of the flag, a `type_error` is raised.

Some flags (e.g., `unknown`) are maintained on a per-module basis. The addressed module is determined by the *Key* argument.

In addition to ISO, SWI-Prolog allows for user-defined Prolog flags. The type of the flag is determined from the initial value and cannot be changed afterwards. Defined types are `boolean` (if the initial value is one of `false`, `true`, `on` or `off`), `atom` if the initial value is any other atom, `integer` if the value is an integer that can be expressed as a 64-bit signed value. Any other initial value results in an untyped flag that can represent any valid Prolog term.

Originally, SWI-Prolog's `set_prolog_flag/2` created a new Prolog flag if the flag *Key* did not exist. It still does this, but now prints a warning. New code must use `create_prolog_flag/3` to introduce new flags. Future versions are likely to replace this printed warning with an existence error.

create_prolog_flag(+*Key*, +*Value*, +*Options***)** *[YAP]*

Create a new Prolog flag. The ISO standard does not foresee creation of new flags, but many libraries introduce new flags. *Options* is a list of the following options:

access(+*Access***)**
Define access-rights for the flag. Values are `read_write` and `read_only`. The default is `read_write`.

type(+*Atom***)**
Define a type-restriction. Possible values are `boolean`, `atom`, `integer`, `float` and `term`. The default is determined from the initial value. Note that `term` restricts the term to be ground.

This predicate behaves as `set_prolog_flag/2` if the flag already exists. See also `user_flags`.

2.12 An overview of hook predicates

SWI-Prolog provides a large number of hooks, mainly to control handling messages, debugging, startup, shut-down, macro-expansion, etc. Below is a summary of all defined hooks with an indication of their portability.

- *portray/1*
  Hook into `write_term/3` to alter the way terms are printed (ISO).

- *message_hook/3*
  Hook into `print_message/2` to alter the way system messages are printed (Quintus/SICStus).

- *library_directory/1*
  Hook into `absolute_file_name/3` to define new library directories (most Prolog systems).

- *file_search_path/2*
  Hook into `absolute_file_name/3` to define new search-paths (Quintus/SICStus).

- *term_expansion/2*
  Hook into `load_files/2` to modify read terms before they are compiled (macro-processing) (most Prolog systems).

- *goal_expansion/2*
  Same as `term_expansion/2` for individual goals (SICStus).

- *prolog_load_file/2*
  Hook into `load_files/2` to load other data-formats for Prolog sources from ‘non-file’ resources. The `load_files/2` predicate is the ancestor of `consult/1`, `use_module/1`, etc.

- *prolog_edit:locate/3*
  Hook into `edit/1` to locate objects (SWI).

- *prolog_edit:edit_source/1*
  Hook into `edit/1` to call an internal editor (SWI).

- *prolog_edit:edit_command/2*
  Hook into `edit/1` to define the external editor to use (SWI).

- *prolog_list_goal/1*
  Hook into the tracer to list the code associated to a particular goal (SWI).

- *prolog_trace_interception/4*
  Hook into the tracer to handle trace-events (SWI).

- *prolog:debug_control_hook/1*
  Hook in `spy/1`, `nospy/1`, `nospyall/0` and `debugging/0` to extend these control-predicates to higher-level libraries.

- *prolog:help_hook/1*
  Hook in `help/0`, `help/1` and `apropos/1` to extend the help-system.

- *resource/3*
  Defines a new resource (not really a hook, but similar) (SWI).

- *exception/3*
  Old attempt to a generic hook mechanism. Handles undefined predicates (SWI).

- *attr_unify_hook/2*
  Unification hook for attributed variables. Can be defined in any module. See section 6.1 for details.

2.13 Automatic loading of libraries

If —at runtime— an undefined predicate is trapped, the system will first try to import the predicate from the module's default module (see section 5.9. If this fails the *auto loader* is activated.[5] On first activation an index to all library files in all library directories is loaded in core (see `library_directory/1`, `file_search_path/2` and `reload_library_index/0`). If the undefined predicate can be located in one of the libraries, that library file is automatically loaded and the call to the (previously undefined) predicate is restarted. By default this mechanism loads the file silently. The `current_prolog_flag/2` key `verbose_autoload` is provided to get verbose loading. The Prolog flag `autoload` can be used to enable/disable the autoload system.

Autoloading only handles (library) source files that use the module mechanism described in chapter 5. The files are loaded with `use_module/2` and only the trapped undefined predicate is imported into the module where the undefined predicate was called. Each library directory must hold a file `INDEX.pl` that contains an index to all library files in the directory. This file consists of lines of the following format:

```
index(Name, Arity, Module, File).
```

---

[5] Actually, the hook user:exception/3 is called; only if this hook fails it calles the autoloader.

The predicate `make/0` updates the autoload index. It searches for all library directories (see `library_directory/1` and `file_search_path/2`) holding the file `MKINDEX.pl` or `INDEX.pl`. If the current user can write or create the file `INDEX.pl` and it does not exist or is older than the directory or one of its files, the index for this directory is updated. If the file `MKINDEX.pl` exists updating is achieved by loading this file, normally containing a directive calling `make_library_index/2`. Otherwise `make_library_index/1` is called, creating an index for all `*.pl` files containing a module.

Below is an example creating an indexed library directory.

```
% mkdir ~/lib/prolog
% cd ~/lib/prolog
% swipl -g true -t 'make_library_index(.)'
```

If there is more than one library file containing the desired predicate, the following search schema is followed:

1. If there is a library file that defines the module in which the undefined predicate is trapped, this file is used.
2. Otherwise library files are considered in the order they appear in the `library_directory/1` predicate and within the directory alphabetically.

autoload_path(+*DirAlias***)**

Add DirAlias to the libraries that are used by the autoloader. This extends the search path `autoload` and reloads the library index. For example:

```
:- autoload_path(library(http)).
```

If this call appears as a directive, it is term-expanded into a clause for user:file_search_path/2 and a directive calling `reload_library_index/0`. This keeps source information and allows for removing this directive.

make_library_index(+*Directory***)**

Create an index for this directory. The index is written to the file 'INDEX.pl' in the specified directory. Fails with a warning if the directory does not exist or is write protected.

make_library_index(+*Directory*, +*ListOfPatterns***)**

Normally used in `MKINDEX.pl`, this predicate creates `INDEX.pl` for *Directory*, indexing all files that match one of the file-patterns in *ListOfPatterns*.

Sometimes library packages consist of one public load file and a number of files used by this load-file, exporting predicates that should not be used directly by the end-user. Such a library can be placed in a sub-directory of the library and the files containing public functionality can be added to the index of the library. As an example we give the XPCE library's `MKINDEX.pl`, including the public functionality of `trace/browse.pl` to the autoloadable predicates for the XPCE package.

```
:- make_library_index('.',
                      [ '*.pl',
                        'trace/browse.pl'
                      ]).
```

reload_library_index

Force reloading the index after modifying the set of library directories by changing the rules for `library_directory/1`, `file_search_path/2`, adding or deleting `INDEX.pl` files. This predicate does *not* update the `INDEX.pl` files. Check `make_library_index/[1,2]` and `make/0` for updating the index files.

Normally, the index is reloaded automatically if a predicate cannot be found in the index and the set of library directories has changed. Using `reload_library_index/0` is necessary if directories are removed or the order of the library directories is changed.

When creating an executable using either `qsave_program/2` or the `-c` command line options, it is necessarry to load all predicates that would normally be autoloaded explicitly. This is discussed in section 10. See `autoload/0`.

2.14 Garbage Collection

SWI-Prolog provides garbage-collection, last-call optimization and atom garbage collection. These features are controlled using Prolog flags (see `current_prolog_flag/2`).

2.15 Syntax Notes

SWI-Prolog syntax is close to ISO-Prolog standard syntax, which is closely compatible with Edinburgh Prolog syntax. A description of this syntax can be found in the Prolog books referenced in the introduction. Below are some non-standard or non-common constructs that are accepted by SWI-Prolog:

- `/* .../* ...*/ ...*/`
  The `/* ...*/` comment statement can be nested. This is useful if some code with `/* ...*/` comment statements in it should be commented out.

2.15.1 ISO Syntax Support

SWI-Prolog offers ISO compatible extensions to the Edinburgh syntax.

Processor Character Set

The processor character set specifies the class of each character used for parsing Prolog source text. Character classification is fixed to use UCS/Unicode as provided by the C-library `wchar_t` based primitives. See also section 2.18.

Character Escape Syntax

Within quoted atoms (using single quotes: `'<atom>'`) special characters are represented using escape sequences. An escape sequence is led in by the backslash (`\`) character. The list of escape sequences is compatible with the ISO standard but contains some extensions, and the interpretation of numerically specified characters is slightly more flexible to improve compatibility.

`\a`

Alert character. Normally the ASCII character 7 (beep).

`\b`

Backspace character.

`\c`

No output. All input characters up to but not including the first non-layout character are skipped. This allows for the specification of pretty-looking long lines. For compatibility with Quintus Prolog. Not supported by ISO. Example:

```
format('This is a long line that looks better if it was \c
       split across multiple physical lines in the input')
```

\⟨RETURN⟩

No output. Skips input to the next non-layout character or to the end of the next line. ISO demands skipping only the newline. We advise using `\c` or putting the layout *before* the `\`, as shown below. Using `\c` is supported by various other Prolog implementations and will remain supported by SWI-Prolog. The style shown below is the most compatible solution.[6]

```
format('This is a long line that looks better if it was \
split across multiple physical lines in the input')
```

instead of

```
format('This is a long line that looks better if it was\
 split across multiple physical lines in the input')
```

`\e`

Escape character (ASCII 27). Not ISO, but widely supported.

`\f`

Form-feed character.

`\n`

Next-line character.

`\r`

Carriage-return only (i.e., go back to the start of the line).

---

[6] Future versions are likely to interpret \⟨*return*⟩ according to ISO.

`\s`
: Space character. Intended to allow writing `0'\s` to get the character code of the space character. Not ISO.

`\t`
: Horizontal tab-character.

`\v`
: Vertical tab-character (ASCII 11).

`\xXX..\`
: Hexadecimal specification of a character. The closing \ is obligatory according to the ISO standard, but optional in SWI-Prolog to enhance compatibility with the older Edinburgh standard. The code `\xa\3` emits the character 10 (hexadecimal 'a') followed by '3'. Characters specified this way are interpreted as Unicode characters. See also `\u`.

`\uXXXX`
: Unicode character specification where the character is specified using *exactly* 4 hexadecimal digits. This is an extension to the ISO standard fixing two problems. First of all, where `\x` defines a numeric character code, it doesn't specify the character set in which the character should be interpreted. Second, it is not needed to use the idiosyncratic closing \ ISO Prolog syntax.

`\UXXXXXXXX`
: Same as `\uXXXX`, but using 8 digits to cover the whole Unicode set.

`\40`
: Octal character specification. The rules and remarks for hexadecimal specifications apply to octal specifications as well.

\\⟨*character*⟩
: Any character immediately preceded by a \ and not covered by the above escape sequences is copied verbatim. Thus, `'\\'` is an atom consisting of a single \ and `'\''` and `''''` both describe the atom with a single `'`.

Character escaping is only available if `current_prolog_flag(character_escapes, true)` is active (default). See `current_prolog_flag/2`. Character escapes conflict with `writef/2` in two ways: `\40` is interpreted as decimal 40 by `writef/2`, but as octal 40 (decimal 32) by `read`. Also, `\l` is translated to a single 'l'. It is advised to use the more widely supported `format/[2,3]` predicate instead. If you insist upon using `writef/2`, either switch `character_escapes` to `false`, or use double \\\\, as in `writef('\\l')`.

Syntax for non-decimal numbers

SWI-Prolog implements both Edinburgh and ISO representations for non-decimal numbers. According to Edinburgh syntax, such numbers are written as ⟨*radix*⟩`'<number>`, where ⟨*radix*⟩ is a number between 2 and 36. ISO defines binary, octal and hexadecimal numbers using `0`*[bxo]*⟨*number*⟩. For example: `A is 0b100 \/ 0xf00` is a valid expression. Such numbers are always unsigned.

Unicode Prolog source

The ISO standard specifies the Prolog syntax in ASCII characters. As SWI-Prolog supports Unicode in source files we must extend the syntax. This section describes the implication for the source files, while writing international source files is described in section 3.1.3.

The SWI-Prolog Unicode character classification is based on version 6.0.0 of the Unicode standard. Please note that `char_type/2` and friends, intended to be used with all text except Prolog source code, is based on the C-library locale-based classification routines.

- *Quoted atoms and strings*
  Any character of any script can be used in quoted atoms and strings. The escape sequences `\uXXXX` and `\UXXXXXXXX` (see section 2.15.1) were introduced to specify Unicode code points in ASCII files.

- *Atoms and Variables*
  We handle them in one item as they are closely related. The Unicode standard defines a syntax for identifiers in computer languages.[7] In this syntax identifiers start with `ID_Start` followed by a sequence of `ID_Continue` codes. Such sequences are handled as a single token in SWI-Prolog. The token is a *variable* iff it starts with an uppercase character or an underscore (_). Otherwise it is an atom. Note that many languages do not have the notion of character-case. In such languages variables *must* be written as `_name`.

- *White space*
  All characters marked as separators (Z*) in the Unicode tables are handled as layout characters.

- *Control and unassigned characters*
  Control and unassigned (C*) characters produce a syntax error if encountered outside quoted atoms/strings and outside comments.

- *Other characters*
  The first 128 characters follow the ISO Prolog standard. Unicode symbol and punctuation characters (general category S* and P*) act as glueing symbol characters (i.e., just like ==: an unquoted sequence of symbol characters are combined into an atom).

  Other characters (this is mainly `No`: *a numeric character of other type*) are currently handled as ‘solo’.

Singleton variable checking

A *singleton variable* is a variable that appears only one time in a clause. It can always be replaced by _, the *anonymous* variable. In some cases however people prefer to give the variable a name. As mistyping a variable is a common mistake, Prolog systems generally give a warning (controlled by `style_check/1`) if a variable is used only once. The system can be informed a variable is known to appear once by *starting* it with an underscore. E.g. `_Name`. Please note that any variable, except plain _, shares with variables of the same name. The term `t(_X, _X)` is equivalent to `t(X, X)`, which is *different* from `t(_, _)`.

As Unicode requires variables to start with an underscore in many languages this schema needs to be extended.[8] First we define the two classes of named variables.

---

[7] `http://www.unicode.org/reports/tr31/`

[8] After a proposal by Richard O’Keefe.

- *Named singleton variables*
  Named singletons start with a double underscore (__) or a single underscore followed by an uppercase letter. E.g. `__var` or `_Var`.

- *Normal variables*
  All other variables are 'normal' variables. Note this makes `_var` a normal variable.[9]

Any normal variable appearing exactly once in the clause *and* any named singleton variables appearing more than once are reported. Below are some examples with warnings in the right column. Singleton messages can be suppressed using the `style_check/1` directive.

test(_).	
test(_a).	Singleton variables: [_a]
test(_12).	Singleton variables: [_12]
test(A).	Singleton variables: [A]
test(_A).	
test(__a).	
test(_, _).	
test(_a, _a).	
test(__a, __a).	Singleton-marked variables appearing more than once: [__a]
test(_A, _A).	Singleton-marked variables appearing more than once: [_A]
test(A, A).	

2.16 Rational trees (cyclic terms)

SWI-Prolog supports rational trees, also known as cyclic terms. 'Supports' is defined that most relevant built-in predicates terminate when faced with rational trees. Almost all SWI-Prolog's built-in term manipulation predicates process terms in a time that is linear to the amount of memory used to represent the term on the stack. The following set of predicates safely handles rational trees: `=../2`, `==/2`, `=@=/2`, `=/2`, `@</2`, `@=</2`, `@>=/2`, `@>/2`, `\==/2`, `\=@=/2`, `\=/2`, `acyclic_term/1`, `bagof/3`, `compare/3`, `copy_term/2`, `cyclic_term/1`, `dif/2`, `duplicate_term/2`, `findall/3`, `ground/1`, `term_hash/2`, `numbervars/[3,4]`, `recorda/3`, `recordz/3`, `setof/3`, `term_variables/2`, `throw/1`, `unify_with_occurs_check/2`, `unifiable/3`, `when/2`, `write/1` (and related predicates) .

In addition, some built-ins recognise rational trees and raise an appropriate exception. Arithmetic evaluation belongs to this group. The compiler (`asserta/1`, etc.) also raises an exception. Future versions may support rational trees. Predicates that could provide meaningful processing of rational trees raise a `representation_error`. Predicates for which rational trees have no meaningful interpretation raise a `type_error`. For example:

```
1 ?- A = f(A), asserta(a(A)).
ERROR: asserta/1: Cannot represent due to `cyclic_term'
2 ?- A = 1+A, B is A.
ERROR: is/2: Type error: `expression' expected, found
             `@(S_1,[S_1=1+S_1])' (cyclic term)
```

---

[9] Some Prolog dialects write variables this way.

2.17 Just-in-time clause indexing

SWI-Prolog provides ‘just-in-time’ indexing over multiple arguments.[10] ‘Just-in-time’ means that clause indexes are not built by the compiler (or `asserta/1` for dynamic predicates), but on the first call to such a predicate where an index might help (i.e., a call where at least one argument is instantiated). This section describes the rules used by the indexing logic. Note that this logic is not ‘set in stone’. The indexing capabilii9ieThecale dsThe ot

2.17.1 Future directions

- The current indexing system is largely prepared for secondary indexes. This implies that if there are many clauses that match a given key, the system could (JIT) create a secondary index. This secondary index could exploit another argument or, if the key denotes a functor, an argument inside the compound term.
- The 'special cases' can be extended. This is notably attractive for static predicates with a relatively small number of clauses where a hash lookup is too costly.

2.17.2 Indexing and portability

The base-line functionality of Prolog implementations provide indexing on constants and functor (name/arity) on the first argument. This must be your assumption if wide portability of your program is important. This can typically be achieved by exploiting `term_hash/2` or `term_hash/4` and/or maintaining multiple copies of a predicate with reordered arguments and wrappers that update all implementations (assert/retract) and selects the appropriate implementation (query).

YAP provides full JIT indexing, including indexing arguments of compound terms. YAP's indexing has been the inspiration for enhancing SWI-Prolog's indexing capabilities.

2.18 Wide character support

SWI-Prolog supports *wide characters*, characters with character codes above 255 that cannot be represented in a single *byte*. *Universal Character Set* (UCS) is the ISO/IEC 10646 standard that specifies a unique 31-bit unsigned integer for any character in any language. It is a superset of 16-bit Unicode, which in turn is a superset of ISO 8859-1 (ISO Latin-1), a superset of US-ASCII. UCS can handle strings holding characters from multiple languages, and character classification (uppercase, lowercase, digit, etc.) and operations such as case conversion are unambiguously defined.

For this reason SWI-Prolog has two representations for atoms and string objects (see section 4.23). If the text fits in ISO Latin-1, it is represented as an array of 8-bit characters. Otherwise the text is represented as an array of 32-bit numbers. This representational issue is completely transparent to the Prolog user. Users of the foreign language interface as described in chapter 9 sometimes need to be aware of these issues though.

Character coding comes into view when characters of strings need to be read from or written to file or when they have to be communicated to other software components using the foreign language interface. In this section we only deal with I/O through streams, which includes file I/O as well as I/O through network sockets.

2.18.1 Wide character encodings on streams

Although characters are uniquely coded using the UCS standard internally, streams and files are byte (8-bit) oriented and there are a variety of ways to represent the larger UCS codes in an 8-bit octet stream. The most popular one, especially in the context of the web, is UTF-8. Bytes 0 ... 127 represent simply the corresponding US-ASCII character, while bytes 128 ... 255 are used for multi-byte encoding of characters placed higher in the UCS space. Especially on MS-Windows the 16-bit Unicode standard, represented by pairs of bytes, is also popular.

Prolog I/O streams have a property called *encoding* which specifies the used encoding that influences `get_code/2` and `put_code/2` as well as all the other text I/O predicates.

The default encoding for files is derived from the Prolog flag `encoding`, which is initialised from the environment. If the environment variable `LANG` ends in "UTF-8", this encoding is assumed. Otherwise the default is `text` and the translation is left to the wide-character functions of the C-library. [11] The encoding can be specified explicitly in `load_files/2` for loading Prolog source with an alternative encoding, `open/4` when opening files or using `set_stream/2` on any open stream. For Prolog source files we also provide the `encoding/1` directive that can be used to switch between encodings that are compatible with US-ASCII (`ascii`, `iso_latin_1`, `utf8` and many locales). See also section 3.1.3 for writing Prolog files with non-US-ASCII characters and section 2.15.1 for syntax issues. For additional information and Unicode resources, please visit `http://www.unicode.org/`.

SWI-Prolog currently defines and supports the following encodings:

octet
Default encoding for `binary` streams. This causes the stream to be read and written fully untranslated.

ascii
7-bit encoding in 8-bit bytes. Equivalent to `iso_latin_1`, but generates errors and warnings on encountering values above 127.

iso_latin_1
8-bit encoding supporting many western languages. This causes the stream to be read and written fully untranslated.

text
C-library default locale encoding for text files. Files are read and written using the C-library functions mbrtowc() and wcrtomb(). This may be the same as one of the other locales, notably it may be the same as `iso_latin_1` for western languages and `utf8` in a UTF-8 context.

utf8
Multi-byte encoding of full UCS, compatible with `ascii`. See above.

unicode_be
Unicode *Big Endian*. Reads input in pairs of bytes, most significant byte first. Can only represent 16-bit characters.

unicode_le
Unicode *Little Endian*. Reads input in pairs of bytes, least significant byte first. Can only represent 16-bit characters.

Note that not all encodings can represent all characters. This implies that writing text to a stream may cause errors because the stream cannot represent these characters. The behaviour of a stream on these errors can be controlled using `set_stream/2`. Initially the terminal stream writes the characters using Prolog escape sequences while other streams generate an I/O exception.

---

[11] The Prolog native UTF-8 mode is considerably faster than the generic mbrtowc() one.

BOM: Byte Order Mark

From section 2.18.1, you may have got the impression that text files are complicated. This section deals with a related topic, making life often easier for the user, but providing another worry to the programmer. **BOM** or *Byte Order Marker* is a technique for identifying Unicode text files as well as the encoding they use. Such files start with the Unicode character 0xFEFF, a non-breaking, zero-width space character. This is a pretty unique sequence that is not likely to be the start of a non-Unicode file and uniquely distinguishes the various Unicode file formats. As it is a zero-width blank, it even doesn't produce any output. This solves all problems, or ...

Some formats start off as US-ASCII and may contain some encoding mark to switch to UTF-8, such as the `encoding="UTF-8"` in an XML header. Such formats often explicitly forbid the use of a UTF-8 BOM. In other cases there is additional information revealing the encoding, making the use of a BOM redundant or even illegal.

The BOM is handled by SWI-Prolog `open/4` predicate. By default, text files are probed for the BOM when opened for reading. If a BOM is found, the encoding is set accordingly and the property `bom`(*true*) is available through `stream_property/2`. When opening a file for writing, writing a BOM can be requested using the option `bom`(*true*) with `open/4`.

2.19 System limits

2.19.1 Limits on memory areas

SWI-Prolog has a number of memory areas which are only enlarged to a certain limit. The internal data representation limits the local-, global- and trail-stack to 128 Mbytes on 32 bit processors, or more generally to $2^{\text{bits-per-pointer}-5}$ bytes. Considering that almost all modern hardware can deal with this amount of memory with ease, the default limits are set to their maximum on 32-bit hardware. The representation limits can easily exceed physical memory on 64-bit hardware. The default limits on 64-bit hardware are double of 32-bit hardware, which allows for storing the same amount of (Prolog) data.

The limits can be changed from the command line as well as at runtime using `set_prolog_stack/2`. The table below shows these areas. The first column gives the option name to modify the size of the area. The option character is immediately followed by a number and optionally by a `k` or `m`. With `k` or no unit indicator, the value is interpreted in Kbytes (1024 bytes), with `m`, the value is interpreted in Mbytes (1024 × 1024 bytes).

The PrologScript facility described in section 2.10.2 provides a mechanism for specifying options with the load file. On Windows the default stack sizes are controlled using the Windows registry on the key `HKEY_CURRENT_USER\Software\SWI\Prolog` using the names `localSize`, `globalSize` and `trailSize`. The value is a `DWORD` expressing the default stack size in Kbytes. A GUI for modifying these values is provided using the XPCE package. To use this, start the XPCE manual tools using `manpce/0`, after which you find *Preferences* in the *File* menu.

Considering portability, applications that need to modify the default limits are advised to do so using `set_prolog_stack/2`.

The heap

With the heap, we refer to the memory area used by malloc() and friends. SWI-Prolog uses the area to store atoms, functors, predicates and their clauses, records and other dynamic data. As of SWI-Prolog

Option	Default	Area name	Description
`-L`	128M	**local stack**	The local stack is used to store the execution environments of procedure invocations. The space for an environment is reclaimed when it fails, exits without leaving choice points, the alternatives are cut off with the !/0 predicate or no choice points have been created since the invocation and the last subclause is started (last call optimisation).
`-G`	128M	**global stack**	The global stack is used to store terms created during Prolog's execution. Terms on this stack will be reclaimed by backtracking to a point before the term was created or by garbage collection (provided the term is no longer referenced).
`-T`	128M	**trail stack**	The trail stack is used to store assignments during execution. Entries on this stack remain alive until backtracking before the point of creation or the garbage collector determines they are no longer needed.
`-A`	1M	**argument stack**	The argument stack is used to store one of the Virtual Machine's registers. The amount of space needed on this stack is determined entirely by the depth in which terms are nested in the clauses that constitute the program. Overflow is unlikely.

Table 2.2: Memory areas

2.8.5, no limits are imposed on the addresses returned by malloc() and friends.

On some machines, the runtime stacks described above are allocated using ‘sparse allocation’. Virtual space up to the limit is claimed at startup and committed and released while the area grows and shrinks. On Win32 platforms this is realised using VirtualAlloc() and friends. On Unix systems this is realised using mmap().

2.19.2 Other Limits

Clauses The only limit on clauses is their arity (the number of arguments to the head), which is limited to 1024. Raising this limit is easy and relatively cheap, removing it is harder.

Atoms and Strings SWI-Prolog has no limits on the sizes of atoms and strings. `read/1` and its derivatives however normally limit the number of newlines in an atom or string to 5 to improve error detection and recovery. This can be switched off with `style_check/1`.

The number of atoms is limited to 16777216 (16M) on 32-bit machines. On 64-bit machines this is virtually unlimited. See also section 9.4.2.

Memory areas On 32-bit hardware, SWI-Prolog data is packed in a 32-bit word, which contains both type and value information. The size of the various memory areas is limited to 128 Mb for each of the areas, except for the program heap, which is not limited. On 64-bit hardware there are no meaningful limits.

Nesting of terms Many built-in predicates process nested terms using recursive C functions. Too deeply nested terms generally cause a fatal crash. All these functions avoid recursion on the right-most argument and therefore terms are not limited on the nesting level of the last argument. This notably covers long lists. Most functions use a stack for correct handling of rational trees (cyclic terms). This stack is segmented, where different segments are allocated using malloc(). Overflow causes a non-graceful exit.

Integers On most systems SWI-Prolog is compiled with support for unbounded integers by means of the GNU GMP library. In practice this means that integers are bound by the global stack size. Too large integers cause a `resource_error`. On systems that lack GMP, integers are 64-bit on 32- as well as 64-bit machines.

Integers up to the value of the `max_tagged_integer` Prolog flag are represented more efficiently on the stack. For clauses and records the difference is much smaller.

Floating point numbers Floating point numbers are represented as C-native double precision floats, 64-bit IEEE on most machines.

2.19.3 Reserved Names

The boot compiler (see `-b` option) does not support the module system. As large parts of the system are written in Prolog itself we need some way to avoid name clashes with the user’s predicates, database keys, etc. Like Edinburgh C-Prolog [Pereira, 1986] all predicates, database keys, etc., that should be hidden from the user start with a dollar (`$`) sign.

2.20 SWI-Prolog and 64-bit machines

Most of today's 64-bit platforms are capable of running both 32-bit and 64-bit applications. This asks for some clarifications on the advantages and drawbacks of 64-bit addressing for (SWI-)Prolog.

2.20.1 Supported platforms

SWI-Prolog can be compiled for a 32- or 64-bit address space on any system with a suitable C-compiler. Pointer arithmetic is based on the type (u)intptr_t from `stdint.h`, with suitable emulation on MS-Windows.

2.20.2 Comparing 32- and 64-bits Prolog

Most of Prolog's memory usage consists of pointers. This indicates the primary drawback: Prolog memory usage almost doubles when using the 64-bit addressing model. Using more memory means copying more data between CPU and main memory, slowing down the system.

What then are the advantages? First of all, SWI-Prolog's addressing of the Prolog stacks does not cover the whole address space due to the use of *type tag bits* and *garbage collection flags*. On 32-bit hardware the stacks are limited to 128MB each. This tends to be too low for demanding applications on modern hardware. On 64-bit hardware the limit is 2^{32} times higher, exceeding the addressing capabilities of today's CPUs and operating systems. This implies Prolog can be started with stack sizes that use the full capabilities of your hardware.

Multi-threaded applications profit much more because every thread has its own set of stacks. The Prolog stacks start small and are dynamically expanded (see section 2.19.1). The C-stack is also dynamically expanded, but the maximum size is *reserved* when a thread is started. Using 100 threads at the maximum default C-stack of 8Mb (Linux) costs 800Mb virtual memory![12]

The implications of theoretical performance loss due to increased memory bandwidth implied by exchanging wider pointers depend on the design of the hardware. We only have data for the popular IA32 vs. AMD64 architectures. Here, it appears that the loss is compensated for by an instruction set that has been optimized for modern programming. In particular, the AMD64 has more registers and the relative addressing capabilities have been improved. Where we see a 10% performance degradation when placing the SWI-Prolog kernel in a Unix shared object, we cannot find a measurable difference on AMD64.

2.20.3 Choosing between 32- and 64-bits Prolog

For those cases where we can choose between 32- and 64-bits, either because the hardware and OS support both or because we can still choose the hardware and OS, we give guidelines for this decision.

First of all, if SWI-Prolog needs to be linked against 32- or 64-bit native libraries, there is no choice as it is not possible to link 32- and 64-bit code into a single executable. Only if all required libraries are available in both sizes and there is no clear reason to use either do the different characteristics of Prolog become important.

Prolog applications that require more than the 128MB stack limit provided in 32-bit addressing mode must use the 64-bit edition. Note however that the limits must be doubled to accommodate the same Prolog application.

---

[12] C-recursion over Prolog data structures is removed from most of SWI-Prolog. When removed from all predicates it will often be possible to use lower limits in threads. See `http://www.swi-prolog.org/Devel/CStack.html`

If the system is tight on physical memory, 32-bit Prolog has the clear advantage of using only slightly more than half of the memory of 64-bit Prolog. This argument applies as long as the application fits in the *virtual address space* of the machine. The virtual address space of 32-bit hardware is 4GB, but in many cases the operating system provides less to user applications.

The only standard SWI-Prolog library adding significantly to this calculation is the RDF database provided by the *semweb* package. It uses approximately 80 bytes per triple on 32-bit hardware and 150 bytes on 64-bit hardware. Details depend on how many different resources and literals appear in the dataset as well as desired additional literal indexes.

Summarizing, if applications are small enough to fit comfortably in virtual and physical memory, simply take the model used by most of the applications on the OS. If applications require more than 128MB per stack, use the 64-bit edition. If applications approach the size of physical memory, fit in the 128MB stack limit and fit in virtual memory, the 32-bit version has clear advantages. For demanding applications on 64-bit hardware with more than about 6GB physical memory the 64-bit model is the model of choice.

Initialising and Managing a Prolog Project

3

Prolog text-books give you an overview of the Prolog language. The manual tells you what predicates are provided in the system and what they do. This chapter explains how to run a project. There is no ultimate ‘right’ way to do this. Over the years we developed some practice in this area and SWI-Prolog’s commands are there to support this practice. This chapter describes the conventions and supporting commands.

The first two sections (section 3.1 and section 3.2) only require plain Prolog. The remainder discusses the use of the built-in graphical tools that require the XPCE graphical library installed on your system.

3.1 The project source files

Organisation of source files depends largely on the size of your project. If you are doing exercises for a Prolog course you’ll normally use one file for each exercise. If you have a small project you’ll work with one directory holding a couple of files and some files to link it all together. Even bigger projects will be organised in sub-projects, each using its own directory.

3.1.1 File Names and Locations

File Name Extensions

The first consideration is what extension to use for the source files. Tradition calls for `.pl`, but conflicts with Perl force the use of another extension on systems where extensions have global meaning, such as MS-Windows. On such systems `.pro` is the common alternative.[1]

All versions of SWI-Prolog load files with the extension `.pl` as well as with the registered alternative extension without explicitly specifying the extension. For portability reasons we propose the following convention:

If there is no conflict because you do not use a conflicting application or the system does not force a unique relation between extension and application, use `.pl`.

With a conflict choose `.pro` and use this extension for the files you want to load through your file manager. Use `.pl` for all other files for maximal portability.

Project Directories

Large projects are generally composed of sub-projects, each using its own directory or directory structure. If nobody else will ever touch your files and you use only one computer, there is little to worry

---

[1] On MS-Windows, the alternative extension is stored in the registry key `HKEY_CURRENT_USER/Software/SWI/Prolog/fileExtension` or `HKEY_LOCAL_MACHINE/Software/SWI/Prolog/fileExte`

about, but this is rarely the case with a large project.

To improve portability, SWI-Prolog uses the POSIX notation for filenames, which uses the forward slash (`/`) to separate directories. Just before reaching the file system, SWI-Prolog uses `prolog_to_os_filename/2` to convert the filename to the conventions used by the hosting operating system. It is *strongly* advised to write paths using the `/`, especially on systems using the `\` for this purpose (MS-Windows). Using `\` violates the portability rules and requires you to *double* the `\` due to the Prolog quoted-atom escape rules.

Portable code should use `prolog_to_os_filename/2` to convert computed paths into system paths when constructing commands for `shell/1` and friends.

Sub-projects using search paths

Thanks to Quintus, Prolog adapted an extensible mechanism for searching files using `file_search_path/2`. This mechanism allows for comfortable and readable specifications.

Suppose you have extensive library packages on graph algorithms, set operations and GUI primitives. These sub-projects are likely candidates for re-use in future projects. A good choice is to create a directory with sub-directories for each of these sub-projects.

Next, there are three options. One is to add the sub-projects to the directory hierarchy of the current project. Another is to use a completely dislocated directory. Third, the sub-project can be added to the SWI-Prolog hierarchy. Using local installation, a typical `file_search_path/2` is:

```
:- prolog_load_context(directory, Dir),
   asserta(user:file_search_path(myapp, Dir)).

user:file_search_path(graph, myapp(graph)).
user:file_search_path(ui,    myapp(ui)).
```

When using sub-projects in the SWI-Prolog hierarchy, one should use the path alias `swi` as basis. For a system-wide installation, use an absolute path.

Extensive sub-projects with a small well-defined API should define a load file with calls to `use_module/1` to import the various library components and export the API.

3.1.2 Project Special Files

There are a number of tasks you typically carry out on your project, such as loading it, creating a saved state, debugging it, etc. Good practice on large projects is to define small files that hold the commands to execute such a task, name this file after the task and give it a file extension that makes starting easy (see section 3.1.1). The task *load* is generally central to these tasks. Here is a tentative list:

- *`load.pl`*
  Use this file to set up the environment (Prolog flags and file search paths) and load the sources. Quite commonly this file also provides convenient predicates to parse command-line options and start the application.

- *`run.pl`*
  Use this file to start the application. Normally it loads `load.pl` in silent-mode, and calls one of the starting predicates from `load.pl`.

- *`save.pl`*
  Use this file to create a saved state of the application by loading `load.pl` and calling `qsave_program/2` to generate a saved state with the proper options.

- *`debug.pl`*
  Loads the program for debugging. In addition to loading `load.pl` this file defines rules for `portray/1` to modify printing rules for complex terms and customisation rules for the debugger and editing environment. It may start some of these tools.

3.1.3 International source files

As discussed in section 2.18, SWI-Prolog supports international character handling. Its internal encoding is UNICODE. I/O streams convert to/from this internal format. This section discusses the options for source files not in US-ASCII.

SWI-Prolog can read files in any of the encodings described in section 2.18. Two encodings are of particular interest. The `text` encoding deals with the current *locale*, the default used by this computer for representing text files. The encodings `utf8`, `unicode_le` and `unicode_be` are *UNICODE* encodings: they can represent—in the same file—characters of virtually any known language. In addition, they do so unambiguously.

If one wants to represent non US-ASCII text as Prolog terms in a source file, there are several options:

- *Use escape sequences*
  This approach describes NON-ASCII as sequences of the form \octal\\. The numerical argument is interpreted as a UNICODE character.[2] The resulting Prolog file is strict 7-bit US-ASCII, but if there are many NON-ASCII characters it becomes very unreadable.

- *Use local conventions*
  Alternatively the file may be specified using local conventions, such as the EUC encoding for Japanese text. The disadvantage is portability. If the file is moved to another machine, this machine must use the same *locale* or the file is unreadable. There is no elegant way if files from multiple locales must be united in one application using this technique. In other words, it is fine for local projects in countries with uniform locale conventions.

- *Using UTF-8 files*
  The best way to specify source files with many NON-ASCII characters is definitely the use of UTF-8 encoding. Prolog can be notified of this encoding in two ways, using a UTF-8 *BOM* (see section 2.18.1) or using the directive `:- encoding(utf8).` Many of today's text editors, including PceEmacs, are capable of editing UTF-8 files. Projects that were started using local conventions can be re-coded using the Unix `iconv` tool or often using commands offered by the editor.

3.2 Using modules

Modules have been debated fiercely in the Prolog world. Despite all counter-arguments we feel they are extremely useful because:

---

[2] To my knowledge, the ISO escape sequence is limited to 3 octal digits, which means most characters cannot be represented.

- *They hide local predicates*
  This is the reason they were invented in the first place. Hiding provides two features. They allow for short predicate names without worrying about conflicts. Given the flat name-space introduced by modules, they still require meaningful module names as well as meaningful names for exported predicates.

- *They document the interface*
  Possibly more important than avoiding name conflicts is their role in documenting which part of the file is for public usage and which is private. When editing a module you may assume you can reorganise anything except the name and the semantics of the exported predicates without worrying.

- *They help the editor*
  The PceEmacs built-in editor does on-the-fly cross-referencing of the current module, colouring predicates based on their origin and usage. Using modules, the editor can quickly find out what is provided by the imported modules by reading just the first term. This allows it to indicate in real-time which predicates are not used or not defined.

Using modules is generally easy. Only if you write meta-predicates (predicates reasoning about other predicates) that are exported from a module is a good understanding required of the resolution of terms to predicates inside a module. Here is a typical example from `readutil`.

```
:- module(read_util,
          [ read_line_to_codes/2,       % +Fd, -Codes
            read_line_to_codes/3,       % +Fd, -Codes, ?Tail
            read_stream_to_codes/2,     % +Fd, -Codes
            read_stream_to_codes/3,     % +Fd, -Codes, ?Tail
            read_file_to_codes/3,       % +File, -Codes, +Options
            read_file_to_terms/3        % +File, -Terms, +Options
          ]).
```

3.3 The test-edit-reload cycle

SWI-Prolog does not enforce the use of a particular editor for writing Prolog source code. Editors are complicated programs that must be mastered in detail for real productive programming. If you are familiar with a specific editor you should not be forced to change. You may specify your favourite editor using the Prolog flag `editor`, the environment variable `EDITOR` or by defining rules for `prolog_edit:edit_source/1` (see section 4.5).

The use of a built-in editor, which is selected by setting the Prolog flag `editor` to `pce_emacs`, has advantages. The XPCE *editor* object, around which the built-in PceEmacs is built, can be opened as a Prolog stream allowing analysis of your source by the real Prolog system.

3.3.1 Locating things to edit

The central predicate for editing something is `edit/1`, an extensible front-end that searches for objects (files, predicates, modules, as well as XPCE classes and methods) in the Prolog database.

If multiple matches are found it provides a choice. Together with the built-in completion on atoms bound to the `TAB` key this provides a quick way to edit objects:

```
?- edit(country).
Please select item to edit:

  1 chat:country/10      '/staff/jan/lib/prolog/chat/countr.pl':16
  2 chat:country/1       '/staff/jan/lib/prolog/chat/world0.pl':72

Your choice?
```

3.3.2 Editing and incremental compilation

One of the nice features of Prolog is that the code can be modified while the program is running. Using pure Prolog you can trace a program, find it is misbehaving, enter a *break environment*, modify the source code, reload it and finally do *retry* on the misbehaving predicate and try again. This sequence is not uncommon for long-running programs. For faster programs one will normally abort after understanding the misbehaviour, edit the source, reload it and try again.

One of the nice features of SWI-Prolog is the availability of `make/0`, a simple predicate that checks all loaded source files to see which ones you have modified. It then reloads these files, considering the module from which the file was loaded originally. This greatly simplifies the trace-edit-verify development cycle. For example, after the tracer reveals there is something wrong with `prove/3`, you do:

```
?- edit(prove).
```

Now edit the source, possibly switching to other files and making multiple changes. After finishing, invoke `make/0`, either through the editor UI (Compile/Make (`Control-C Control-M`)) or on the top-level, and watch the files being reloaded.[3]

```
?- make.
% show compiled into photo_gallery 0.03 sec, 3,360 bytes
```

3.4 Using the PceEmacs built-in editor

3.4.1 Activating PceEmacs

Initially `edit/1` uses the editor specified in the `EDITOR` environment variable. There are two ways to force it to use the built-in editor. One is to set the Prolog flag `editor` to `pce_emacs` and the other is by starting the editor explicitly using the `emacs/[0,1]` predicates.

---

[3] Watching these files is a good habit. If expected files are not reloaded you may have forgotten to save them from the editor or you may have been editing the wrong file (wrong directory).

3.4.2 Bluffing through PceEmacs

PceEmacs closely mimics Richard Stallman's GNU-Emacs commands, adding features from modern window-based editors to make it more acceptable for beginners.[4]

At the basis, PceEmacs maps keyboard sequences to methods defined on the extended *editor* object. Some frequently used commands are, with their key-binding, presented in the menu bar above each editor window. A complete overview of the bindings for the current *mode* is provided through Help/Show key bindings (`Control-h Control-b`).

Edit modes

Modes are the heart of (Pce)Emacs. Modes define dedicated editing support for a particular kind of (source-)text. For our purpose we want *Prolog mode*. There are various ways to make PceEmacs use Prolog mode for a file.

- *Using the proper extension*
  If the file ends in `.pl` or the selected alternative (e.g. `.pro`) extension, Prolog mode is selected.
- *Using* `#!/path/to/pl`
  If the file is a *Prolog Script* file, starting with the line `#!/path/to/pl options -s`, Prolog mode is selected regardless of the extension.
- *Using* `-*- Prolog -*-`
  If the above sequence appears in the first line of the file (inside a Prolog comment) Prolog mode is selected.
- *Explicit selection*
  Finally, using File/Mode/Prolog (`y`)ou can switch to Prolog mode explicitly.

Frequently used editor commands

Below we list a few important commands and how to activate them.

- *Cut/Copy/Paste*
  These commands follow Unix/X11 traditions. You're best suited with a three-button mouse. After selecting using the left-mouse (double-click uses word-mode and triple line-mode), the selected text is *automatically* copied to the clipboard (X11 primary selection on Unix). *Cut* is achieved using the DEL key or by typing something else at the location. *Paste* is achieved using the middle-mouse (or wheel) button. If you don't have a middle-mouse button, pressing the left- and right-button at the same time is interpreted as a middle-button click. If nothing helps, there is the Edit/Paste menu entry. Text is pasted at the caret location.
- *Undo*
  Undo is bound to the GNU-Emacs Control-_ as well as the MS-Windows Control-Z sequence.
- *Abort*
  Multi-key sequences can be aborted at any stage using Control-G.

---

[4] Decent merging with MS-Windows control-key conventions is difficult as many conflict with GNU-Emacs. Especially the cut/copy/paste commands conflict with important GNU-Emacs commands.

- *Find*
  Find (Search) is started using Control-S (forward) or Control-R (backward). PceEmacs implements *incremental search*. This is difficult to use for novices, but very powerful once you get the clue. After one of the above start keys, the system indicates search mode in the status line. As you are typing the search string, the system searches for it, extending the search with every character you type. It illustrates the current match using a green background.

  If the target cannot be found, PceEmacs warns you and no longer extends the search string.[5] During search, some characters have special meaning. Typing anything but these characters commits the search, re-starting normal edit mode. Special commands are:

  Control-S
  : Search forwards for next.

  Control-R
  : Search backwards for next.

  Control-W
  : Extend search to next word boundary.

  Control-G
  : Cancel search, go back to where it started.

  ESC
  : Commit search, leaving caret at found location.

  Backspace
  : Remove a character from the search string.

- *Dynamic Abbreviation*
  Also called *dabbrev*, dynamic abbreviation is an important feature of Emacs clones to support programming. After typing the first few letters of an identifier, you may press Alt-/, causing PceEmacs to search backwards for identifiers that start the same and use it to complete the text you typed. A second Alt-/ searches further backwards. If there are no hits before the caret, it starts searching forwards. With some practice, this system allows for entering code very fast with nice and readable identifiers (or other difficult long words).

- *Open (a file)*
  Is called File/Find file (`Control-x Control-f`). By default the file is loaded into the current window. If you want to keep this window, press Alt-s or click the little icon at the bottom left to make the window *sticky*.

- *Split view*
  Sometimes you want to look at two places in the same file. To do this, use Control-x 2 to create a new window pointing to the same file. Do not worry, you can edit as well as move around in both. Control-x 1 kills all other windows running on the same file.

These are the most commonly used commands. In section 3.4.3 we discuss specific support for dealing with Prolog source code.

---

[5] GNU-Emacs keeps extending the string, but why? Adding more text will not make it match.

3.4.3 Prolog Mode

In the previous section (section 3.4.2) we explained the basics of PceEmacs. Here we continue with Prolog-specific functionality. Possibly the most interesting is *Syntax highlighting*. Unlike most editors where this is based on simple patterns, PceEmacs syntax highlighting is achieved by Prolog itself actually reading and interpreting the source as you type it. There are three moments at which PceEmacs checks (part of) the syntax.

- *After typing a* `.`
  After typing a `.` that is not preceded by a *symbol* character, the system assumes you completed a clause, tries to find the start of this clause and verifies the syntax. If this process succeeds it colours the elements of the clause according to the rules given below. Colouring is done using information from the last full check on this file. If it fails, the syntax error is displayed in the status line and the clause is not coloured.

- *After the command* Control-c Control-s
  Acronym for **C**heck **S**yntax, it performs the same checks as above for the clause surrounding the caret. On a syntax error, however, the caret is moved to the expected location of the error.[6]

- *After pausing for two seconds*
  After a short pause (2 seconds), PceEmacs opens the edit buffer and reads it as a whole, creating an index of defined, called, dynamic, imported and exported predicates. After completing this, it re-reads the file and colours all clauses and calls with valid syntax.

- *After typing* Control-l Control-l
  The Control-l command re-centers the window (scrolls the window to make the caret the center of the window). Typing this command twice starts the same process as above.

The colour schema itself is defined in `emacs/prolog_colour`. The colouring can be extended and modified using multifile predicates. Please check this source file for details. In general, underlined objects have a popup (right-mouse button) associated with common commands such as viewing the documentation or source. **Bold** text is used to indicate the definition of objects (typically predicates when using plain Prolog). Other colours follow intuitive conventions. See table 3.4.3.

Layout support Layout is not 'just nice', it is *essential* for writing readable code. There is much debate on the proper layout of Prolog. PceEmacs, being a rather small project, supports only one particular style for layout.[7] Below are examples of typical constructs.

```
head(arg1, arg2).

head(arg1, arg2) :- !.

head(Arg1, arg2) :- !,
        call1(Arg1).

head(Arg1, arg2) :-
```

---

[6] In most cases the location where the parser cannot proceed is further down the file than the actual error location.

[7] Defined in Prolog in the file `emacs/prolog_mode`, you may wish to extend this. Please contribute your extensions!

Clauses	
Blue bold	Head of an exported predicate
Red bold	Head of a predicate that is not called
Black bold	Head of remaining predicates
Calls in the clause body	
Blue	Call to built-in or imported predicate
Red	Call to undefined predicate
Purple	Call to dynamic predicate
Other entities	
Dark green	Comment
Dark blue	Quoted atom or string
Brown	Variable

Table 3.1: Colour conventions

```
        (   if(Arg1)
        ->  then
        ;   else
        ).

head(Arg1) :-
        (   a
        ;   b
        ).

head :-
        a(many,
          long,
          arguments(with,
                    many,
                    more),
          and([ a,
                long,
                list,
                with,
                a,
              | tail
              ])).
```

PceEmacs uses the same conventions as GNU-Emacs. The TAB key indents the current line according to the syntax rules. Alt-q indents all lines of the current clause. It provides support for head, calls (indented 1 tab), if-then-else, disjunction and argument lists broken across multiple lines as illustrated above.

Finding your way around

The command Alt-. extracts name and arity from the caret location and jumps (after conformation or edit) to the definition of the predicate. It does so based on the source-location database of loaded predicates also used by `edit/1`. This makes locating predicates reliable if all sources are loaded and up-to-date (see `make/0`).

In addition, references to files in `use_module/[1,2]`, `consult/1`, etc. are red if the file cannot be found and underlined blue if the file can be loaded. A popup allows for opening the referenced file.

3.5 The Graphical Debugger

SWI-Prolog offers two debuggers. One is the traditional text console-based 4-port Prolog tracer and the other is a window-based source level debugger. The window-based debugger requires XPCE installed. It operates based on the `prolog_trace_interception/4` hook and other low-level functionality described in chapter B.

Window-based tracing provides a much better overview due to the eminent relation to your source code, a clear list of named variables and their bindings as well as a graphical overview of the call and choice-point stack. There are some drawbacks though. Using a textual trace on the console, one can scroll back and examine the past, while the graphical debugger just presents a (much better) overview of the current state.

3.5.1 Invoking the window-based debugger

Whether the text-based or window-based debugger is used is controlled using the predicates `guitracer/0` and `noguitracer/0`. Entering debug mode is controlled using the normal predicates for this: `trace/0` and `spy/1`. In addition, PceEmacs prolog mode provides the command Prolog/Break at (`Control-c b`) to insert a break-point at a specific location in the source code.

The graphical tracer is particulary useful for debugging threads. The tracer must be loaded from the `main` thread before it can be used from a background thread.

guitracer

This predicate installs the above-mentioned hooks that redirect tracing to the window-based environment. No window appears. The debugger window appears as actual tracing is started through `trace/0`, by hitting a spy-point defined by `spy/1` or a break-point defined using the PceEmacs command Prolog/Break at (`Control-c b`).

noguitracer

Disable the hooks installed by `guitracer/0`, reverting to normal text console-based tracing.

gtrace

Utility defined as `guitracer,trace`.

gdebug

Utility defined as `guitracer,debug`.

gspy(+*Predicate***)**

Utility defined as `guitracer,spy(Predicate).`

3.6 The Prolog Navigator

Another tool is the *Prolog Navigator*. This tool can be started from PceEmacs using the command Browse/Prolog navigator, from the GUI debugger or using the programmatic IDE interface described in section 3.8.

3.7 Cross referencer

A cross-referencers is a tool examining the caller-callee relation between predicates and using this information to explicate dependency relations between source files, find calls to non-existing predicates and predicates for which no callers can be found. Cross-referencing is useful during program development, reorganisation, cleanup, porting and other program maintenance tasks. The dynamic nature of Prolog makes the task non-trivial. Goals can be created dynamically `call/1` after construction of a goal term. Abtract interpretation can find some of such calls, but the ultimately they can come from external communication, making it completely impossible to predict the callee. In other words, the cross-referencer has only partial understanding of the program and its results are necessarily incomplete. Still, it provides valuable information to the developer.

SWI-Prolog's cross-referencer is split into two parts. The standard Prolog library `prolog_xref` is an extensible library for information gathering described in section A.21 and the XPCE

library `pce_xref` provides a graphical frontend for the cross-referencer described here. We demonstrate the tool on CHAT80, a natural language question and answer system by Fernando C.N. Pereira and David H.D. Warren.

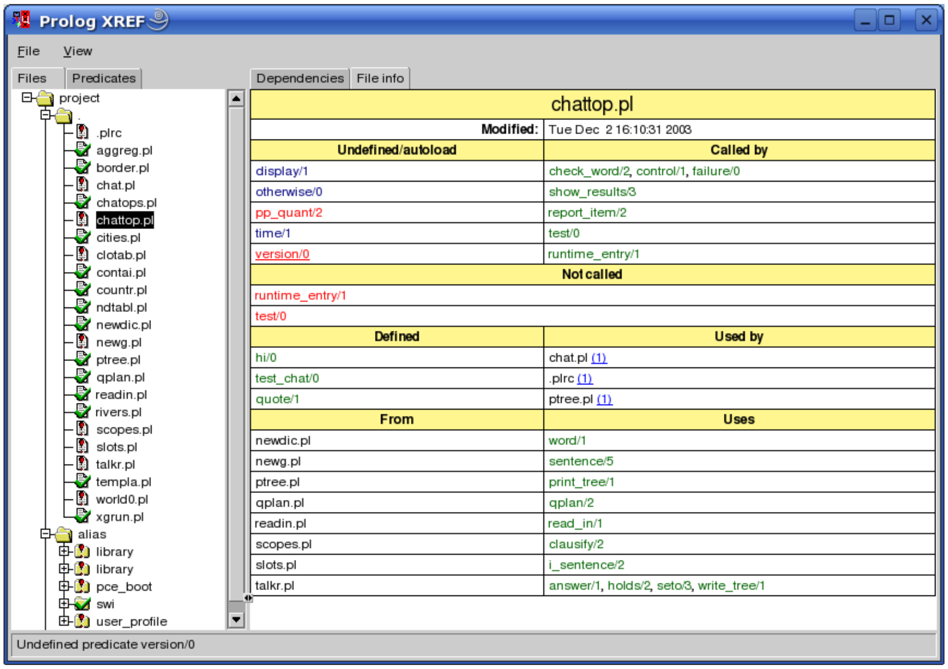
gxref

Run cross-referencer on all currently loaded files and present a graphical overview of the result. As the predicate operates on the currently loaded application it must be run after loading the application.

The **left window** (see figure 3.1 provides browsers for loaded files and predicates. To avoid long file paths the file hierarchy has three main branches. The first is the current directory holding the sources. The second is marked `alias` and below it are the file-search-path aliases (see `file_search_path/2` and `absolute_file_name/3`). Here you find files loaded from the system as well as modules of the program loaded from other locations using file search path. All loaded files that fall outside these categories are below the last branch called `/`. File where the system found suspicious dependencies are marked with an exclamation mark. This also holds for directories holding such files. Clicking on a file opens a *File info* window in the right pane.

The **File info** window shows a file, its main properties, its undefined and not-called predicates and its import- and export relations to other files in the project. Both predicates and files can be opened by clicking on them. The number of callers in a file for a certain predicate is indicated with a blue underlined number. A left-click will open a list and allows to edit the calling predicate.

The **Dependencies** (see figure 3.2) window displays a graphical overview of dependencies between files. Using the background menu a complete graph of the project can be created. It is also possible to drag files onto the graph window and use the menu on the nodes to incrementally expand the graph. The underlined blue text indicates the number of predicates used in the destination file. Left-clicking opens a menu to open the definition or select one of the callers.



chattop.pl

Modified: Tue Dec 2 16:10:31 2003

Undefined/autoload	**Called by**
display/1	check_word/2, control/1, failure/0
otherwise/0	show_results/3
pp_quant/2	report_item/2
time/1	test/0
version/0	runtime_entry/1

Not called

runtime_entry/1

test/0

Defined	**Used by**
hi/0	chat.pl (1)
test_chat/0	.plrc (1)
quote/1	ptree.pl (1)

From	**Uses**
newdic.pl	word/1
newg.pl	sentence/5
ptree.pl	print_tree/1
qplan.pl	qplan/2
readin.pl	read_in/1
scopes.pl	clausify/2
slots.pl	i_sentence/2
talkr.pl	answer/1, holds/2, seto/3, write_tree/1

Undefined predicate version/0

Figure 3.1: File info for `chattop.pl`, part of CHAT80

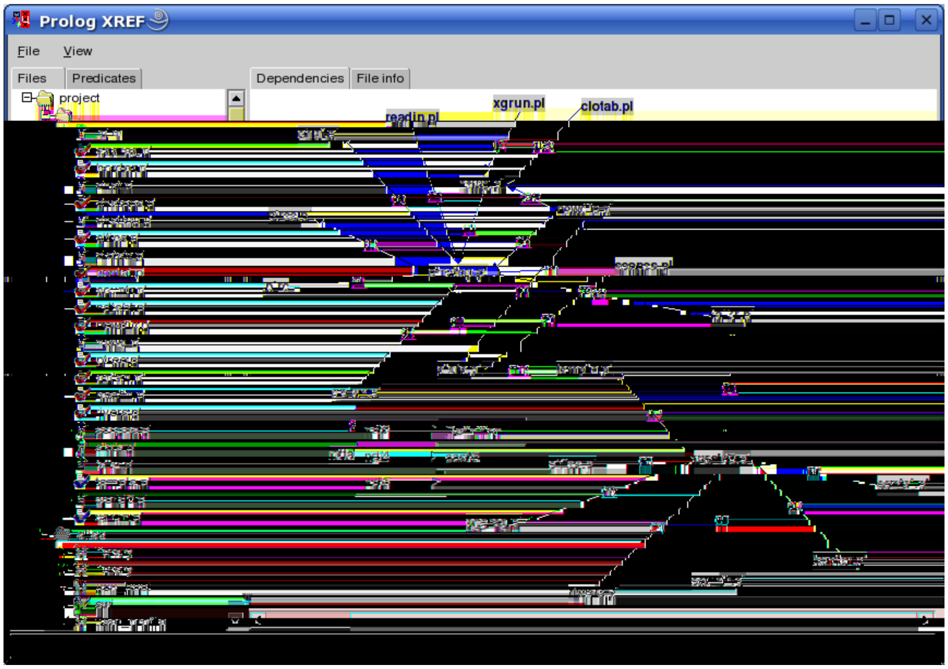


Figure 3.2: Dependencies between source files of CHAT80

Module and non-module files The cross-referencer threads module and non-module project files differently. Module files have explicit import and export relations and the tool shows the usage and consistency of the relations. Using the menu-command Header the tool creates a consistent import list for the module that can be included in the file. The tool computes the dependency relations between the non-module files. If the user wishes to convert the project into a module-based one the Header command generates an appropriate module header and import list. Note that the cross-referencer may have missed dependencies and does not deal with meta-predicates defined in one module and called in another. Such problems must be resolved manually.

Settings The following settings can be controlled from the settings menu:

Warn autoload
By default disabled. If enabled, modules that require predicates to be autoloaded are flagged with a warning and the file info window of a module shows the required autoload predicates.

Warn not called
If enabled (default), the file-overview shows an alert icon for files that have predicates that are not called.

3.8 Accessing the IDE from your program

Over the years a collection of IDE components have been developed, each with its own interface. In addition, some of these components require each other, and loading IDE components must be on demand to avoid the IDE being part of a saved state (see `qsave_program/2`). For this reason, access to the IDE is concentrated on a single interface called `prolog_ide/1`:

prolog_ide(+*Action***)**
This predicate ensures the IDE-enabling XPCE component is loaded, creates the XPCE class *prolog_ide* and sends *Action* to its one and only instance `\index{@prolog_ide}\objectname{prolog_ide}`. *Action* is one of the following:

open_navigator(+*Directory***)**
Open the Prolog Navigator (see section 3.6) in the given *Directory*.

open_debug_status
Open a window to edit spy- and trace-points.

open_query_window
Open a little window to run Prolog queries from a GUI component.

thread_monitor
Open a graphical window indicating existing threads and their status.

debug_monitor
Open a graphical front-end for the `debug` library that provides an overview of the topics and catches messages.

xref
Open a graphical front-end for the cross-referencer that provides an overview of predicates and their callers.

3.9 Summary of the IDE

The SWI-Prolog development environment consists of a number of interrelated but not (yet) integrated tools. Here is a list of the most important features and tips.

- *Atom completion*
  The console[8] completes a partial atom on the TAB key and shows alternatives on the command Alt-?.
- *Use* `edit/1` *for finding locations*
  The command `edit/1` takes the name of a file, module, predicate or other entity registered through extensions and starts the user's preferred editor at the right location.
- *Select editor*
  External editors are selected using the `EDITOR` environment variable, by setting the Prolog flag `editor`, or by defining the hook prolog_edit:edit_source/1.
- *Update Prolog after editing*
  Using `make/0`, all files you have edited are re-loaded.
- *PceEmacs*
  Offers syntax highlighting and checking based on real-time parsing of the editor's buffer, layout support and navigation support.
- *Using the graphical debugger*
  The predicates `guitracer/0` and `noguitracer/0` switch between traditional text-based and window-based debugging. The tracer is activated using the `trace/0`, `spy/1` or menu items from PceEmacs or the Prolog Navigator.
- *The Prolog Navigator*
  Shows the file structure and structure inside the file. It allows for loading files, editing, setting spy-points, etc.

---

[8] On Windows this is realised by swipl-win.exe, on Unix through the GNU readline library, which is included automatically when found by `configure`.

4

Built-in predicates

4.1 Notation of Predicate Descriptions

We have tried to keep the predicate descriptions clear and concise. First the predicate name is printed in bold face, followed by the arguments in italics. Arguments are preceded by a mode indicator There is no complete agreement on mode indicators in the Prolog community. We use the following definitions:[1]

+	Argument must be fully instantiated to a term that satisfies the required argument type. Think of the argument as *input*.
-	Argument must be unbound. Think of the argument as *output*.
?	Argument must be bound to a *partial term* of the indicated type. Note that a variable is a partial term for any type. Think of the argument as either *input* or *output* or *both* input and output. E.g. In `stream_property(S, reposition(Bool))`, the `reposition` part of the term is input and the uninstantiated *Bool* is output.
:	Argument is a meta-argument. Implies +. See section 5 for more information on module-handing.
@	Argument is not further instantiated. Typically used for type-tests.
!	Argument contains a mutable structure that may be modified using `setarg/3` or `nb_setarg/3`.

Referring to a predicate in running text is done using a *predicate indicator*. The canonical and most generic form of a predicate indicator is a term ⟨*module*⟩:⟨*name*⟩/⟨*arity*⟩. If the module is irrelevant (built-in predicate) or can be inferred from the context it is often omitted. Compliant to the ISO standard draft on DCG (see section 4.12), SWI-Prolog also allows for [⟨*module*⟩]:⟨*name*⟩//⟨*arity*⟩ to refer to a grammar rule. For all non-negative arity, ⟨*name*⟩//⟨*arity*⟩ is the same as ⟨*name*⟩/¡arity+2¿, regardless on whether or not the referenced predicate is defined or can be used as a grammar rule. The //-notation can be used in all places that traditionally allow for a predicate indicator, e.g., the module declaration, `spy/1`, and `dynamic/1`.

4.2 Character representation

In traditional (Edinburgh-) Prolog, characters are represented using *character-codes*. Character codes are integer indices into a specific character set. Traditionally the character set was 7-bits US-ASCII.

---

[1] These definitions are taken from PlDoc. The current manual has only one mode declaration per predicate and therefore predicates with mode (+,-) and (-,+) are described as (?,?). The `@`-mode is often replaced by +.

8-bit character sets have been allowed for a long time, providing support for national character sets, of which iso-latin-1 (ISO 8859-1) is applicable to many western languages.

ISO Prolog introduces three types, two of which are used for characters and one for accessing binary streams (see `open/4`). These types are:

- *code*
  A *character-code* is an integer representing a single character. As files may use multi-byte encoding for supporting different character sets (utf-8 encoding for example), reading a code from a text-file is in general not the same as reading a byte.
- *char*
  Alternatively, characters may be represented as *one-character-atoms*. This is a natural representation, hiding encoding problems from the programmer as well as providing much easier debugging.
- *byte*
  Bytes are used for accessing binary-streams.

In SWI-Prolog, character-codes are *always* the Unicode equivalent of the encoding. I.e., if `get_code/1` reads from a stream encoded as `KOI8-R` (used for the Cyrillic alphabet), it returns the corresponding Unicode code-points. Similar, assembling or deassembling atoms using `atom_codes/2` interprets the codes as Unicode points. See section 2.18.1 for details.

To ease the pain of the two character representations (code and char), SWI-Prolog's built-in predicates dealing with character-data work as flexible as possible: they accept data in any of these formats as long as the interpretation is unambiguous. In addition, for output arguments that are instantiated, the character is extracted before unification. This implies that the following two calls are identical, both testing whether the next input characters is an `a`.

```
peek_code(Stream, a).
peek_code(Stream, 97).
```

The two character representations are handled by a large number of built-in predicates, all of which are ISO-compatible. For converting between code and character there is `char_code/2`. For breaking atoms and numbers into characters are are `atom_chars/2`, `atom_codes/2`, `number_codes/2` and `number_chars/2`. For character I/O on streams there is `get`

A traditional Prolog source file contains Prolog clauses and directives, but no *module-declaration*. They are normally loaded using `consult/1` or `ensure_loaded/1`. Currently, a non-module file can only be loaded into a single module.[2]

A module Prolog source file starts with a module declaration. The subsequent Prolog code is loaded into the specified module and only the *public* predicates are made available to the context loading the module. Module files are normally loaded using `use_module/[1,2]`. See chapter 5 for details.

An include Prolog source file is loaded using the `include/1` directive and normally contains only directives.

Prolog source-files are located using `absolute_file_name/3` with the following options:

```
locate_prolog_file(Spec, Path) :-
        absolute_file_name(Spec,
                           [ file_type(prolog),
                             access(read)
                           ],
                           Path).
```

The `file_type`(*prolog*) option is used to determine the extension of the file using `prolog_file_type/2`. The default extension is `.pl`. *Spec* allows for the *path-alias* construct defined by `absolute_file_name/3`. The most commonly used path-alias is `library`(*LibraryFile*). The example below loads the library file `ordsets.pl` (containing predicates for manipulating ordered sets).

```
:- use_module(library(ordsets)).
```

SWI-Prolog recognises grammar rules (DCG) as defined in [Clocksin & Melish, 1987]. The user may define additional compilation of the source file by defining the dynamic predicates `term_expansion/2` and `goal_expansion/2`. Transformations by `term_expansion/2` overrule the systems grammar rule transformations. It is not allowed to use `assert/1`, `retract/1` or any other database predicate in `term_expansion/2` other than for local computational purposes.[3] Code that needs to create additional clauses must use `compile_aux_clauses/1`. See `library(apply_macros)` for an example.

A *directive* is an instruction to the compiler. Directives are used to set (predicate) properties (see section 4.14), set flags (see `set_prolog_fag/2`) and load files (this section). Directives are terms of the form `:-` ⟨*term*⟩.. Here are some examples:

```
:- use_module(library(lists)).
:- dynamic
        store/2.                % Name, Value
```

---

[2] This limitation may be lifted in the future. Existing limitations in SWI-Prolog's source-code administration make this non-trivial.

[3] It does work for normal loading, but not for `qcompile/1`.

Predicate	if	must_be_module	import
`consult/1`	`true`	`false`	all
`ensure_loaded/1`	`not_loaded`	`false`	all
`use_module/1`	`not_loaded`	`true`	all
`use_module/2`	`not_loaded`	`true`	specified
`reexport/1`	`not_loaded`	`true`	all
`reexport/2`	`not_loaded`	`true`	specified

Table 4.1: Properties of the file-loading predicates. The *import* column specifies what is imported if the loaded file is a module file.

The directive `initialization/1` can be used to run arbitrary Prolog goals. The specified goal is started *after* loading the file in which it appears has completed.

SWI-Prolog compiles code as it is read from the file and directives are executed as *goals*. This implies that directives may call any predicate that has been defined before the point where the directive appears. It also accepts `?-` ⟨*term*⟩. as a synonym.

SWI-Prolog does not have a separate `reconsult/1` predicate. Reconsulting is implied automatically by the fact that a file is consulted which is already loaded.

Advanced topics are handled in subsequent sections: mutual dependend files (section 4.3.2), multi-threaded loading (section 4.3.2) and reloading running code (section 4.3.2).

The core of the family of loading predicates is `load_files/2`. The predicates `consult/1`, `ensure_loaded/1`, `use_module/1`, `use_module/2` and `reexport/1` pass the file argument directly to `load_files/2` and pass additional options as expressed in the table 4.1:

load_files(*:Files, +Options***)**

The predicate `load_files/2` is the parent of all the other loading predicates except for `include/1`. It currently supports a subset of the options of Quintus `load_files/2`. *Files* is either a single source-file, or a list of source-files. The specification for a source-file is handed to `absolute_file_name/2`. See this predicate for the supported expansions. *Options* is a list of options using the format

OptionName(*OptionValue*)

The following options are currently supported:

autoload(*Bool***)**

If `true` (default `false`), indicate this load is a *demand* load. This implies that, depending on the setting of the Prolog flag `verbose_autoload` the load-action is printed at level `informational` or `silent`. See also `print_message/2` and `current_prolog_flag/2`.

derived_from(*File***)**

Indicate that the loaded file is derived from *File*. Used by `make/0` to time-check and load the original file rather than the derived file.

encoding(*Encoding***)**

Specify the way characters are encoded in the file. Default is taken from the Prolog flag `encoding`. See section 2.18.1 for details.

expand(*Bool*)

If `true`, run the filenames through `expand_file_name/2` and load the returned files. Default is `false`, except for `consult/1` which is intended for interactive use. Flexible location of files is defined by `file_search_path/2`.

format(*+Format*)

Used to specify the file format if data is loaded from a stream using the `stream`(*Stream*) option. Default is `source`, loading Prolog source text. If `qlf`, load QLF data (see `qcompile/1`).

if(*Condition*)

Load the file only if the specified condition is satisfied. The value `true` loads the file unconditionally, `changed` loads the file if it was not loaded before, or has been modified since it was loaded the last time, `not_loaded` loads the file if it was not loaded before.

imports(*Import*)

Specify what to import from the loaded module. The default for `use_module/1` is `all`. *Import* is passed from the second argument of `use_module/2`. Traditionally it is a list of predicate indicators to import. As part of the SWI-Prolog/YAP integration, we also support *Pred* as *Name* to import a predicate under another name. Finally, *Import* can be a term `except`(*Exceptions*), where *Exceptions* is a list of predicate indicators that specify predicates that are *not* imported or *Pred* as *Name* terms to denote renamed predicates. See also `reexport/2` and `use_module/2`.[4]

If *Import* equals `all`, all operators are imported as well. Otherwise, operators are *not* imported. Operators can be imported selectively by adding terms `op`(*Pri,Assoc,Name*) to the *Imports* list. If such a term is encountered, all exported operators that unify with this term are imported. Typically, this construct will be used with all arguments unbound to import all operators or with only *Name* bound to import a particular operator.

modified(*TimeStamp*)

Claim that the source was load modified at *TimeStamp* without checking the source. This option is intended to be used together with the `stream`(*Input*) option, for example after extracting the time from an HTTP server or database.

must_be_module(*Bool*)

If `true`, raise an error if the file is not a module file. Used by `use_module/[1,2]`.

qcompile(*Atom*)

How to deal with quick-load-file compilation by `qcompile/1`. Values are

never

Default. Do not use qcompile, unless called explicitely

auto

Use qcompile for all writeable files. See comment below.

large

Use qcompile if the file is 'large'. Currently, files larger than 100 Kbytes are considered large.

part

If this `load_file/2` appears in a directive of a file that is compiled into Quick

---

[4] BUG: *Name/Arity* as *NewName* is currently implemented using a *link clause*. This harms efficiency and does not allow for querying the relation through `predicate_property/2`.

Load Format using `qcompile/1`, the contents of the argument files are included in the `.qlf` file instead of the loading directive.

If this option is not present, it used the value of the prolog flag `qcompile` as default.

redefine_module(+*Action***)**

Defines what to do if a file is loaded that provides a module that is already loaded from another file. *Action* is one of `false` (default), which prints an error and refuses to load the file, or `true`, which uses `unload_file/1` on the old file and then proceeds loading the new file. Finally, there is `ask` that starts interaction with the user. Ask is only provided if user_input is associated with a terminal.

reexport(*Bool***)**

If `true` re-export the imported predicate. Used by `reexport/1` and `reexport/2`.

silent(*Bool***)**

If `true`, load the file without printing a message. The specified value is the default for all files loaded as a result of loading the specified files. This option writes the Prolog flag `verbose_load` with the negation of *Bool*.

stream(*Input***)**

This SWI-Prolog extension compiles the data from the stream *Input*. If this option is used, *Files* must be a single atom which is used to identify the source-location of the loaded clauses as well as remove all clauses if the data is re-consulted.

This option is added to allow compiling from non-file locations such as databases, the web, the *user* (see `consult/1`) or other servers. It can be combined with `format`(*qlf*) to load QLF data from a stream.

The `load_files/2` predicate can be hooked to load other data or data from other objects than files. See `prolog_load_file/2` for a description and `http_load` for an example.

consult(*:Files***)**

Read *File* as a Prolog source file. Calls to `consult/1` may be abbreviated by just typing a number of file names in a list. Examples:

```
?- consult(load).        % consult load or load.pl
?- [library(lists)].     % load library lists
?- [user].               % Type program on the terminal
```

The predicate `consult/1` is equivalent to load_files(Files, []), except for handling the special file `user`, which reads clauses from the terminal. See also the `stream`(*Input*) option of `load_files/2`.

ensure_loaded(*:File***)**

If the file is not already loaded, this is equivalent to `consult/1`. Otherwise, if the file defines a module, import all public predicates. Finally, if the file is already loaded, is not a module file and the context module is not the global user module, `ensure_loaded/1` will call `consult/1`.

With the semantics, we hope to get as close as possible to the clear semantics without the presence of a module system. Applications using modules should consider using `use_module/[1,2]`.

Equivalent to load_files(Files, [if(not_loaded)]).[5]

include(+*File***)**

Pretend the terms in *File* are in the source-file in which `:- include(File)` appears. The include construct is only honoured if it appears as a directive in a source-file. Normally *File* contains a sequence of directives.

require(+*ListOfNameAndArity***)**

Declare that this file/module requires the specified predicates to be defined "with their commonly accepted definition". This predicate originates from the Prolog portability layer for XPCE. It is intended to provide a portable mechanism for specifying that this module requires the specified predicates.

The implementation normally first verifies whether the predicate is already defined. If not, it will search the libraries and load the required library.

SWI-Prolog, having autoloading, does **not** load the library. Instead it creates a procedure header for the predicate if it does not exist. This will flag the predicate as 'undefined'. See also `check/0` and `autoload/0`.

encoding(+*Encoding***)**

This directive can appear anywhere in a source file to define how characters are encoded in the remainder of the file. It can be used in files that are encoded with a superset of US-ASCII, currently UTF-8 and ISO Latin-1. See also section 2.18.1.

make

Consult all source files that have been changed since they were consulted. It checks *all* loaded source files: files loaded into a compiled state using `pl -c ...` and files loaded using consult or one of its derivatives. The predicate `make/0` is called after `edit/1`, automatically reloading all modified files. If the user uses an external editor (in a separate window), `make/0` is normally used to update the program after editing. In addition, `make/0` updates the autoload indices (see section 2.13) and runs `list_undefined/0` from the `check` library to report on undefined predicates.

library_directory(?*Atom***)**

Dynamic predicate used to specify library directories. Default `./lib`, `~/lib/prolog` and the system's library (in this order) are defined. The user may add library directories using `assertz/1`, `asserta/1` or remove system defaults using `retract/1`. Deprecated. New code should use `file_search_path/2`.

file_search_path(+*Alias*, ?*Path***)**

Dynamic predicate used to specify 'path-aliases'. This feature is best described using an example. Given the definition

```
file_search_path(demo, '/usr/lib/prolog/demo').
```

the file specification `demo(myfile)` will be expanded to `/usr/lib/prolog/demo/myfile`. The second argument of `file_search_path/2` may be another alias.

---

[5] On older versions the condition used to be if(changed). Poor time management on some machines or due to copying often caused problems. The `make/0` predicate deals with updating the running system after changing the source code.

Below is the initial definition of the file search path. This path implies `swi(`⟨*Path*⟩`)` refers to a file in the SWI-Prolog home directory. The alias `foreign(`⟨*Path*⟩`)` is intended for storing shared libraries (`.so` or `.DLL` files). See also `load_foreign_library/[1,2]`.

```
user:file_search_path(library, X) :-
        library_directory(X).
user:file_search_path(swi, Home) :-
        current_prolog_flag(home, Home).
user:file_search_path(foreign, swi(ArchLib)) :-
        current_prolog_flag(arch, Arch),
        atom_concat('lib/', Arch, ArchLib).
user:file_search_path(foreign, swi(lib)).
user:file_search_path(path, Dir) :-
        getenv('PATH', Path),
        (   current_prolog_flag(windows, true)
        ->  atomic_list_concat(Dirs, (;), Path)
        ;   atomic_list_concat(Dirs, :, Path)
        ),
        member(Dir, Dirs).
```

The `file_search_path/2` expansion is used by all loading predicates as well as by `absolute_file_name/[2,3]`.

The Prolog flag `verbose_file_search` can be set to `true` to help debugging Prolog's search for files.

expand_file_search_path(+*Spec*, -*Path***)**
Unifies *Path* with all possible expansions of the file name specification *Spec*. See also `absolute_file_name/3`.

prolog_file_type(?*Extension*, ?*Type***)**
This dynamic multifile predicate defined in module `user` determines the extensions considered by `file_search_path/2`. *Extension* is the filename extension without the leading dot, *Type* denotes the type as used by the `file_type(`*Type*`)` option of `file_search_path/2`. Here is the initial definition of `prolog_file_type/2`:

```
user:prolog_file_type(pl,       prolog).
user:prolog_file_type(Ext,      prolog) :-
        current_prolog_flag(associate, Ext),
        Ext \== pl.
user:prolog_file_type(qlf,      qlf).
user:prolog_file_type(Ext,      executable) :-
        current_prolog_flag(shared_object_extension, Ext).
```

Users can add extensions used for Prolog source files to avoid conflicts (for example with `perl`) as well as to be compatible with another Prolog implementation. We suggest using `.pro` for avoiding conflicts with `perl`. Overriding the system definitions can stop the system from finding libraries.

source_file(*?File***)**

True if *File* is a loaded Prolog source file. *File* is the absolute and canonical path to the source-file.

source_file(*?Pred, ?File***)**

Is true if the predicate specified by *Pred* was loaded from file *File*, where *File* is an absolute path name (see `absolute_file_name/2`). Can be used with any instantiation pattern, but the database only maintains the source file for each predicate. See also `clause_property/2`.

source_file_property(*?File, ?Property***)**

Is true when *Property* is a property of the loaded file *File*. If *File* is non-var, it can be a file-specification that is valid for `load_files/2`. Defined properties are:

derived_from(*Original, OriginalModified***)**

File was generated from the file *Original*, which was last-modified at *OriginalModified* at the time it was loaded. This property is available if *File* was loaded using the `derived_from(`*Original*`)` option to `load_files/2`.

includes(*IncludedFile, IncludedFileModified***)**

File used `include/1` to include *IncludedFile*. The last-modified time of *IncludedFile* was *IncludedFileModified* at the time it was included.

load_context(*Module, Location***)**

Module is the module into which the file was loaded. If *File* is a module, this is the module into which the exports are imported. Otherwise it is the module into which the clauses of the non-module file are loaded. *Location* describes the file-location from which the file was loaded. It is either a term ⟨*file*⟩:⟨*line*⟩ or the atom `user` if the file was loaded from the terminal or another unknown source.

modified(*Stamp***)**

File-modification time when *File* was loaded. This is used by `make/0` to find files whose modification time is different from when it was loaded.

module(*Module***)**

File is a module-file that declares the modules *Module*.

unload_file(*+File***)**

Remove all clauses loaded from *File*. If *File* loaded a module, clear the module's export-list and disassociates it from the file. *File* is a canonical file-name or a file-indicator that is valid for `load_files/2`.

This predicare shall be used with care. The multi-threaded nature of SWI-Prolog makes removing static code unsafe. Attempts to do this should be reserved to development or situations where the application can guarantee that none of the clauses associated to *File* are active.

prolog_load_context(*?Key, ?Value***)**

Obtain context information during compilation. This predicate can be used from directives appearing in a source file to get information about the file being loaded. See also `source_location/2` and `if/1`. The following keys are defined:

Key	**Description**
`module`	Module into which file is loaded
`source`	File loaded. Returns the original Prolog file when loading a `.qlf` file. Compatible with SICStus Prolog.
`file`	Currently equivalent to `source`. In future versions it may report a different values for files being loaded using `include/1`.
`stream`	Stream identifier (see `current_input/1`)
`directory`	Directory in which `source` lives.
`dialect`	Compatibility mode. See `expects_dialect/1`.
`term_position`	Position of last term read. Term of the form `'$stream_position'(0,`⟨*Line*⟩`,0,0,0)`. See also `stream_position_data/3`.
`script`	Boolean that indicates whether the file is loaded as a script file (see `-s`).

The `directory` is commonly used add rules to `file_search_path/2`, setting up a search-path for finding files with `absolute_file_name/3`. E.g.,

```
:- dynamic user:file_search_path/2.
:- multifile user:file_search_path/2.

:- prolog_load_context(directory, Dir),
   asserta(user:file_search_path(my_program_home, Dir)).


        ...
        absolute_file_name(my_program_home('README.TXT'), ReadMe,
                           [ access(read) ]),
        ...
```

source_location(-*File*, -*Line***)**

If the last term has been read from a physical file (i.e., not from the file `user` or a string), unify *File* with an absolute path to the file and *Line* with the line-number in the file. New code should use `prolog_load_context/2`.

at_halt(:*Goal***)**

Register *Goal* to be run from `PL_cleanup()`, which is called when the system halts. The hooks are run in the reverse order they were registered (FIFO). Success or failure executing a hook is ignored. If the hook raises an exception this is printed using `print_message/2`. An attempt to call `halt/[0,1]` from a hook is ignored.

:- initialization(:*Goal***)** *[ISO]*

Call *Goal* *after* loading the source-file in which this directive appears has been completed. In addition, *Goal* is executed if a saved-state created using `qsave_program/1` is restored.

The ISO standard only allows for using `:- Term` if *Term* is a *directive*. This means that arbitrary goals can only be called from a directive by means of the `initialization/1` directive. SWI-Prolog does not enforce this rule.

The `initialization/1` directive must be used to do program initialization in saved-states (see `qsave_program/1`). A saved state contains the predicates, Prolog flags and operators present at the moment the state was created. Other resources (records, foreign resources, etc.) must be recreated using `initialization/1` directives or from the entry-goal of the saved-state.

Upto SWI-Prolog 5.7.11, *Goal* was executed immediately rather than after loading the program-text in which the directive appears as dictated by the ISO standard. In many cases the exact moment of execution is irrelevant, but there are exceptions. For example, `load_foreign_library/1` must be executed immediately to make the loaded foreign predicates available for exporting. SWI-Prolog now provides the directive `use_foreign_library/1` to ensure immediate loading as well as loading after restoring a saved state. If the system encounters a directive `:- initialization(load_foreign_library(...))`, it will load the foreign library immediately and issue a warning to update your code. This behaviour can be extended by providing clauses for the multifile hook predicate `prolog:initialize_now`(*Term, Advice*), where *Advice* is an atom that gives advice how to resolve the compatibility issue.

initialization(*:Goal, +When***)**

Similar to `initialization/1`, but allows for specifying when *Goal* is executed while loading the program-text:

now

Execute *Goal* immediately.

after_load

Execute *Goal* after loading program-text. This is the same as `initialization/1`.

restore

Do not execute *Goal* while loading the program, but *only* when restoring a state.

compiling

True if the system is compiling source files with the `-c` option or `qcompile/1` into an intermediate code file. Can be used to perform conditional code optimisations in `term_expansion/2` (see also the `-O` option) or to omit execution of directives during compilation.

expansion/2 and expand

4.3.1 Conditional compilation and program transformation

ISO Prolog defines no way for program transformations such as macro expansion or conditional compilation. Expansion through `term_expansion/2` and `expand_term/2` can be seen as part of the de-facto standard. This mechanism can do arbitrary translation between valid Prolog terms read from the source file to Prolog terms handed to the compiler. As `term_expansion/2` can return a list, the transformation does not need to be term-to-term.

Various Prolog dialects provide the analogousbe term-to-0 g2mrGerm read during consulting are given to this predicate. If the predicate succeeds Prolog will assert *Term2* in the database rather then the read term (*Term1*). *Term2* may be a term of the form '?- *Goal*' or ':- *Goal*'. *Goal* is then treated as a directive. If *Term2* is a list all terms of the list are stored in the database or called (for directives). If *Term2* is of the form below, the system will assert *Clause* and record the indicated source-location with it.

`'$source_location'(`⟨*File*⟩`,` ⟨*Line*⟩`):`⟨*Clause*⟩

When compiling a module (see chapter 5 and the directive `module/2`), `expand_term/2` will first try `term_expansion/2` in the module being compiled to allow for term-expansion rules that are local to a module. If there is no local definition, or the local definition fails to translate the term, `expand_term/2` will try `term_expansion/2` in module `user`. For compatibility with SICStus and Quintus Prolog, this feature should not be used. See also `expand_term/2`, `goal_expansion/2` and `expand_goal/2`.

expand_term(+*Term1, -Term2***)**

This predicate is normally called by the compiler on terms read from the input to perform preprocessing. It consists of three steps, where each step processes the output of the previous step.

1. Test conditional compilation directives and translate all input to `[]` if we are in a 'false-branch' of the conditional compilation. See section 4.3.1.
2. Call `term_expansion/2`. This predicate is first tried in the module that is being compiled and then in the module `user`.
3. Call DGC expansion (`dcg_translate_rule/2`)
4. Call `expand_goal/2` on each body-term that appears in the output of the previous steps.

goal_expansion(+*Goal1, -Goal2***)**

Like `term_expansion/2`, `goal_expansion/2` provides for macro-expansion of Prolog source-code. Between `expand_term/2` and the actual compilation, the body of clauses analysed and the goals are handed to `expand_goal/2`, which uses the `goal_expansion/2` hook to do user-defined expansion.

The predicate `goal_expansion/2` is first called in the module that is being compiled, and then on the `user` module. If *Goal* is of the form *Module*:*Goal* where *Module* is instantiated, `goal_expansion/2` is called on *Goal* using rules from module *Module* followed by `user`.

Only goals appearing in the body of clauses when reading a source-file are expanded using this mechanism, and only if they appear literally in the clause, or as an argument to a defined meta-predicate that is annotated using '0' (see `meta_predicate/1`). Other cases need a real predicate definition.

expand_goal(+*Goal1, -Goal2***)**

This predicate is normally called by the compiler to perform preprocessing using `goal_expansion/2`. The predicate computes a fixed-point by applying transformations until there are no more changes. If optimisation is enabled (see `-O` and `optimise`), `expand_goal/2` simplifies the result by removing unneeded calls to `true/0` and `fail/0` as well as unreachable branches.

compile_aux_clauses(+*Clauses***)**

Compile clauses on behalf of `goal_expansion/2`. This predicate compiled the argument clauses into static predicates, associating the predicates with the current file but avoid changing the notion of current predicate and therefore discontiguous warnings.

dcg_translate_rule(+*In*, -*Out***)**

This predicate performs the translation of a term `Head-->Body` into a normal Prolog clause. Normally this functionality should be accessed using `expand_term/2`.

preprocessor(-*Old*, +*New***)**

Read the input file via an external process that acts as preprocessor. A preprocessor is specified as an atom. The first occurrence of the string '`%f`' is replaced by the name of the file to be loaded. The standard output of resulting command is loaded. To use the Unix C preprocessor one should define:

```
?- preprocessor(Old, '/lib/cpp -C -P %f'), consult(...).

Old = none
```

Using `cpp` for Prolog preprocessing is not ideal as the tokenization rules for comment and quoted strings differ between C and Prolog. Another problem is availability and compatibility with regard to option processing of `cpp`.

Conditional compilation

Conditional compilation builds on the same principle as `term_expansion/2`, `goal_expansion/2` and the expansion of grammar rules to compile sections of the source-code conditionally. One of the reasons for introducing conditional compilation is to simplify writing portable code. See section C for more information. Here is a simple example:

```
:- if(\+source_exports(library(lists), suffix/2)).

suffix(Suffix, List) :-
        append(_, Suffix, List).

:- endif.
```

Note that these directives can only appear as separate terms in the input. Typical usage scenarios include:

- Load different libraries on different dialects
- Define a predicate if it is missing as a system predicate
- Realise totally different implementations for a particular part of the code due to different capabilities.
- Realise different configuration options for your software.

:- if(*:Goal*)

Compile subsequent code only if *Goal* succeeds. For enhanced portability, *Goal* is processed by `expand_goal/2` before execution. If an error occurs, the error is printed and processing proceeds as if *Goal* has failed.

:- elif(*:Goal*)

Equivalent to :- else. :-if(Goal) ... :- endif. In a sequence as below, the section below the first matching elif is processed, If no test succeeds the else branch is processed.

```
:- if(test1).
section_1.
:- elif(test2).
section_2.
:- elif(test3).
section_3.
:- else.
section_else.
:- endif.
```

:- else

Start ‘else’ branch.

:- endif

End of conditional compilation.

4.3.2 Loading files, active code and threads

Traditionally, Prolog environments allow for reloading files holding currently active code. In particular, the following sequence is valid use of the development environment:

- Trace a goal
- Find unexpected behaviour of a predicate
- Enter a *break* using the **b** command
- Fix the sources and reload them using `make/0`
- Exit the break, *retry* using the **r** command

Goals running during the reload keep running on the old definition, while new goals use the reloaded definition, which is why the *retry* must be used *after* the reload. This implies that clauses of predicates that are active during the reload cannot be reclaimed. Normally a small amount of dead clauses should not be an issue during development. Such clauses can be reclaimed with `garbage_collect_clauses/0`.

garbage_collect_clauses

Cleanup all *dirty* predicates, where dirty predicates are defined to be predicates that have both old and new definitions due to reloading a source file while the predicate was active. Of course, predicates that are active using `garbage_collect_clauses/0` cannot be reclaimed and remain *dirty*. Predicate are -like atoms- shared resources and therefore all threads are suspended during the execution of this predicate.

Compilation of mutual dependent code

Large programs are generally split into multiple files. If file *A* accesses predicates from file *B* which accesses predicates from file *A*, we consider this a mutual or circular dependency. If traditional load predicates (e.g., `consult/1`) are used to include file *B* from *A* and *A* from *B*, loading either file results in a loop. This is because `consult/1` is mapped to `load_files/2` using the option `if(true)`(.) Such programs are typically loaded using a *load-file* that consults all required (non-module) files. If modules are used, the dependencies are made explicit using `use_module/1` statements. The `use_module/1` predicate however maps to `load_files/2` with the option `if(not_loaded)`(.) A `use_module/1` on an already loaded file merely makes the public predicates of the used module available.

Summarizing, mutual dependency of source-files is fully supported with no precautions when using modules. Modules can use each other in an arbitrary dependency graph. When using `consult/1`, predicate dependencies between loaded files can still be arbitrary, but the consult relations between files must be a proper tree.

Compilation with multiple threads

This section discusses compiling files for the first time. For re-loading see section 4.3.2.

In older versions, compilation was thread-safe due to a global *lock* in `load_files/2` and the code dealing with *autoloading* (see section 2.13). Besides unnecessary stalling when multiple threads trap unrelated undefined predicates, this easily leads to deadlocks, notably if threads are started from an `initialization/1` directive.[6]

Starting with version 5.11.27, the autoloader is no longer locked and multiple threads can compile files concurrently. This requires special precautions only if multiple threads wish to load the same file at the same time. Therefore, `load_files/2` checks atomically whether some other thread is already loading the file. If not, it starts loading the file. If another thread is already loading the file, the thread blocks until the other thread finishes loading the file. After waiting, and if the file is a module file, it will make the public predicates available.

Note that this schema does not prevent deadlocks under all situations. Consider two mutually dependent (see section 4.3.2) module files *A* and *B*, where thread 1 starts loading *A* and thread 2 starts loading *B* at the same time. Both threads will deadlock when trying to load the used module.

The current implementation does not detect such cases and the involved threads will freeze. This problem can be avoided if a mutually dependend collection of files is always loaded from the same start-file.

Reloading running code

This section discusses *not re*-loading of code. Initial loading of code is discussed in section 4.3.2.

As of version 5.5.30, there is basic thread-safety for reloading source files while other threads are executing code defined in these source files. Reloading a file freezes all threads after marking the active predicates originating from the file being reloaded. The threads are resumed after the file has been loaded. In addition, after completing loading the outermost file, the system runs `garbage_collect_clauses/0`.

---

[6] Although such goals are started after loading the file in which they appear, the calling thread is still likely to hold the 'load' lock because it is compiling the file from which the file holding the directive is loaded.

What does that mean? Unfortunately it does *not* mean we can 'hot-swap' modules. Consider the case where thread A is executing the recursive predicate P. We 'fix' P and reload. The already running goals for P continue to run the old definition, but new recursive calls will use the new definition! Many similar cases can be constructed with dependent predicates.

It provides some basic security for reloading files in multi-threaded applications during development. In the above scenarios the system does not crash uncontrolled, but behaves like any broken program: it may return the wrong bindings, wrong truth value or raise an exception.

Future versions may have an 'update now' facility. Such a facility can be implemented on top of the *logical update view*. It would allow threads to do a controlled update between processing independent jobs.

4.3.3 Quick load files

SWI-Prolog supports compilation of individual or multiple Prolog source files into 'Quick Load Files'. A 'Quick Load Files' (`.qlf` file) stores the contents of the file in a precompiled format.

These files load considerably faster than source files and are normally more compact. They are machine independent and may thus be loaded on any implementation of SWI-Prolog. Note however that clauses are stored as virtual machine instructions. Changes to the compiler will generally make old compiled files unusable.

Quick Load Files are created using `qcompile/1`. They are loaded using `consult/1` or one of the other file-loading predicates described in section 4.3. If consult is given the explicit `.pl` file, it will load the Prolog source. When given the `.qlf` file, it will load the file. When no extension is specified, it will load the `.qlf` file when present and the `.pl` file otherwise.

qcompile(*:File***)**

Takes a file specification as `consult/1`, etc. and, in addition to the normal compilation, creates a *Quick Load File* from *File*. The file-extension of this file is `.qlf`. The base name of the Quick Load File is the same as the input file.

If the file contains '`:- consult(`*+File*`)`', '`:- [`*+File*`]`' or `:- load_files(`*+File*`, [qcompile(part), ...])` statements, the referred files are compiled into the same `.qlf` file. Other directives will be stored in the `.qlf` file and executed in the same fashion as when loading the `.pl` file.

For `term_expansion/2`, the same rules as described in section 2.10 apply.

Conditional execution or optimisation may test the predicate `compiling/0`.

Source references (`source_file/2`) in the Quick Load File refer to the Prolog source file from which the compiled code originates.

qcompile(*:File, +Options***)**

As `qcompile/1`, but processes additional options as defined by `load_files/2`.[7]

4.4 Editor Interface

SWI-Prolog offers an extensible interface which allows the user to edit objects of the program: predicates, modules, files, etc. The editor interface is implemented by `edit/1` and consists of three parts: *locating*, *selecting* and *starting the editor*. Any of these parts may be customized. See section 4.4.1.

---

[7] BUG: Option processing is currently incomplete.

The built-in edit specifications for `edit/1` (see prolog_edit:locate/3) are described in the table below.

Fully specified objects	
⟨*Module*⟩:⟨*Name*⟩/⟨*Arity*⟩	Refers a predicate
module(⟨*Module*⟩)	Refers to a module
file(⟨*Path*⟩)	Refers to a file
source_file(⟨*Path*⟩)	Refers to a loaded source-file
Ambiguous specifications	
⟨*Name*⟩/⟨*Arity*⟩	Refers this predicate in any module
⟨*Name*⟩	Refers to (1) named predicate in any module with any arity, (2) a (source) file or (3) a module.

edit(+*Specification***)**

First exploits prolog_edit:locate/3 to translate *Specification* into a list of *Locations*. If there is more than one 'hit', the user is asked to select from the locations found. Finally, prolog_edit:edit_source/1 is used to invoke the user's preferred editor. Typically, `edit/1` can be handed the name of a predicate, module, basename of a file, XPCE class, XPCE method, etc.

edit

Edit the 'default' file using `edit/1`. The default file is the file loaded with the command-line option `-s` or, in windows, the file loaded by double-clicking from the Windows shell.

4.4.1 Customizing the editor interface

The predicates described in this section are *hooks* can that be defined to disambiguate specifications given to `edit/1`, find the related source and open an editor at the given source location.

prolog_edit:locate(+*Spec, -FullSpec, -Location***)**

Where *Spec* is the specification provided through `edit/1`. This multifile predicate is used to enumerate locations at with an object satisfying the given *Spec* can be found. *FullSpec* is unified with the complete specification for the object. This distinction is used to allow for ambiguous specifications. For example, if *Spec* is an atom, which appears as the base-name of a loaded file and as the name of a predicate, *FullSpec* will be bound to `file`(*Path*) or *Name*/*Arity*.

Location is a list of attributes of the location. Normally, this list will contain the term `file`(*File*) and —if available— the term `line`(*Line*).

prolog_edit:locate(+*Spec, -Location***)**

Same as prolog_edit:locate/3, but only deals with fully-specified objects.

prolog_edit:edit_source(+*Location***)**

Start editor on *Location*. See prolog_edit:locate/3 for the format of a location term. This multi-file predicate is normally not defined. If it succeeds, `edit/1` assumes the editor is started.

If it fails, `edit/1` uses its internal defaults, which are defined by the Prolog flag `editor` and/or the environment variable `EDITOR`. The following rules apply. If the Prolog flag `editor` is of the format `$`⟨*name*⟩, the editor is determined by the environment variable ⟨*name*⟩.

Else, if this flag is `pce_emacs` or `built_in` *and* XPCE is loaded or can be loaded, the built-in Emacs clone is used. Else, if the environment `EDITOR` is set, this editor is used. Finally, `vi` is used as default on Unix systems and `notepad` on Windows.

See the default user preferences file `dotfiles/dotplrc` for examples.

prolog_edit:edit_command(+*Editor*, -*Command***)**

Determines how *Editor* is to be invoked using `shell/1`. *Editor* is the determined editor (see `edit_source/1`), without the full path specification, and without possible (exe) extension. *Command* is an atom describing the command. The following %-sequences are relaced in *Command* before the result is handed to `shell/1`:

%e	Replaced by the (OS) command name of the editor
%f	Replaced by the (OS) full path-name of the file
%d	Replaced by the line-number

If the editor can deal with starting at a specified line, two clauses should be provided. The first pattern invokes the editor with a line-number, while the second is used if the line number is unknown.

The default contains definitions for `vi`, `emacs`, `emacsclient`, `vim`, `notepad`* and `wordpad`*. Starred editors do not provide starting at a given line number.

Please contribute your specifications to `bugs@swi-prolog.org`.

prolog_edit:load

Normally not-defined multifile predicate. This predicate may be defined to provide loading hooks for user-extensions to the edit module. For example, XPCE provides the code below to load `swi_edit`, containing definitions to locate classes and methods as well as to bind this package to the PceEmacs built-in editor.

```
:- multifile prolog_edit:load/0.

prolog_edit:load :-
        ensure_loaded(library(swi_edit)).
```

4.5 List the program, predicates or clauses

listing(:*Pred***)**

List predicates specified by *Pred*. *Pred* may be a predicate name (atom), which lists all predicates with this name, regardless or their arity. It can also be a predicate indicator (⟨*name*⟩/⟨*arity*⟩ or ⟨*name*⟩//⟨*arity*⟩), possibly qualified with a module. For example: `?- listing(lists:member/2).`.

A listing is produced by enumerating the clauses of the predicate using `clause/2` and printing each clause using `portray_clause/1`. This implies that the variable names are generated (*A*, *B*, ...) and the layout is defined by rules in `portray_clause/1`.

listing
List all predicates from the calling module using `listing/1`. E.g., `?- listing.` lists clauses in the default `user` module and `?- lists:listing.` lists the clauses in the module `lists`.

portray_clause(+*Clause***)**
Pretty print a clause. A clause should be specified as a term '⟨*Head*⟩ `:-` ⟨*Body*⟩'. Facts are represented as '⟨*Head*⟩ `:- true`' or simply ⟨*Head*⟩. Variables in the clause are written as A, B, ... Singleton variables are written as _. See also `portray_clause/2`.

portray_clause(+*Stream*, +*Clause***)**
Pretty print a clause to *Stream*. See `portray_clause/1` for details.

4.6 Verify Type of a Term

Type tests are semi deterministic predicates that succeed if the argument satisfies the requested type. Type-test predicates have no error condition and do not instantiate their argument. See also library `error`.

var(@*Term***)** *[ISO]*
True if *Term* currently is a free variable.

nonvar(@*Term***)** *[ISO]*
True if *Term* currently is not a free variable.

integer(@*Term***)** *[ISO]*
True if *Term* is bound to an integer.

float(@*Term***)** *[ISO]*
True if *Term* is bound to a floating point number.

rational(@*Term***)**
True if *Term* is bound to a rational number. Rational numbers include integers.

rational(@*Term*, -*Numerator*, -*Denominator***)**
True if *Term* is a rational number with given *Numerator* and *Denominator*. The *Numerator* and *Denominator* are in canonical form, which means *Denominator* is a positive integer and there are no common divisors between *Numerator* and *Denominator*.

number(@*Term***)** *[ISO]*
True if *Term* is bound to an integer or floating point number.[8]

atom(@*Term***)** *[ISO]*
True if *Term* is bound to an atom.

blob(@*Term*, ?*Type***)**
True if *Term* is a *blob* of type *Type*. See section 9.4.7.

---

[8] As rational numbers are not atomic in the current implementation and we do not want to break the rule that `number/1` implies `atomic/1`, `number/1` fails on rational numbers. This will change if rational numbers become atomic.

string(@*Term***)**

True if *Term* is bound to a string. Note that string here refers to the built-in atomic type string as described in section 4.23, Text in double quotes such as `"hello"` creates a *list* of *character codes*. We illustrate the issues in the example queries below.

```
?- write("hello").
[104, 101, 108, 108, 111]
?- string("hello").
No
?- is_list("hello").
Yes
```

atomic(@*Term***)** *[ISO]*

True if *Term* is bound to an atom, string, integer or floating point number. Note that string refers to the built-in type. See `string/1`. Strings in the classical Prolog sense are lists and therefore compound.

compound(@*Term***)** *[ISO]*

True if *Term* is bound to a compound term. See also `functor/3` and =../2.

callable(@*Term***)** *[ISO]*

True if *Term* is bound to an atom or a compound term, so it can be handed without type-error to `call/1`, `functor/3` and =../2.

ground(@*Term***)** *[ISO]*

True if *Term* holds no free variables.

cyclic_term(@*Term***)**

True if *Term* contains cycles, i.e. is an infinite term. See also `acyclic_term/1` and section 2.16.[9]

acyclic_term(@*Term***)** *[ISO]*

True if *Term* does not contain cycles, i.e. can be processed recursively in finite time. See also `cyclic_term/1` and section 2.16.

4.7 Comparison and Unification of Terms

Although unification is mostly done implicitly while matching the head of a predicate, it is also provided by the predicate =/2.

?Term1 **=** *?Term2* *[ISO]*

Unify *Term1* with *Term2*. True if the unification succeeds. For behaviour on cyclic terms see the Prolog flag `occurs_check`. It acts as if defined by the following fact.

```
=(Term, Term).
```

---

[9] The predicates `cyclic_term/1` and `acyclic_term/1` are compatible with SICStus Prolog. Some Prolog systems supporting cyclic terms use `is_cyclic/1`.

@Term1 **\=** *@Term2* *[ISO]*

Equivalent to `\+Term1 = Term2`. See also `dif/2`.

4.7.1 Standard Order of Terms

Comparison and unification of arbitrary terms. Terms are ordered in the so called "standard order". This order is defined as follows:

1. *Variables* < *Numbers* < *Atoms* < *Strings* < *Compound Terms*[10]
2. Variables are sorted by address. Attaching attributes (see section 6.1) does not affect the ordering.
3. *Atoms* are compared alphabetically.
4. *Strings* are compared alphabetically.
5. *Numbers* are compared by value. Mixed integer/float are compared as floats. If the comparison is equal, the float is considered the smaller value. If the Prolog flag `iso` is defined, all floating point numbers precede all integers.
6. *Compound* terms are first checked on their arity, then on their functor-name (alphabetically) and finally recursively on their arguments, leftmost argument first.

@Term1 **==** *@Term2* *[ISO]*

True if *Term1* is equivalent to *Term2*. A variable is only identical to a sharing variable.

@Term1 **\==** *@Term2* *[ISO]*

Equivalent to `\+Term1 == Term2`.

@Term1 **@<** *@Term2* *[ISO]*

True if *Term1* is before *Term2* in the standard order of terms.

@Term1 **@=<** *@Term2* *[ISO]*

True if both terms are equal (`==/2`) or *Term1* is before *Term2* in the standard order of terms.

@Term1 **@>** *@Term2* *[ISO]*

True if *Term1* is after *Term2* in the standard order of terms.

@Term1 **@>=** *@Term2* *[ISO]*

True if both terms are equal (`==/2`) or *Term1* is after *Term2* in the standard order of terms.

compare(*?Order, @Term1, @Term2***)** *[ISO]*

Determine or test the *Order* between two terms in the standard order of terms. *Order* is one of <, > or =, with the obvious meaning.

---

[10] Strings might be considered atoms in future versions. See also section 4.23

4.7.2 Special unification and comparison predicates

This section describes special purpose variations on Prolog unification. The predicate `unify_with_occurs_check/2` provides sound unification and is part of the ISO standard. The predicate `subsumes_term/2` defines 'one-sided-unification' and is part of the ISO proposal established in Edinburgh (2010). Finally, `unifiable/3` is a 'what-if' version of unification that is often used as a building block in constraint reasoners.

unify_with_occurs_check(+*Term1*, +*Term2***)** *[ISO]*

As `=/2`, but using *sound-unification*. That is, a variable only unifies to a term if this term does not contain the variable itself. To illustrate this, consider the two queries below.

```
1 ?- A = f(A).
A = f(A).
2 ?- unify_with_occurs_check(A, f(A)).
false.
```

The first creates a *cyclic-term*, also called a *rational tree*. The second executes logically sound unification and thus fails. Note that the behaviour of unification through `=/2` as well as implicit unification in the head can be changed using the Prolog flag `occurs_check`.

The SWI-Prolog implementation of `unify_with_occurs_check/2` is cycle-safe and only guards against *creating* cycles, not against cycles that may already be present in one of the arguments. This is illustrated in the following two queries:

```
?- X = f(X), Y = X, unify_with_occurs_check(X, Y).
X = Y, Y = f(Y).
?- X = f(X), Y = f(Y), unify_with_occurs_check(X, Y).
X = Y, Y = f(Y).
```

Some other Prolog systems interpret `unify_with_occurs_check/2` as if defined by the clause below, causing failure on the above two queries. Direct use of `acyclic_term/1` is portable and more appropriate for such applications.

```
unify_with_occurs_check(X,X) :- acyclic_term(X).
```

+*Term1* **=@=** +*Term2*

True if *Term1* is a *variant* of (or *structurally equivalent* to) *Term2*. Testing for a variant is weaker than equivalence (`==/2`), but stronger than unification (`=/2`). Two terms A and B are variants iff there exists a renaming of the variables in A that makes A equivalent (==) to B and visa-versa.[11] Examples:

---

[11] Row 7 and 8 of this table may come as a surprise, but row 8 is satisfied by (left-to-right) $A \rightarrow C$, $B \rightarrow A$ and (right-to-left) $C \rightarrow A$, $A \rightarrow B$. If the same variable appears in different locations in the left and right term, the variant-relation can be broken by consistent binding of both terms. E.g., after binding the first argument in row 8 to a value both terms are no longer variant.

1	`a =@= A`	false
2	`A =@= B`	true
3	`x(A,A) =@= x(B,C)`	false
4	`x(A,A) =@= x(B,B)`	true
5	`x(A,A) =@= x(A,B)`	false
6	`x(A,B) =@= x(C,D)`	true
7	`x(A,B) =@= x(B,A)`	true
8	`x(A,B) =@= x(C,A)`	true

A term is always a variant of a copy of itself. Term copying takes place in e.g., `copy_term/2`, `findall/3` or proving a clause added with `asserta/1`. In the pure Prolog world (i.e., without attributed variables), `=@=/2` behaves as if defined below. With attributed variables, variant of the attributes is tested rather than trying to satisfy the constraints.

```
A =@= B :-
        copy_term(A, Ac),
        copy_term(B, Bc),
        numbervars(Ac, 0, N),
        numbervars(Bc, 0, N),
        Ac == Bc.
```

The SWI-Prolog implementation is cycle-safe and can deal with variables that are shared between the left and right argument. Its performance is comparable to `==/2`, both on success and (early) failure. [12]

This predicate is known by the name `variant/2` in some other Prolog systems. Be aware of possible differences in semantics if the arguments contain attributed variables or share variables.[13]

+*Term1* **\=@=** +*Term2*

Equivalent to `'\+Term1 =@= Term2'`. See `=@=/2` for details.

subsumes_term(@*Generic*, @*Specific***)** *[ISO]*

True if *Generic* can be made equivalent to *Specific* by only binding variables in *Generic*. The current implementation performs the unification and ensures that the variable set of *Specific* is not changed by the unification. On success, the bindings are undone.[14]

term_subsumer(+*Special1*, +*Special2*, -*General***)**

General is the most specific term that is a generalisation of *Special1* and *Special2*. The implementation can handle cyclic terms.

unifiable(@*X*, @*Y*, -*Unifier***)**

If *X* and *Y* can unify, unify *Unifier* with a list of *Var* = *Value*, representing the bindings required

---

[12] The current implementation is contributed by Kuniaki Mukai.

[13] In many systems variant is implemented using two calls to `subsumes_term/2`.

[14] This predicate is often named `subsumes_chk/2` in older Prolog dialects. The current name was established in the ISO WG17 meeting in Edinburgh (2010). The `chk` postfix was considered to refer to determinism as in e.g., `memberchk/2`.

to make *X* and *Y* equivalent.[15] This predicate can handle cyclic terms. Attributed variables are handles as normal variables. Associated hooks are *not* executed.

?=(*@Term1, @Term2***)**
Succeeds, if the syntactic equality of *Term1* and *Term2* can be decided safely, i.e. if the result of `Term1 == Term2` will not change due to further instantiation of either term. It behaves as if defined by `?=(X,Y) :- \+ unifiable(X,Y,[_|_]).`

4.8 Control Predicates

The predicates of this section implement control structures. Normally the constructs in this section, except for `repeat/0`, are translated by the compiler. Please note that complex goals passed as arguments to meta-predicates such as `findall/3` below cause the goal to be compiled to a temporary location before execution. It is faster to define a sub-predicate (i.e. `one_character_atom/1` in the example below) and make a call to this simple predicate.

```
one_character_atoms(As) :-
        findall(A, (current_atom(A), atom_length(A, 1)), As).
```

fail *[ISO]*
Always fail. The predicate `fail/0` is translated into a single virtual machine instruction.

false *[ISO]*
Same as fail, but the name has a more declarative connotation.

true *[ISO]*
Always succeed. The predicate `true/0` is translated into a single virtual machine instruction.

repeat *[ISO]*
Always succeed, provide an infinite number of choice points.

! *[ISO]*
Cut. Discard choice points of parent frame and frames created after the parent frame. As of SWI-Prolog 3.3, the semantics of the cut are compliant with the ISO standard. This implies that the cut is transparent to `;/2`, `->/2` and `*->/2`. Cuts appearing in the *condition* part of `->/2` and `*->/2` as well as in `\+/1` are local to the condition.[16]

```
t1 :- (a, !, fail ; b).          % cuts a/0 and t1/0
t2 :- (a -> b, !  ; c).          % cuts b/0 and t2/0
t3 :- call((a, !, fail ; b)).    % cuts a/0
t4 :- \+(a, !, fail ; b).        % cuts a/0
```

:Goal1 `,` *:Goal2* *[ISO]*
Conjunction. True if both 'Goal1' and 'Goal2' can be proved. It is defined as (this definition does not lead to a loop as the second comma is handled by the compiler):

---

[15] This predicate was introduced for the implementation of `dif/2` and `when/2` after discussion with Tom Schrijvers and Bart Demoen. None of us is really happy with the name and therefore suggestions for a new name are welcome.

[16] Up to version 4.0.6, the sequence X=!, X acted as a true cut. This feature has been deleted for ISO compliance.

```
Goal1, Goal2 :- Goal1, Goal2.
```

:Goal1 **;** *:Goal2* *[ISO]*

The 'or' predicate is defined as:

```
Goal1 ; _Goal2 :- Goal1.
_Goal1 ; Goal2 :- Goal2.
```

:Goal1 **|** *:Goal2*

Equivalent to `;/2`. Retained for compatibility only. New code should use `;/2`.

:Condition **->** *:Action* *[ISO]*

If-then and If-Then-Else. The `->/2` construct commits to the choices made at its left-hand side, destroying choice-points created inside the clause (by `;/2`), or by goals called by this clause. Unlike `!/0`, the choice-point of the predicate as a whole (due to multiple clauses) is **not** destroyed. The combination `;/2` and `->/2` acts as if defines by:

```
If -> Then; _Else :- If, !, Then.
If -> _Then; Else :- !, Else.
If -> Then :- If, !, Then.
```

Please note that (If -> Then) acts as (If -> Then ; **fail**), making the construct *fail* if the condition fails. This unusual semantics is part of the ISO and all de-facto Prolog standards.

:Condition ***->** *:Action ; :Else*

This construct implements the so-called 'soft-cut'. The control is defined as follows: If *Condition* succeeds at least once, the semantics is the same as (*Condition*, *Action*). If *Condition* does not succeed, the semantics is that of (\+ *Condition*, *Else*). In other words, If *Condition* succeeds at least once, simply behave as the conjunction of *Condition* and *Action*, otherwise execute *Else*.

The construct *A* `*->` *B*, i.e. without an *Else* branch, is translated as the normal conjunction *A*, *B*.[17]

\+ *:Goal* *[ISO]*

True if 'Goal' cannot be proven (mnemonic: + refers to *provable* and the backslash (\) is normally used to indicate negation in Prolog).

4.9 Meta-Call Predicates

Meta-call predicates are used to call terms constructed at run time. The basic meta-call mechanism offered by SWI-Prolog is to use variables as a subclause (which should of course be bound to a valid goal at runtime). A meta-call is slower than a normal call as it involves actually searching the database at runtime for the predicate, while for normal calls this search is done at compile time.

---

[17] BUG: The decompiler implemented by `clause/2` returns this construct as a normal conjunction too.

call(*:Goal***)** *[ISO]*

Invoke *Goal* as a goal. Note that clauses may have variables as subclauses, which is identical to `call/1`.

call(*:Goal, +ExtraArg1, . . .***)** *[ISO]*

Append *ExtraArg1, ExtraArg2, . . .* to the argument list of *Goal* and call the result. For example, `call(plus(1), 2, X)` will call `plus(1, 2, X)`, binding *X* to 3.

The call/[2..] construct is handled by the compiler. The predicates `call/[2-8]` are defined as real (meta-)predicates and are available to inspection through `current_predicate/1`, `predicate_property/2`, etc.[18] Higher arities are handled by the compiler and runtime system, but the predicates are not accessible for inspection.[19]

apply(*:Goal, +List***)**

Append the members of *List* to the arguments of *Goal* and call the resulting term. For example: `apply(plus(1), [2, X])` calls `plus(1, 2, X)`. New code should use call/[2..] if the length of *List* is fixed.

not(*:Goal***)**

True if *Goal* cannot be proven. Retained for compatibility only. New code should use `\+/1`.

once(*:Goal***)** *[ISO]*

Defined as:

```
once(Goal) :-
        Goal, !.
```

`once/1` can in many cases be replaced with `->/2`. The only difference is how the cut behaves (see !/0). The following two clauses are identical:

```
1) a :- once((b, c)), d.
2) a :- b, c -> d.
```

ignore(*:Goal***)**

Calls *Goal* as `once/1`, but succeeds, regardless of whether *Goal* succeeded or not. Defined as:

```
ignore(Goal) :-
        Goal, !.
ignore(_).
```

call_with_depth_limit(*:Goal, +Limit, -Result***)**

If *Goal* can be proven without recursion deeper than *Limit* levels, `call_with_depth_limit/3` succeeds, binding *Result* to the deepest recursion level used during the proof. Otherwise, *Result* is unified with `depth_limit_exceeded` if the

---

[18] Arities 2..8 are demanded by ISO/IEC 13211-1:1995/Cor.2:2012.

[19] Future versions of the reflective predicate may fake the presense of `call/9..`. Full logical behaviour, generationg all these pseudo predicates is probably undesirable and will become impossible if *max_arity* is removed.

limit was exceeded during the proof, or the entire predicate fails if *Goal* fails without exceeding *Limit*.

The depth-limit is guarded by the internal machinery. This may differ from the depth computed based on a theoretical model. For example, `true/0` is translated into an inlined virtual machine instruction. Also, `repeat/0` is not implemented as below, but as a non-deterministic foreign predicate.

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

As a result, `call_with_depth_limit/3` may still loop infinitely on programs that should theoretically finish in finite time. This problem can be cured by using Prolog equivalents to such built-in predicates.

This predicate may be used for theorem-provers to realise techniques like *iterative deepening*. It was implemented after discussion with Steve Moyle `smoyle@ermine.ox.ac.uk`.

setup_call_cleanup(*:Setup, :Goal, :Cleanup***)**

Calls `(once(Setup), Goal)`. If *Setup* succeeds, *Cleanup* will be called exactly once after *Goal* is finished: either on failure, deterministic success, commit, or an exception. The execution of *Setup* is protected from asynchronous interrupts like `call_with_time_limit/2` (package clib) or `thread_signal/2`. In most uses, *Setup* will perform temporary side-effects required by *Goal* that are finally undone by *Cleanup*.

Success or failure of *Cleanup* is ignored and choice-points it created are destroyed (as `once/1`). If *Cleanup* throws an exception, this is executed as normal.[20]

Typically, this predicate is used to cleanup permanent data storage required to execute *Goal*, close file-descriptors, etc. The example below provides a non-deterministic search for a term in a file, closing the stream as needed.

```
term_in_file(Term, File) :-
        setup_call_cleanup(open(File, read, In),
                           term_in_stream(Term, In),
                           close(In) ).

term_in_stream(Term, In) :-
        repeat,
        read(In, T),
        (   T == end_of_file
        ->  !, fail
        ;   T = Term
        ).
```

Note that it is impossible to implement this predicate in Prolog. The closest approximation would be to read all terms into a list, close the file and call `member/2`. Without

---

[20] BUG: During the execution of *Cleanup*, garbage collection and stack-shifts are disabled.

`setup_call_cleanup/3` there is no way to gain control if the choice-point left by repeat is removed by a cut or an exception.

`setup_call_cleanup/3` can also be used to test determinism of a goal, providing a portable alternative to `deterministic/1`:

```
?- setup_call_cleanup(true,(X=1;X=2), Det=yes).

X = 1 ;

X = 2,
Det = yes ;
```

This predicate is under consideration for inclusion into the ISO standard. For compatibility with other Prolog implementations see `call_cleanup/2`.

setup_call_catcher_cleanup(*:Setup, :Goal, +Catcher, :Cleanup***)**

Similar to `setup_call_cleanup`(*Setup, Goal, Cleanup*) with additional information on the reason of calling *Cleanup*. Prior to calling *Cleanup*, *Catcher* unifies with the termination code (see below). If this unification fails, *Cleanup* is *not* called.

exit

Goal succeeded without leaving any choice-points.

fail

Goal failed.

!

Goal succeeded with choice-points and these are now discarded by the execution of a cut (or other pruning of the search tree such as if-then-else).

exception(*Exception***)**

Goal raised the given *Exception*.

external_exception(*Exception***)**

Goal succeeded with choice-points and these are now discarded due to an exception. For example:

```
?- setup_call_catcher_cleanup(true, (X=1;X=2),
                              Catcher, writeln(Catcher)),
   throw(ball).
external_exception(ball)
ERROR: Unhandled exception: Unknown message: ball
```

call_cleanup(*:Goal, :Cleanup***)**

Same as `setup_call_cleanup`(*true, Goal, Cleanup*). This is provided for compatibility with a number of other Prolog implementations only. Do not use `call_cleanup/2`, if you perform side-effects prior to calling, that will be undone by *Cleanup*. Instead, use `setup_call_cleanup/3` with an appropriate first argument to perform those side-effects.

call_cleanup(*:Goal, +Catcher, :Cleanup***)**
Same as `setup_call_catcher_cleanup`(*true, Goal, Catcher, Cleanup*). The same warning as for `call_cleanup/2` applies.

4.10 ISO compliant Exception handling

SWI-Prolog defines the predicates `catch/3` and `throw/1` for ISO compliant raising and catching of exceptions. In the current implementation (4.0.6), most of the built-in predicates generate exceptions, but some obscure predicates merely print a message, start the debugger and fail, which was the normal behaviour before the introduction of exceptions.

catch(*:Goal, +Catcher, :Recover***)** *[ISO]*
Behaves as `call/1` if no exception is raised when executing *Goal*. If an exception is raised using `throw/1` while *Goal* executes, and the *Goal* is the innermost goal for which *Catcher* unifies with the argument of `throw/1`, all choice-points generated by *Goal* are cut, the system backtracks to the start of `catch/3` while preserving the thrown exception term and *Recover* is called as in `call/1`.

The overhead of calling a goal through `catch/3` is very comparable to `call/1`. Recovery from an exception is much slower, especially if the exception-term is large due to the copying thereof.

throw(*+Exception***)** *[ISO]*
Raise an exception. The system looks for the innermost `catch/3` ancestor for which *Exception* unifies with the *Catcher* argument of the `catch/3` call. See `catch/3` for details.

ISO demands `throw/1` to make a copy of *Exception*, walk up the stack to a `catch/3` call, backtrack and try to unify the copy of *Exception* with *Catcher*. SWI-Prolog delays making a copy of *Exception* and backtracking until it actually found a matching `catch/3` goal. The advantage is that we can start the debugger at the first possible location while preserving the entire exception context if there is no matching `catch/3` goal. This approach can lead to different behaviour if *Goal* and *Catcher* of `catch/3` call share variables. We assume this to be highly unlikely and could not think of a scenario where this is useful.[21]

If an exception is raised in a callback from C (see chapter 9) and not caught in the same call-back, `PL_next_solution()` fails and the exception context can be retrieved using `PL_exception()`.

4.10.1 Debugging and exceptions

Before the introduction of exceptions in SWI-Prolog a runtime error was handled by printing an error message, after which the predicate failed. If the Prolog flag `debug_on_error` was in effect (default), the tracer was switched on. The combination of the error message and trace information is generally sufficient to locate the error.

With exception handling, things are different. A programmer may wish to trap an exception using `catch/3` to avoid it reaching the user. If the exception is not handled by user-code, the interactive top-level will trap it to prevent termination.

---

[21] I'd like to acknowledge Bart Demoen for his clarifications on these matters.

If we do not take special precautions, the context information associated with an unexpected exception (i.e., a programming error) is lost. Therefore, if an exception is raised, which is not caught using `catch/3` and the top-level is running, the error will be printed, and the system will enter trace mode.

If the system is in an non-interactive callback from foreign code and there is no `catch/3` active in the current context, it cannot determine whether or not the exception will be caught by the external routine calling Prolog. It will then base its behaviour on the Prolog flag `debug_on_error`:

- *current_prolog_flag(debug_on_error, false)*
  The exception does not trap the debugger and is returned to the foreign routine calling Prolog, where it can be accessed using `PL_exception()`. This is the default.
- *current_prolog_flag(debug_on_error, true)*
  If the exception is not caught by Prolog in the current context, it will trap the tracer to help analysing the context of the error.

While looking for the context in which an exception takes place, it is advised to switch on debug mode using the predicate `debug/0`. The hook `prolog_exception_hook/4` can be used to add more debugging facilities to exceptions. An example is the library `http/http_error`, generating a full stack trace on errors in the HTTP server library.

4.10.2 The exception term

Built-in predicates generates exceptions using a term `error`(*Formal, Context*). The first argument is the 'formal' description of the error, specifying the class and generic defined context information. When applicable, the ISO error-term definition is used. The second part describes some additional context to help the programmer while debugging. In its most generic form this is a term of the form `context`(*Name/Arity, Message*), where *Name/Arity* describes the built-in predicate that raised the error, and *Message* provides an additional description of the error. Any part of this structure may be a variable if no information was present.

4.10.3 Printing messages

The predicate `print_message/2` is used to print a message term in a human readable format. The other predicates from this section allow the user to refine and extend the message system. A common usage of `print_message/2` is to print error messages from exceptions. The code below prints errors encountered during the execution of *Goal*, without further propagating the exception and without starting the debugger.

```
        ...,
        catch(Goal, E,
              ( print_message(error, E),
                fail
              )),
        ...
```

Another common use is to defined `message_hook/3` for printing messages that are normally *silent*, suppressing messages, redirecting messages or make something happen in addition to printing the message.

print_message(+*Kind*, +*Term***)**

The predicate `print_message/2` is used to print messages, notably from exceptions, in a human-readable format. *Kind* is one of `informational`, `information`, `banner`, `warning`, `error`, `help` or `silent`. A human-readable message is printed to the stream `user_error`.

If the Prolog flag `verbose` is `silent`, messages with *Kind* `informational`, or `banner` are treated as `silent`. See `-q`. The *Kind* `information` is intended for informational messages that should also be printed in silent mode, typically because the user requests of the information. The predicate `time/1` is an example.

The predicate `print_message/2` first translates the *Term* into a list of 'message lines' (see `print_message_lines/3` for details). Next, it calls the hook `message_hook/3` to allow the user intercepting the message. If `message_hook/3` fails it prints the message unless *Kind* is silent.

The `print_message/2` predicate and its rules are in the file ⟨*plhome*⟩`/boot/messages.pl`, which may be inspected for more information on the error messages and related error terms. If you need to write messages from your own predicates, it is recommended to reuse the existing message terms if applicable. If no existing message term is applicable, invent a fairly unique term that represents the event and define a rule for the multifile predicate prolog:message//1. See section 4.10.3 for a deeper discussion and examples.

See also `message_to_string/2`.

print_message_lines(+*Stream*, +*Prefix*, +*Lines***)**

Print a message (see `print_message/2`) that has been translated to a list of message elements. The elements of this list are:

⟨*Format*⟩**-**⟨*Args*⟩

Where *Format* is an atom and *Args* is a list of format argument. Handed to `format/3`.

flush

If this appears as the last element, *Stream* is flushed (see `flush_output/1`) and no final newline is generated. This is combined with a subsequent message that starts with `at_same_line` to complete the line.

at_same_line

If this appears as first element, no prefix is printed for the first line and the line-position is not forced to 0 (see `format/1`, `~N`).

ansi(+*Attributes*, +*Format*, +*Args***)**

This message may be intercepted by means of the hook prolog:message_line_element/2. The library `ansi_term` implements this hook to achieve coloured output. If it is not intercepted it invokes `format`(*Stream, Format, Args*).

nl

A new line is started. If the message is not complete, *Prefix* is printed before the remainder of the message.

begin(*Kind*, *Var***)**
end(*Var***)**

The entire message is headed by `begin`(*Kind, Var*) and ended by `end`(*Var*). This feature is used by e.g., library `ansi_term` to colour entire messages.

⟨*Format*⟩
Handed to `format/3` as `format`(*Stream, Format, []*). Deprecated because it is ambiguous of *Format* collides with one of the atomic commands.

See also `print_message/2` and `message_hook/3`.

message_hook(+*Term,* +*Kind,* +*Lines***)**
Hook predicate that may be defined in the module `user` to intercept messages from `print_message/2`. *Term* and *Kind* are the same as passed to `print_message/2`. *Lines* is a list of format statements as described with `print_message_lines/3`. See also `message_to_string/2`.

This predicate must be defined dynamic and multifile to allow other modules defining clauses for it too.

prolog:message_line_element(+*Stream,* +*Term***)**
This hook is called to print the individual elements of a message from `print_message_lines/3`. This hook is used by e.g., library `ansi_term` to colour messages on ANSI capable terminals.

message_to_string(+*Term,* -*String***)**
Translates a message-term into a string object (see section 4.23). Primarily intended to write messages to Windows in XPCE (see section 1.5) or other GUI environments.

Printing from libraries

Libraries should *not* use `format/3` or other output predicates directly. Libraries that print informational output directory to the console are hard to use from code that depend on your textual output, such as a CGI script. The predicates in section 4.10.3 define the API for dealing with messages. The idea behind this is that a library that wants to provide information about its status, progress, events or problems calls `print_message/2`. The first argument is the *level*. The supported levels are described with `print_message/2`. Libraries typically use `informational` and `warning`, while libraries should use exceptions for errors (see `throw/1`, `type_error/2`, etc.).

The second argument is an arbitrary Prolog term that carries te information of the message, but *not* the precise text. The text is defined by the grammar rule prolog:message//1. This distinction is made to allow for translations and to allow hooks processing the information in a different way (e.g., translate progress messages into a progress-bar).

For example, suppose we have a library that must download data from the internet (e.g., based on `http_open/3`). The library wants to print the progress after each downloaded file. The code below is a good skeleton:

```
download_urls(List) :-
        length(List, Total),
        forall(nth1(I, List, URL),
               (   download_url(URL),
                   print_message(informational,
                                 download_url(URL, I, Total)))).
```

The programmer can now specify the default textual output using the rule below. Note that this rule may be in the same file or anywhere else. Notably, the application may come with several rule-sets for different languages. This, and the user-hook example below are the reason to represent the message as a compound term rather than a string. This is similar to using message-numbers in non-symbolic languages. The documentation of `print_message_lines/3` describes the elements that may appear in the output list.

```
:- multifile
        prolog:message//1.

prolog:message(download_url(URL, I, Total)) -->
        { Perc is round(I*100/Total) },
        [ 'Downloaded ~w; ~D from ~D (~d%)'-[URL, I, Total, Perc] ].
```

A *user* of the library may define rules for `message_hook/3`. The rule below acts on the message-content. Other applications can act on the message-level and, for example, popup a message-box for warnings and errors.

```
:- multifile user:message_hook/3.

message_hook(download_url(URL, I, Total), _Kind, _Lines) :-
        <send this information to a GUI component>
```

In addition, using the commandline option `-q`, the user can disable all *informational* messages.

4.11 Handling signals

As of version 3.1.0, SWI-Prolog is capable to handle software interrupts (signals) in Prolog as well as in foreign (C) code (see section 9.4.13).

Signals are used to handle internal errors (execution of a non-existing CPU instruction, arithmetic domain errors, illegal memory access, resource overflow, etc.), as well as for dealing asynchronous inter-process communication.

Signals are defined by the POSIX standard and part of all Unix machines. The MS-Windows Win32 provides a subset of the signal handling routines, lacking the vital functionality to raise a signal in another thread for achieving asynchronous inter-process (or inter-thread) communication (Unix kill() function).

on_signal(+*Signal*, -*Old*, :*New***)**

Determines the reaction on *Signal*. *Old* is unified with the old behaviour, while the behaviour is switched to *New*. As with similar environment-control predicates, the current value is retrieved using `on_signal(Signal, Current, Current)`.

The action description is an atom denoting the name of the predicate that will be called if *Signal* arrives. `on_signal/3` is a meta-predicate, which implies that ⟨*Module*⟩:⟨*Name*⟩ refers the ⟨*Name*⟩/1 in the module ⟨*Module*⟩. The handler is called with a single argument: the name of the signal as an atom. The Prolog names for signals is explained below.

Two predicate-names have special meaning. `throw` implies Prolog will map the signal onto a Prolog exception as described in section 4.10. `default` resets the handler to the settings active before SWI-Prolog manipulated the handler.

Signals bound to a foreign function through `PL_signal()` are reported using the term `$foreign_function`(*Address*).

After receiving a signal mapped to `throw`, the exception raised has the structure

`error(signal(`⟨*SigName*⟩`,` ⟨*SigNum*⟩`),` ⟨*Context*⟩`)`

The signal names are defined by the POSIX standard as symbols of the form `SIG`⟨*SIGNAME*⟩. The Prolog name for a signal is the lowercase version of ⟨*SIGNAME*⟩. The predicate `current_signal/3` may be used to map between names and signals.

Initially, some signals are mapped to `throw`, while all other signals are `default`. The following signals throw an exception: `fpe`, `alrm`, `xcpu`, `xfsz` and `vtalrm`.

current_signal(*?Name, ?Id, ?Handler***)**
Enumerate the currently defined signal handling. *Name* is the signal name, *Id* is the numerical identifier and *Handler* is the currently defined handler (see `on_signal/3`).

4.11.1 Notes on signal handling

Before deciding to deal with signals in your application, please consider the following:

- *Portability*
  On MS-Windows, the signal interface is severely limited. Different Unix brands support different sets of signals, and the relation between signal name and number may vary. Currently, the system only supports signals numbered 1 to 32[22]. Installing a signal outside the limited set of supported signals in MS-Windows crashes the application.

- *Safety*
  Immediately delivered signals (see below) are unsafe. This implies that foreign functions called from a handler cannot safely use the SWI-Prolog API and cannot use C longjmp(). Handlers defined as `throw` are unsafe. Handlers defined to call a predicate are safe. Note that the predicate can call `throw/1`, but the delivery is delayed until Prolog is in a safe state.

  The C-interface described in section 9.4.13 provides the option `PL_SIGSYNC` to select either safe synchronous or unsafe asynchronous delivery.

- *Time of delivery*
  Using `throw` or a foreign handler, signals are delivered immediately (as defined by the OS). When using a Prolog predicate, delivery is delayed to a safe moment. Blocking system calls or foreign loops may cause long delays. Foreign code can improve on that by calling `PL_handle_signals()`.

  Signals are blocked when the garbage collector is active.

---

[22] TBD: the system should support the Unix realtime signals

4.12 DCG Grammar rules

,Grammar rules form a comfortable interface to *difference-lists*. They are designed both to support writing parsers that build a parse tree from a list of characters or tokens as for generating a flat list from a term.

Grammar rules look like ordinary clauses using `-->/2` for separating the head and body rather than `:-/2`. Expanding grammar rules is done by `expand_term/2`, which adds two additional argument to each term for representing the difference list.

The body of a grammar rule can contain three types of terms. A callable term is interpreted as a reference to a grammar-rule. Code between `{...}` is interpreted as plain Prolog code and finally, a list is interpreted as a sequence of *literals*. The Prolog control-constructs (`\+/1`, `->/2`, `;`//2, `,/2` and `!/0`) can be used in grammar rules.

We illustrate the behaviour by defining a rule-set for parsing an integer.

```
integer(I) -->
        digit(D0),
        digits(D),
        { number_codes(I, [D0|D])
        }.

digits([D|T]) -->
        digit(D), !,
        digits(T).
digits([]) -->
        [].

digit(D) -->
        [D],
        { code_type(D, digit)
        }.
```

Grammar rule-sets are called using the built-in predicates `phrase/2` and `phrase/3`:

phrase(+*DCGBody, ?List***)**

Equivalent to `phrase(`*DCGBody*`,` *InputList*`, [])`.

phrase(+*DCGBody, ?List, ?Rest***)**

True when *DCGBody* applies to the difference *List*/*Rest*. Although *DCGBody* is typically a *callable* term that denotes a grammar rule, it can be any term that is valid as the body of DCG rule.

The example below calls the rule set 'integer' defined in section 4.12, binding *Rest* to the remainder of the input after matching the integer.

```
?- phrase(integer(X), "42 times", Rest).
X = 42
Rest = [32, 116, 105, 109, 101, 115]
```

The next example calls example exploits a complete body.

```
digit_weigth(W) -->
        [D],
        { code_type(D, digit(W)) }.

?- phrase(("Version ",
           digit_weigth(Major),".",digit_weigth(Minor)),
          "Version 3.4").
Major = 3,
Minor = 4.
```

See also `portray_text/1`, which can be used to print lists of character codes as a string to the toplevel and debugger to facilitate debugging of DCGs that process character codes. The library `apply_macros` compiles `phrase/3` if the argument is sufficiently instantiated, eliminating the runtime overhead of translating *DCGBody* and meta-calling.

As stated above, grammar rules are a general interface to difference lists. To illustrate that, we show a DCG-based implementation of `reverse/2`:

```
reverse(List, Reversed) :-
        phrase(reverse(List), Reversed).

reverse([])     --> [].
reverse([H|T]) --> reverse(T), [H].
```

4.13 Database

SWI-Prolog offers three different database mechanisms. The first one is the common assert/retract mechanism for manipulating the clause database. As facts and clauses asserted using `assert/1` or one of its derivatives become part of the program these predicates compile the term given to them. `retract/1` and `retractall/1` have to unify a term and therefore have to decompile the program. For these reasons the assert/retract mechanism is expensive. On the other hand, once compiled, queries to the database are faster than querying the recorded database discussed below. See also `dynamic/1`.

The second way of storing arbitrary terms in the database is using the 'recorded database'. In this database terms are associated with a *key*. A key can be an atom, small integer or term. In the last case only the functor and arity determine the key. Each key has a chain of terms associated with it. New terms can be added either at the head or at the tail of this chain.

Following the Edinburgh tradition, SWI-Prolog provides database keys to clauses and records in the recorded database. As of 5.9.10, these keys are represented by non-textual atoms ('blobs', see section 9.4.7), which makes accessing the database through references safe.

The third mechanism is a special purpose one. It associates an integer or atom with a key, which is an atom, integer or term. Each key can only have one atom or integer associated with it.

abolish(*:PredicateIndicator***)** *[ISO]*

Removes all clauses of a predicate with functor *Functor* and arity *Arity* from the database. All predicate attributes (dynamic, multifile, index, etc.) are reset to their defaults. Abolishing an imported predicate only removes the import link; the predicate will keep its old definition in its definition module.

According to the ISO standard, `abolish/1` can only be applied to dynamic procedures. This is odd, as for dealing with dynamic procedures there is already `retract/1` and `retractall/1`. The `abolish/1` predicate has been introduced in DEC-10 Prolog precisely for dealing with static procedures. In SWI-Prolog, `abolish/1` works on static procedures, unless the prolog flag `iso` is set to `true`.

It is advised to use `retractall/1` for erasing all clauses of a dynamic predicate.

abolish(*+Name, +Arity***)**

Same as `abolish`(*Name/Arity*). The predicate `abolish/2` conforms to the Edinburgh standard, while `abolish/1` is ISO compliant.

copy_predicate_clauses(*:From, :To***)**

Copy all clauses of predicate *From* to *To*. The predicate *To* must be dynamic or undefined. If *To* is undefined, it is created as a dynamic predicate holding a copy of the clauses of *From*. If *To* is a dynamic predicate, the clauses of *From* are added (as in `assertz/1`) to the clauses of *To*. *To* and *From* must have the same arity. Acts as if defined by the program below, but at a much better performance by avoiding decompilation and compilation.

```
copy_predicate_clauses(From, To) :-
        head(From, MF:FromHead),
        head(To, MT:ToHead),
        FromHead =.. [_|Args],
        ToHead =.. [_|Args],
        forall(clause(MF:FromHead, Body),
               assertz(MT:ToHead, Body)).

head(From, M:Head) :-
        strip_module(From, M, Name/Arity),
        functor(Head, Name, Arity).
```

redefine_system_predicate(*+Head***)**

This directive may be used both in module `user` and in normal modules to redefine any system predicate. If the system definition is redefined in module `user`, the new definition is the default definition for all sub-modules. Otherwise the redefinition is local to the module. The system definition remains in the module `system`.

Redefining system predicate facilitates the definition of compatibility packages. Use in other context is discouraged.

retract(*+Term***)** *[ISO]*

When *Term* is an atom or a term it is unified with the first unifying fact or clause in the database. The fact or clause is removed from the database.

retractall(+*Head***)** *[ISO]*

All facts or clauses in the database for which the *head* unifies with *Head* are removed. If *Head* refers to a predicate that is not defined, it is implicitly created as a dynamic predicate. See also `dynamic/1`.[23]

asserta(+*Term***)** *[ISO]*

Assert a fact or clause in the database. *Term* is asserted as the firsr fact or clause of the corresponding predicate. Equivalent to `assert/1`, but *Term* is asserted as first clause or fact of the predicate.

assertz(+*Term***)** *[ISO]*

Equivalent to `asserta/1`, but *Term* is asserted as the last clause or fact of the predicate.

assert(+*Term***)**

Equivalent to `assertz/1`. Deprecated: new code should use `assertz/1`.

asserta(+*Term, -Reference***)**

Asserts a clause as `asserta/1` and unifies *Reference* with a handle to this clause. The handle can be used to access this specific clause using `clause/3` and `erase/1`.

assertz(+*Term, -Reference***)**

Equivalent to `asserta/1`, asserting the new clause as the last clause of the predicate.

assert(+*Term, -Reference***)**

Equivalent to `assertz/2`.

recorda(+*Key*, +*Term, -Reference***)**

Assert *Term* in the recorded database under key *Key*. *Key* is a small integer (range `min_tagged_integer` ...`max_tagged_integer`, atom or compound term. If the key is a compound term, only the name and arity define the key. *Reference* is unified with an opaque handle to the record (see `erase/1`).

recorda(+*Key*, +*Term***)**

Equivalent to `recorda(`*Key*`,` *Value*`, _)`.

recordz(+*Key*, +*Term, -Reference***)**

Equivalent to `recorda/3`, but puts the *Term* at the tail of the terms recorded under *Key*.

recordz(+*Key*, +*Term***)**

Equivalent to `recordz(`*Key*`,` *Value*`, _)`.

recorded(?*Key, ?Value, ?Reference***)**

True if *Value* is recorded under *Key* and has the given database *Reference*. If *Reference* is given, this predicate is semi-deterministic. Otherwise, it must be considered non-deterministic. If neither *Reference* nor *Key* is given, the triples are generated as in the code snippet below.[24] See also `current_key/1`.

---

[23] The ISO standard only allows using `dynamic/1` as a *directive*.

[24] Note that, without a given *Key*, some implementations return triples in the order defined by `recorda/2` and `recordz/2`.

```
        current_key(Key),
        recorded(Key, Value, Reference)
```

recorded(+*Key*, -*Value***)**

Equivalent to `recorded(`*Key*`, `*Value*`, _)`.

erase(+*Reference***)**

Erase a record or clause from the database. *Reference* is an db-reference returned by `recorda/3` or `recorded/3`, `clause/3`, `assert/2`, `asserta/2` or `assertz/2`. Fail silently if the referenced object no longer exists.

instance(+*Reference*, -*Term***)**

Unify *Term* with the referenced clause or database record. Unit clauses are represented as *Head* :- *Body*.

flag(+*Key*, -*Old*, +*New***)**

Key is an atom, integer or term. As with the recorded database, if *Key* is a term, only the name and arity are used to locate the flag. Unify *Old* with the old value associated with *Key*. If the key is used for the first time *Old* is unified with the integer 0. Then store the value of *New*, which should be an integer, float, atom or arithmetic expression, under *Key*. `flag/3` is a fast mechanism for storing simple facts in the database. The flag database is shared between threads and updates are atomic, making it suitable for generating unique integer counters.[25]

4.13.1 Update view

Traditionally, Prolog systems used the *immediate update view*: new clauses became visible to predicates backtracking over dynamic predicates immediately and retracted clauses became invisible immediately.

Starting with SWI-Prolog 3.3.0 we adhere the *logical update view*, where backtrackable predicates that enter the definition of a predicate will not see any changes (either caused by `assert/1` or `retract/1`) to the predicate. This view is the ISO standard, the most commonly used and the most 'safe'.[26] Logical updates are realised by keeping reference-counts on predicates and *generation* information on clauses. Each change to the database causes an increment of the generation of the database. Each goal is tagged with the generation in which it was started. Each clause is flagged with the generation it was created as well as the generation it was erased. Only clauses with 'created' ... 'erased' interval that encloses the generation of the current goal are considered visible.

4.13.2 Indexing databases

The indexing capabilities of SWI-Prolog are described in section 2.17. Summarizing, SWI-Prolog creates indexes for any applicable argument, but indexes only on one argument and does not index on arguments of compound terms. The predicates below provide building blocks to circumvent the limitations of the current indexing system.

---

[25] The `flag/3` predicate is not portable. Non-backtrackable global variables (`nb_setval/2`) and non-backtrackable assignment (`nb_setarg/3`) are more widely recognised special-purpose alternatives for non-backtrackable and/or global state.

[26] For example, using the immediate update view, no call to a dynamic predicate is deterministic.

Programs that aim at portability should consider using `term_hash/2` and `term_hash/4` to design their database such that indexing on constant or functor (name/arity reference) on the first argument is sufficient.

term_hash(+*Term, -HashKey***)** *[det]*

If *Term* is a ground term (see `ground/1`), *HashKey* is unified with a positive integer value that may be used as a hash-key to the value. If *Term* is not ground, the predicate leaves *HashKey* an unbound variable. Hash keys are in the range $0 ::: 16; 777; 215$, the maximal integer that can be stored efficiently on both 32 and 64 bit platforms.

This predicate may be used to build hash-tables as well as to exploit argument-indexing to find complex terms more quickly.

The hash-key does not rely on temporary information like addresses of atoms and may be assumed constant over different invocations and versions of SWI-Prolog.[27] Hashes differ between big and little endian machines. The `term_hash/2` predicate is cycle-safe.[28]

term_hash(+*Term, +Depth, +Range, -HashKey***)** *[det]*

As `term_hash/2`, but only considers *Term* to the specified *Depth*. The toplevel term has depth 1, its arguments have depth 2, etc. I.e. $Depth = 0$ hashes nothing; $Depth = 1$ hashes atomic values or the functor and arity of a compound term, not its arguments; $Depth = 2$ also indexes the immediate arguments, etc.

HashKey is in the range $[0 ::: Range - 1]$. *Range* must be in the range $[1 ::: 2147483647]$

variant_sha1(+*Term, -SHA1***)** *[det]*

Compute an SHA1-hash from *Term*. The hash is represented as a 40-byte hexadecimal atom. Unlike `term_hash/2` and friends, this predicate produces a hash-key for non-ground terms. The hash is invariant over variable-renaming (see `=@=/2`) and constants over different invocations of Prolog.[29]

This predicate raises an exeption when trying to compute the hash on a cyclic term or attributed term. Attributed terms are not handled because `subsumes_chk/2` is not considered well defined for attributed terms. Cyclic terms are not supported because this would require establishing a canonical cycle. I.e., given A=[a—A] and B=[a,a—B], *A* and *B* should produce the same hash. This is not (yet) implemented.

This hash was developed for lookup of solutions to a goal stored in a table. By using a cryptographic hash, heuristic algorithms can often ignore the possibility of hash-colisions and thus avoid storing the goal-term itself as well as testing using `=@=/2`.

4.14 Declaring predicates properties

This section describes directives which manipulate attributes of predicate definitions. The functors `dynamic/1`, `multifile/1` and `discontiguous/1` are operators of priority 1150 (see `op/3`), which implies the list of predicates they involve can just be a comma separated list:

---

[27] Last change: version 5.10.4

[28] BUG: All arguments that (indirecly) lead to the a cycle have the same hash-key.

[29] BUG: The hash depends on word-order (big/little-endian) and the wordsize (32/64 bits).

```
:- dynamic
        foo/0,
        baz/2.
```

On SWI-Prolog all these directives are just predicates. This implies they can also be called by a program. Do not rely on this feature if you want to maintain portability to other Prolog implementations.

dynamic *:PredicateIndicator, . . .* *[ISO]*
Informs the interpreter that the definition of the predicate(s) may change during execution (using `assert/1` and/or `retract/1`). In the multi-threaded version, the clauses of dynamic predicates are shared between the threads. The directive `thread_local/1` provides an alternative where each threads has its own clause-list for the predicate. Dynamic predicates can be turned into static ones using `compile_predicates/1`.

compile_predicates(*:ListOfNameArity***)**
Compile a list of specified dynamic predicates (see `dynamic/1` and `assert/1`) into normal static predicates. This call tells the Prolog environment the definition will not change anymore and further calls to `assert/1` or `retract/1` on the named predicates raise a permission error. This predicate is designed to deal with parts of the program that is generated at runtime but does not change during the remainder of the program execution.[30]

multifile *:PredicateIndicator, . . .* *[ISO]*
Informs the system that the specified predicate(s) may be defined over more than one file. This stops `consult/1` from redefining a predicate when a new definition is found.

discontiguous *:PredicateIndicator, . . .* *[ISO]*
Informs the system that the clauses of the specified predicate(s) might not be together in the source file. See also `style_check/1`.

public *:PredicateIndicator, . . .*
Instructs the cross-referencer that the predicate can be called. It has no semantics.[31]

4.15 Examining the program

current_atom(*-Atom***)**
Successively unifies *Atom* with all atoms known to the system. Note that `current_atom/1` always succeeds if *Atom* is instantiated to an atom.

current_blob(*?Blob, ?Type***)**
Examine the type or enumerate blobs of the given *Type*. Typed blobs are supported through the foreign language interface for storing arbitrary BLOBS (Binary Large Object) or handles to external entities. See section 9.4.7 for details.

---

[30] The specification of this predicate is from Richard O'Keefe. The implementation is allowed to optimise the predicate. This is not yet implemented. In multi-threaded Prolog however, static code runs faster as it does not require synchronisation. This is particularly true on SMP hardware.

[31] This declaration is compatible to SICStus. In YAP, `public/1` instructs the compiler to keep the source. As the source is always available in SWI-Prolog, our current interpretation also enhanced the compatibility to YAP.

current_functor(*?Name, ?Arity***)**
Successively unifies *Name* with the name and *Arity* with the arity of functors known to the system.

current_flag(*-FlagKey***)**
Successively unifies *FlagKey* with all keys used for flags (see `flag/3`).

current_key(*-Key***)**
Successively unifies *Key* with all keys used for records (see `recorda/3`, etc.).

current_predicate(*:PredicateIndicator***)** *[ISO]*
True if *PredicateIndicator* is a currently defined predicate. A predicate is considered defined if it exists in the specified module, is imported into the module or is defined in one of the modules from which the predicate will be imported if it is called (see section 5.9). Note that `current_predicate/1` does *not* succeed for predicates that can be *autoloaded*. See also `current_predicate/2` and `predicate_property/2`.

If *PredicateIndicator* is not fully specified, the predicate only generates values that are defined in or already imported into the target module. Generating all callable predicates therefore requires enumerating modules using `current_module/1`. Generating predicates callable in a given module requires enumerating the import modules using `import_module/2` and the auto-loadable predicates using the `predicate_property/2` `autoload`.

current_predicate(*?Name, :Head***)**
Classical pre-ISO implementation of `current_predicate/1`, where the predicate is represented by the head-term. The advantage is that this can be used for checking existence of a predicate before calling it without the need for `functor/3`:

```
call_if_exists(G) :-
        current_predicate(_, G),
        call(G).
```

Because of this intended usage, `current_predicate/2` also succeeds if the predicate can be autoloaded. Unfortunately, checking the autoloader makes this predicate relatively slow; in particular because a failed lookup of the autoloader will cause the autoloader to verify that its index is up-to-date.

predicate_property(*:Head, ?Property***)**
True when *Head* refers to a predicate that has property *Property*. With sufficiently instantiated *Head*, `predicate_property/2` tries to resolve the predicate the same way as calling it would do: if the predicate is not defined it scans the default modules (see `default_module/2`) and finally tries the autoloader. Unlike calling, failure to find the target predicate causes `predicate_property/2` to fail silently. If *Head* is not sufficiently bound, only currently locally defined and already imported predicates are enumerated. See `current_predicate/1` for enumerating all predicates. A common issue concerns *generating* all built-in predicates. This can be achieved using the code below.

```
generate_built_in(Name/Arity) :-
        predicate_property(system:Head, built_in),
        functor(Head, Name, Arity),
        \+ sub_atom(Name, 0, _, _, $).  % discard reserved names
```

Property is one of:

autoload(*File***)**
Is true if the predicate can be autoloaded from the file *File*. Like `undefined`, this property is *not* generated.

built_in
Is true if the predicate is locked as a built-in predicate. This implies it cannot be redefined in its definition module and it can normally not be seen in the tracer.

dynamic
Is true if `assert/1` and `retract/1` may be used to modify the predicate. This property is set using `dynamic/1`.

exported
Is true if the predicate is in the public list of the context module.

imported_from(*Module***)**
Is true if the predicate is imported into the context module from module *Module*.

file(*FileName***)**
Unify *FileName* with the name of the source file in which the predicate is defined. See also `source_file/2` and the propery `line_count`. Note that this reports the file of the first clause of a predicate. A more robust interface can be achieved using `nth_clause/3` and `clause_property/2`.

foreign
Is true if the predicate is defined in the C language.

indexed(*Indexes***)**
Indexes[32] is a list of additional (hash) indexes on the predicate. Each element of the list is a term *ArgSpec*-*Index*. Currently *ArgSpec* is an integer denoting the argument position and *Index* is a term `hash`(*Buckets, Speedup, IsList*). Here *Buckets* is the number of buckets in the hash and *Speedup* is the expected speedup relative to trying all clauses linearly. *IsList* indicates that a list is created for all clauses with the same key. This is currently not used.

interpreted
Is true if the predicate is defined in Prolog. We return true on this because, although the code is actually compiled, it is completely transparent, just like interpreted code.

iso
Is true if the predicate is covered by the ISO standard (ISO/IEC 13211-1).

line_count(*LineNumber***)**
Unify *LineNumber* with the line number of the first clause of the predicate. Fails if the predicate is not associated with a file. See also `source_file/2`. See also the `file` property above, notably the reference to `clause_property/2`.

---

[32] This predicate property should be used for analysis and statistics only. The exact representation of *Indexes* may change between versions.

multifile
Is true there may be multiple (or no) files providing clauses for the predicate. This property is set using `multifile/1`.

meta_predicate(*Head*)
If the predicate is declared as a meta-predicate using `meta_predicate/1`, Unify *Head* with the head-pattern. The head-pattern is a compound term with the same name and arity as the predicate where each argument of the term is a meta predicate specifier. See `meta_predicate/1` for details.

nodebug
Details of the predicate are not shown by the debugger. This is the default for built-in predicates. User predicates can be compiled this way using the Prolog flag `generate_debug_info`.

notrace
Do not show ports of this predicate in the debugger.

number_of_clauses(*ClauseCount*)
Unify *ClauseCount* to the number of clauses associated with the predicate. Fails for foreign predicates.

public
Predicate is declared public using `public/1`. Note that without further definition, public predicates are considered undefined and this property is *not* reported.

thread_local
If true (only possible on the multi-threaded version) each thread has its own clauses for the predicate. This property is set using `thread_local/1`.

transparent
Is true if the predicate is declared transparent using the `module_transparent/1` or `meta_predicate/1` declaration. In the latter case the property `meta_predicate`(*Head*) is also provided. See chapter 5 for details.

undefined
Is true if a procedure definition block for the predicate exists, but there are no clauses for it and it is not declared dynamic or multifile. This is true if the predicate occurs in the body of a loaded predicate, an attempt to call it has been made via one of the meta-call predicates or the predicate had a definition in the past. See the library package `check` for example usage.

visible
True when predicate can be called without raising a predicate existence error. This means that the predicate is (1) defined, (2) can be inherited from one of the default modules (see `default_module/2`) or (3) can be autoloaded. The behaviour is logically consistent iff the propery `visible` is provided explicitly. If the property is left unbound, only defined predicates are enumerated.

volatile
If true, the clauses are not saved into a saved-state by `qsave_program/[1,2]`. This property is set using `volatile/1`.

dwim_predicate(*+Term, -Dwim*)
'Do What I Mean' ('dwim') support predicate. *Term* is a term, which name and arity are used

as a predicate specification. *Dwim* is instantiated with the most general term built from *Name* and the arity of a defined predicate that matches the predicate specified by *Term* in the 'Do What I Mean' sense. See `dwim_match/2` for 'Do What I Mean' string matching. Internal system predicates are not generated, unless the access level is `system` (see `access_level`). Backtracking provides all alternative matches.

clause(*:Head, ?Body***)** *[ISO]*

True if *Head* can be unified with a clause head and *Body* with the corresponding clause body. Gives alternative clauses on backtracking. For facts *Body* is unified with the atom *true*.

clause(*:Head, ?Body, ?Reference***)**

Equivalent to `clause/2`, but unifies *Reference* with a unique reference to the clause (see also `assert/2`, `erase/1`). If *Reference* is instantiated to a reference the clause's head and body will be unified with *Head* and *Body*.

nth_clause(*?Pred, ?Index, ?Reference***)**

Provides access to the clauses of a predicate using their index number. Counting starts at 1. If *Reference* is specified it unifies *Pred* with the most general term with the same name/arity as the predicate and *Index* with the index-number of the clause. Otherwise the name and arity of *Pred* are used to determine the predicate. If *Index* is provided *Reference* will be unified with the clause reference. If *Index* is unbound, backtracking will yield both the indices and the references of all clauses of the predicate. The following example finds the 2nd clause of `append/3`:

```
?- use_module(library(lists)).
...
?- nth_clause(append(_,_,_), 2, Ref), clause(Head, Body, Ref).
Ref = <clause>(0x994290),
Head = lists:append([_G23|_G24], _G21, [_G23|_G27]),
Body = append(_G24, _G21, _G27).
```

clause_property(*+ClauseRef, -Property***)**

Queries properties of a clause. *ClauseRef* is a reference to a clause as produced by `clause/3`, `nth_clause/3` or `prolog_frame_attribute/3`. *Property* is one of the following:

file(*FileName***)**

Unify *FileName* with the name of the source file in which the clause is defined. Fails if the clause is not associated to a file.

line_count(*LineNumber***)**

Unify *LineNumber* with the line number of the clause. Fails if the clause is not associated to a file.

fact

True if the clause has no body.

erased

True if the clause has been erased, but not yet reclaimed because it is referenced.

4.16 Input and output

SWI-Prolog provides two different packages for input and output. The native I/O system is based on the ISO standard predicates `open/3`, `close/1` and friends.[33] Being more widely portable and equipped with a clearer and more robust specification, new code is encouraged to use these predicates for manipulation of I/O streams.

Section 4.16.2 describes `tell/1`, `see/1` and friends, providing I/O in the spirit of the traditional Edinburgh standard. These predicates are layered on top of the ISO predicates. Both packages are fully integrated; the user may switch freely between them.

4.16.1 ISO Input and Output Streams

The predicates described in this section provide ISO compliant I/O, where streams are explicitly created using the predicate `open/3`. The resulting stream identifier is then passed as a parameter to the reading and writing predicates to specify the source or destination of the data.

This schema is not vulnerable to filename and stream ambiguities as well as changes to the working directory. On the other hand, using the notion of current-I/O simplifies reusability of code without the need to pass arguments around. E.g., see `with_output_to/2`.

SWI-Prolog streams are, compatible to the ISO standard, either input or output streams. To accomodate portability to other systems, a pair of streams can be packed into a *stream-pair*. See `stream_pair/3` for details.

SWI-Prolog stream-handles are unique symbols that have no syntactical representation. They are written as `\bnfmeta{stream}(hex-number)`, which is not valid input for `read/1`. They are realised using a *blob* of type `stream` (see `blob/2` and section 9.4.7).

open(+*SrcDest*, +*Mode*, -*Stream*, +*Options***)** *[ISO]*

ISO compliant predicate to open a stream. *SrcDest* is either an atom, specifying a file, or a term ‘`pipe(`*Command*`)`’, like `see/1` and `tell/1`. *Mode* is one of `read`, `write`, `append` or `update`. Mode `append` opens the file for writing, positioning the file-pointer at the end. Mode `update` opens the file for writing, positioning the file-pointer at the beginning of the file without truncating the file. *Stream* is either a variable, in which case it is bound to an integer identifying the stream, or an atom, in which case this atom will be the stream identifier.[34] The *Options* list can contain the following options:

type(*Type***)**

Using type `text` (default), Prolog will write a text-file in an operating-system compatible way. Using type `binary` the bytes will be read or written without any translation. See also the option `encoding`.

alias(*Atom***)**

Gives the stream a name. Below is an example. Be careful with this option as stream-names are global. See also `set_stream/2`.

```
?- open(data, read, Fd, [alias(input)]).

        ...,
```

---

[33] Actually based on Quintus Prolog, providing this interface before the ISO standard existed.

[34] New code should use the `alias`(*Alias*) option for compatibility with the ISO standard

```
        read(input, Term),
        ...
```

encoding(*Encoding*)
Define the encoding used for reading and writing text to this stream. The default encoding for type `text` is derived from the Prolog flag `encoding`. For `binary` streams the default encoding is `octet`. For details on encoding issues, see section 2.18.1.

bom(*Bool*)
Check for a BOM (*Byte Order Marker*) or write one. If omitted, the default is `true` for mode `read` and `false` for mode `write`. See also `stream_property/2` and especially section 2.18.1 for a discussion on this feature.

eof_action(*Action*)
Defines what happens if the end of the input stream is reached. Action `eof_code` makes `get0/1` and friends return -1 and `read/1` and friends return the atom `end_of_file`. Repetitive reading keeps yielding the same result. Action `error` is like `eof_code`, but repetitive reading will raise an error. With action `reset`, Prolog will examine the file again and return more data if the file has grown.

buffer(*Buffering*)
Defines output buffering. The atom `full` (default) defines full buffering, `line` buffering by line, and `false` implies the stream is fully unbuffered. Smaller buffering is useful if another process or the user is waiting for the output as it is being produced. See also `flush_output/[0,1]`. This option is not an ISO option.

close_on_abort(*Bool*)
If `true` (default), the stream is closed on an abort (see `abort/0`). If `false`, the stream is not closed. If it is an output stream, it will be flushed however. Useful for logfiles and if the stream is associated to a process (using the `pipe/1` construct).

lock(*LockingMode*)
Try to obtain a lock on the open file. Default is `none`, which does not lock the file. The value `read` or `shared` means other processes may read the file, but not write it. The value `write` or `exclusive` means no other process may read or write the file.

Locks are acquired through the POSIX function fcntl() using the command `F_SETLKW`, which makes a blocked call wait for the lock to be released. Please note that fcntl() locks are *advisory* and therefore only other applications using the same advisory locks honour your lock. As there are many issues around locking in Unix, especially related to NFS (network file system), please study the fcntl() manual page before trusting your locks!

The `lock` option is a SWI-Prolog extension.

wait(*Bool*)
This option can be combined with the `lock` option. If `false` (default `true`), the open call returns immediately with an exception if the file is locked. The exception has the format `permission_error`(*lock, source_sink, SrcDest*).

The option `reposition` is not supported in SWI-Prolog. All streams connected to a file may be repositioned.

open(+*SrcDest*, +*Mode*, ?*Stream***)** *[ISO]*
Equivalent to `open/4` with an empty option-list.

open_null_stream(*?Stream*)

Open an output stream that produces no output. All counting functions are enabled on such a stream. It can be used to discard output (like Unix `/dev/null`) or exploit the counting properties. The initial encoding of *Stream* is `utf8`, enabling arbitrary Unicode output. The encoding can be changed to determine byte-counts of the output in a particular encoding or validate output is possible in a particular encoding. For example, the code below determines the number of characters emitted when writing *Term*.

```
write_length(Term, Len) :-
        open_null_stream(Out),
        write(Out, Term),
        character_count(Out, Len0),
        close(Out),
        Len = Len0.
```

close(*+Stream*) *[ISO]*

Close the specified stream. If *Stream* is not open, an existence error is raised. However, closing a stream multiple times may crash Prolog. This is particularly true for multi-threaded applications.

If the closed stream is the current input or output stream, the terminal is made the current input or output.

close(*+Stream, +Options*) *[ISO]*

Provides `close`(*Stream, [force(true)]*) as the only option. Called this way, any resource error (such as write-errors while flushing the output buffer) are ignored.

stream_property(*?Stream, ?StreamProperty*) *[ISO]*

ISO compatible predicate for querying status of open I/O streams. *StreamProperty* is one of:

alias(*Atom*)

If *Atom* is bound, test of the stream has the specified alias. Otherwise unify *Atom* with the first alias of the stream.[35]

buffer(*Buffering*)

SWI-Prolog extension to query the buffering mode of this stream. *Buffering* is one of `full`, `line` or `false`. See also `open/4`.

buffer_size(*Integer*)

SWI-Prolog extension to query the size of the I/O buffer associated to a stream in bytes. Fails of the stream is not buffered.

bom(*Bool*)

If present and `true`, a BOM (*Byte Order Mark*) was detected while opening the file for reading or a BOM was written while opening the stream. See section 2.18.1 for details.

close

bom(

close_on_exec(*Bool*)
Determine whether or not the stream is closed when executing a new process (exec() in Unix, CreateProcess() in Windows). Default is to close streams. This maps to fcntl() `F_SETFD` using the flag `FD_CLOEXEC` on Unix and (negated) `HANDLE_FLAG_INHERIT` on Windows.

encoding(*Encoding*)
Query the encoding used for text. See section 2.18.1 for an overview of wide character and encoding issues in SWI-Prolog.

end_of_stream(*E*)
If *Stream* is an input stream, unify *E* with one of the atoms `not`, `at` or `past`. See also `at_end_of_stream/[0,1]`.

eof_action(*A*)
Unify *A* with one of `eof_code`, `reset` or `error`. See `open/4` for details.

file_name(*Atom*)
If *Stream* is associated to a file, unify *Atom* to the name of this file.

file_no(*Integer*)
If the stream is associated with a POSIX file-descriptor, unify *Integer* with the descriptor number. SWI-Prolog extension used primarily for integration with foreign code. See also Sfileno() from `SWI-Stream.h`.

input
True if *Stream* has mode `read`.

mode(*IOMode*)
Unify *IOMode* to the mode given to `open/4` for opening the stream. Values are: `read`, `write`, `append` and the SWI-Prolog extension `update`.

newline(*NewlineMode*)
One of `posix` or `dos`. If `dos`, text-streams will emit `\r\n` for `\n` and discard `\r` from input streams. Default depends on the operating system.

nlink(*-Count*)
Number of hard links to the file. This expresses the number of 'names' the file has. Not supported on all operating systems and the value might be bogus. See the documentation of fstat() for your OS and the value `st_nlink`.

output
True if *Stream* has mode `write`, `append` or `update`.

position(*Term*)
Unify *Term* with the current stream-position. A stream-position is an opaque term whose fields can be extracted using `stream_position_data/3`. See also `set_stream_position/2`.

reposition(*Bool*)
Unify *Bool* with *true* if the position of the stream can be set (see `seek/4`). It is assumed the position can be set if the stream has a *seek-function* and is not based on a POSIX file-descriptor that is not associated to a regular file.

representation_errors(*Mode*)
Determines behaviour of character output if the stream cannot represent a character. For

example, an ISO Latin-1 stream cannot represent Cyrillic characters. The behaviour is one of `error` (throw and I/O error exception), `prolog` (write `\...\` escape code or `xml` (write `&#...;` XML character entity). The initial mode is `prolog` for the user streams and `error` for all other streams. See also section 2.18.1 and `set_stream/2`.

timeout(*-Time***)**

Time is the timeout currently associated with the stream. See `set_stream/2` with the same option. If no timeout is specified, *Time* is unified to the atom `infinite`.

type(*T***)**

Unify *Bool* with `text` or `binary`.

tty(*Bool***)**

This property is reported with *Bool* equals `true` if the stream is associated with a terminal. See also `set_stream/2`.

current_stream(*?Object, ?Mode, ?Stream***)**

The predicate `current_stream/3` is used to access the status of a stream as well as to generate all open streams. *Object* is the name of the file opened if the stream refers to an open file, an integer file-descriptor if the stream encapsulates an operating-system stream or the atom `[]` if the stream refers to some other object. *Mode* is one of `read` or `write`.

is_stream(*+Term***)**

True if *Term* is a stream name or valid stream handle. This predicate realises a safe test for the existence of a stream alias or handle.

stream_pair(*?StreamPair, ?Read, ?Write***)**

This predicate can be used in mode (-,+,+) to create a *stream-pair* from an input stream and an output stream. Stream-pairs can be used by all I/O operations on streams, where the operation selects the appropriate member of the pair. The predicate `close/1` closes both streams of the pair. Mode (+,-,-) can be used to get access to the underlying streams.

set_stream_position(*+Stream, +Pos***)** *[ISO]*

Set the current position of *Stream* to *Pos*. *Pos* is a term as returned by `stream_property/2` using the `position`(*Pos*) property. See also `seek/4`.

stream_position_data(*?Field, +Position, -Data***)**

Extracts information from the opaque stream position term as returned by `stream_property/2` requesting the `position`(*Position*) property. *Field* is one of `line_count`, `line_position`, `char_count` or `byte_count`. See also `line_count/2`, `line_position/2`, `character_count/2` and `byte_count/2`.[36]

seek(*+Stream, +Offset, +Method, -NewLocation***)**

Reposition the current point of the given *Stream*. *Method* is one of `bof`, `current` or `eof`, indicating positioning relative to the start, current point or end of the underlying object. *NewLocation* is unified with the new offset, relative to the start of the stream.

Positions are counted in 'units'. A unit is 1 byte, except for text-files using 2-byte Unicode encoding (2 bytes) or *wchar* encoding (sizeof(wchar_t)). The latter guarantees comfortable interaction with wide-character text-objects. Otherwise, the use of `seek/4` on

---

[36] Introduced in version 5.6.4 after extending the position term with a byte-count. Compatible with SICStus Prolog.

non-binary files (see `open/4`) is of limited use, especially when using multi-byte text-encodings (e.g. UTF-8) or multi-byte newline files (e.g. DOS/Windows). On text-files, SWI-Prolog offers reliable backup to an old position using `stream_property/2` and `set_stream_position/2`. Skipping *N* character codes is achieved calling `get_code/2` *N* times or using `copy_stream_data/3`, directing the output to a null-stream (see `open_null_stream/1`). If the seek modifies the current location, the line number and character position in the line are set to 0.

If the stream cannot be repositioned, a `permission_error` is raised. If applying the offset would result in a file-position less then zero, a `domain_error` is raised. Behaviour when seeking to positions beyond the size of the underlying object depend on the object and possibly the operating system. The predicate `seek/4` is compatible with Quintus Prolog, though the error conditions and signalling is ISO compliant. See also `stream_property/2` and `set_stream_position/2`.

set_stream(+*Stream*, +*Attribute***)**

Modify an attribute of an existing stream. *Attribute* specifies the stream property to set. See also `stream_property/2` and `open/4`.

alias(*AliasName***)**

Set the alias of an already created stream. If *AliasName* is the name of one of the standard streams is used, this stream is rebound. Thus, `set_stream(S, current_input)` is the same as `set_input/1` and by setting the alias of a stream to `user_input`, etc. all user terminal input is read from this stream. See also `interactor/0`.

buffer(*Buffering***)**

Set the buffering mode of an already created stream. Buffering is one of `full`, `line` or `false`.

buffer_size(+*Size***)**

Set the size of the I/O buffer of the underlying stream to *Size* bytes.

close_on_abort(*Bool***)**

Determine whether or not the stream is closed by `abort/0`. By default streams are closed.

close_on_exec(*Bool***)**

Set the `close_on_exec` property. See `stream_property/2`.

encoding(*Atom***)**

Defines the mapping between bytes and character codes used for the stream. See section 2.18.1 for supported encodings.

eof_action(*Action***)**

Set end-of-file handling to one of `eof_code`, `reset` or `error`.

file_name(*FileName***)**

Set the file name associated to this stream. This call can be used to set the file for error-locations if *Stream* corresponds to *FileName* and is not obtained by opening the file directly but, for example, through a network service.

line_position(*LinePos***)**

Set the line-position attribute of the stream. This feature is intended to correct position management of the stream after sending a terminal escape sequence (e.g., setting ANSI character attributes). Setting this attribute raises a permission error if the stream does not record positions. See `line_position/2` and `stream_property/2` (property `position`).

newline(*NewlineMode*)
Set input or output translation for newlines. See corresponding `stream_property/2` for details. In addition to the detected modes, an input stream can be set in mode `detect`. It will be set to `dos` if a `\r` character was removed.

timeout(*Seconds*)
This option can be used to make streams generate an exception if it takes longer than *Seconds* before any new data arrives at the stream. The value *infinite* (default) makes the stream block indefinitely. Like `wait_for_input/3`, this call only applies to streams that support the select() system call. For further information about timeout handling, see `wait_for_input/3`. The exception is of the form

`error(timeout_error`*(read, Stream), _*)

type(*Type*)
Set the type of the stream to one of `text` or `binary`. See also `open/4` and the `encoding` property of streams.

record_position(*Bool*)
Do/do not record the line-count and line-position (see `line_count/2` and `line_position/2`).

representation_errors(*Mode*)
Change the behaviour when writing characters to the stream that cannot be represented by the encoding. See also `stream_property/2` and section 2.18.1.

tty(*Bool*)
Modify whether Prolog thinks there is a terminal (i.e. human interaction) connected to this stream. On Unix systems the initial value comes from isatty(). On Windows, the initial user streams are supposed to be associated to a terminal. See also `stream_property/2`.

set_prolog_IO(+*In*, +*Out*, +*Error*)
Prepare the given streams for interactive behaviour normally associated to the terminal. *In* becomes the `user_input` and `current_input` of the calling thread. *Out* becomes `user_output` and `current_output`. If *Error* equals *Out* an unbuffered stream is associated to the same destination and linked to `user_error`. Otherwise *Error* is used for `user_error`. Output buffering for *Out* is set to `line` and buffering on *Error* is disabled. See also `prolog/0` and `set_stream/2`. The *clib* package provides the library `prolog_server` creating a TCP/IP server for creating an interactive session to Prolog.

4.16.2 Edinburgh-style I/O

The package for implicit input and output destination is (almost) compatible with Edinburgh DEC-10 and C-Prolog. The reading and writing predicates refer to resp. the *current* input- and output stream. Initially these streams are connected to the terminal. The current output stream is changed using `tell/1` or `append/1`. The current input stream is changed using `see/1`. The streams current value can be obtained using `telling/1` for output- and `seeing/1` for input streams.

Source and destination are either a file, `user`, or a term 'pipe(*Command*)'. The reserved stream name `user` refers to the terminal.[37] In the predicate descriptions below we will call the source/destination argument '*SrcDest*'. Below are some examples of source/destination specifications.

`?- see(data).`	% Start reading from file 'data'.
`?- tell(user).`	% Start writing to the terminal.
`?- tell(pipe(lpr)).`	% Start writing to the printer.

Another example of using the `pipe/1` construct is shown below.[38] Note that the `pipe/1` construct is not part of Prolog's standard I/O repertoire.

```
getwd(Wd) :-
        seeing(Old), see(pipe(pwd)),
        collect_wd(String),
        seen, see(Old),
        atom_codes(Wd, String).

collect_wd([C|R]) :-
        get0(C), C \== -1, !,
        collect_wd(R).
collect_wd([]).
```

The effect of `tell/1` is not undone on backtracking, and since the stream handle is not specified explicitly in further I/O operations when using Edinburgh-style I/O, you may write to unintended streams more easily than when using ISO compliant I/O. For example, the following query writes both "a" and "b" into the file 'out' :

```
?- (tell(out), write(a), false ; write(b)), told.
```

Compatibility notes

Unlike Edinburgh Prolog systems, `telling/1` and `seeing/1` do not return the filename of the current input/output, but the stream-identifier, to ensure the design pattern below works under all circumstances.[39]

```
        ...,
        telling(Old), tell(x),
        ...,
        told, tell(Old),
        ...,
```

---

[37] The ISO I/O layer uses `user_input`, `user_output` and `user_error`.

[38] As of version 5.3.15, the pipe construct is supported in the MS-Windows version, both for `swipl.exe` and `swipl-win.exe`. The implementation uses code from the LUA programming language (`http://www.lua.org`).

[39] Filenames can be ambiguous and SWI-Prolog streams can refer to much more than just files.

The predicates `tell/1` and `see/1` first check for `user`, the `pipe`(*command*) and a stream-handle. Otherwise, if the argument is an atom it is first compared to open streams associated to a file with *exactly* the same name. If such a stream, created using `tell/1` or `see/1` exists, output (input) is switch to the open stream. Otherwise a file with the specified name is opened.

The behaviour is compatible with Edinburgh Prolog. This is not without problems. Changing directory, non-file streams, multiple names referring to the same file easily lead to unexpected behaviour. New code, especially when managing multiple I/O channels should consider using the ISO I/O predicates defined in section 4.16.1.

see(+*SrcDest***)**

Open *SrcDest* for reading and make it the current input (see `set_input/1`). If *SrcDest* is a stream-handle, just makes this stream the current input. See the introduction of section 4.16.2 for details.

tell(+*SrcDest***)**

Open *SrcDest* for writing and make it the current output (see `set_output/1`). If *SrcDest* is a stream-handle, just makes this stream the current output. See the introduction of section 4.16.2 for details.

append(+*File***)**

Similar to `tell/1`, but positions the file pointer at the end of *File* rather than truncating an existing file. The pipe construct is not accepted by this predicate.

seeing(?*SrcDest***)**

Same as `current_input/1`, except that `user` is returned if the current input is the stream `user_input` to improve compatibility with traditional Edinburgh I/O. See the introduction of section 4.16.2 for details.

telling(?*SrcDest***)**

Same as `current_output/1`, except that `user` is returned if the current output is the stream `user_output` to improve compatibility with traditional Edinburgh I/O. See the introduction of section 4.16.2 for details.

seen

Close the current input stream. The new input stream becomes *user_input*.

told

Close the current output stream. The new output stream becomes *user_output*.

4.16.3 Switching Between Edinburgh and ISO I/O

The predicates below can be used for switching between the implicit- and the explicit stream based I/O predicates.

set_input(+*Stream***)** *[ISO]*

Set the current input stream to become *Stream*. Thus, open(file, read, Stream), set_input(Stream) is equivalent to see(file).

set_output(+*Stream***)** *[ISO]*

Set the current output stream to become *Stream*. See also `with_output_to/2`.

current_input(-*Stream***)** *[ISO]*

Get the current input stream. Useful to get access to the status predicates associated with streams.

current_output(-*Stream***)** *[ISO]*

Get the current output stream.

4.16.4 Write onto atoms, code-lists, etc.

with_output_to(+*Output, :Goal***)**

Run *Goal* as `once/1`, while characters written to the current output is sent to *Output*. The predicate is SWI-Prolog specific, inspired by various posts to the mailinglist. It provides a flexible replacement for predicates such as `sformat/3`, `swritef/3`, `term_to_atom/2`, `atom_number/2` converting numbers to atoms, etc. The predicate `format/3` accepts the same terms as output argument.

Applications should generally avoid creating atoms by breaking and concatenating other atoms as the creation of large numbers of intermediate atoms generally leads to poor performance, even more so in multi-threaded applications. This predicate supports creating difference-lists from character data efficiently. The example below defines the DCG rule `term//1` to insert a term in the output:

```
term(Term, In, Tail) :-
        with_output_to(codes(In, Tail), write(Term)).

?- phrase(term(hello), X).

X = [104, 101, 108, 108, 111]
```

A Stream handle or alias

Temporary switch current output to the given stream. Redirection using `with_output_to/2` guarantees the original output is restored, also if *Goal* fails or raises an exception. See also `call_cleanup/2`.

atom(-*Atom***)**

Create an atom from the emitted characters. Please note the remark above.

string(-*String***)**

Create a string-object as defined in section 4.23.

codes(-*Codes***)**

Create a list of character codes from the emitted characters, similar to `atom_codes/2`.

codes(-*Codes, -Tail***)**

Create a list of character codes as a difference-list.

chars(-*Chars***)**

Create a list of one-character-atoms codes from the emitted characters, similar to `atom_chars/2`.

chars(-*Chars, -Tail***)**

Create a list of one-character-atoms as a difference-list.

4.17 Status of streams

wait_for_input(+*ListOfStreams, -ReadyList, +TimeOut***)**

Wait for input on one of the streams in *ListOfStreams* and return a list of streams on which input is available in *ReadyList*. `wait_for_input/3` waits for at most *TimeOut* seconds. *Timeout* may be specified as a floating point number to specify fractions of a second. If *Timeout* equals `infinite`, `wait_for_input/3` waits indefinitely.[40]

This predicate can be used to implement timeout while reading and to handle input from multiple sources. The following example will wait for input from the user and an explicitly opened second terminal. On return, *Inputs* may hold `user_input` or *P4* or both.

```
?- open('/dev/ttyp4', read, P4),
   wait_for_input([user_input, P4], Inputs, 0).
```

This predicate relies on the select() call on most operating systems. On Unix this call is implemented for any stream referring to a file-handle, which implies all OS-based streams: sockets, terminals, pipes, etc. On non-Unix systems select() is generally only implemented for socket-based streams. See also `socket` from the `clib` package.

Note that `wait_for_input/3` returns streams that have data waiting. This does not mean you can, for example, call `read/2` on the stream without blocking as the stream might hold an incomplete term. The predicate `set_stream/2` using the option `timeout`(*Seconds*) can be used to make the stream generate an exception if no new data arrives for within the timeout. Suppose two processes communicate by exchanging Prolog terms. The following code makes the server immune for clients that write an incomplete term:

```
        ...,
        tcp_accept(Server, Socket, _Peer),
        tcp_open(Socket, In, Out),
        set_stream(In, timeout(10)),
        catch(read(In, Term), _, (close(Out), close(In), fail)),
        ...,
```

byte_count(+*Stream, -Count***)**

Byte-position in *Stream*. For binary streams this is the same as `character_count/2`. For text files the number may be different due to multi-byte encodings or additional record separators (such as Control-M in Windows).

character_count(+*Stream, -Count***)**

Unify *Count* with the current character index. For input streams this is the number of characters read since the open, for output streams this is the number of characters written. Counting starts at 0.

line_count(+*Stream, -Count***)**

Unify *Count* with the number of lines read or written. Counting starts at 1.

---

[40] For compatibility reasons, a *Timeout* value of 0 (integer) also waits indefinitely. To call select() without giving up the CPU pass the float 0.0. If compatibility with versions older than 5.1.3 is desired pass the float value 1.0e-7.

line_position(+*Stream, -Count***)**

Unify *Count* with the position on the current line. Note that this assumes the position is 0 after the open. Tabs are assumed to be defined on each 8-th character and backspaces are assumed to reduce the count by one, provided it is positive.

4.18 Primitive character I/O

See section 4.2 for an overview of supported character representations.

nl *[ISO]*

Write a newline character to the current output stream. On Unix systems `nl/0` is equivalent to `put(10)`.

nl(+*Stream***)** *[ISO]*

Write a newline to *Stream*.

put(+*Char***)**

Write *Char* to the current output stream, *Char* is either an integer-expression evaluating to a character code or an atom of one character. Deprecated. New code should use `put_char/1` or `put_code/1`.

put(+*Stream,* +*Char***)**

Write *Char* to *Stream*. See `put/1` for details.

put_byte(+*Byte***)** *[ISO]*

Write a single byte to the output. *Byte* must be an integer between 0 and 255.

put_byte(+*Stream,* +*Byte***)** *[ISO]*

Write a single byte to a stream. *Byte* must be an integer between 0 and 255.

put_char(+*Char***)** *[ISO]*

Write a character to the current output, obeying the encoding defined for the current output stream. Note that this may raise an exception if the encoding of *Stream* cannot represent *Char*.

put_char(+*Stream,* +*Char***)** *[ISO]*

Write a character to *Stream*, obeying the encoding defined for *Stream*. Note that this may raise an exception if the encoding of *Stream* cannot represent *Char*.

put_code(+*Code***)** *[ISO]*

Similar to `put_char/1`, but using a *character code*. *Code* is a non-negative integer. Note that this may raise an exception if the encoding of *Stream* cannot represent *Code*.

put_code(+*Stream,* +*Code***)** *[ISO]*

Same as `put_code/1` but directing *Code* to *Stream*.

tab(+*Amount***)**

Writes *Amount* spaces on the current output stream. *Amount* should be an expression that evaluates to a positive integer (see section 4.26).

tab(+*Stream,* +*Amount***)**

Writes *Amount* spaces to *Stream*.

flush_output *[ISO]*

Flush pending output on current output stream. `flush_output/0` is automatically generated by `read/1` and derivatives if the current input stream is `user` and the cursor is not at the left margin.

flush_output(+*Stream***)** *[ISO]*

Flush output on the specified stream. The stream must be open for writing.

ttyflush

Flush pending output on stream *user*. See also `flush_output/[0,1]`.

get_byte(-*Byte***)** *[ISO]*

Read the current input stream and unify the next byte with *Byte* (an integer between 0 and 255. *Byte* is unified with -1 on end of file.

get_byte(+*Stream, -Byte***)** *[ISO]*

Read the next byte from *Stream*, returning an integer between 0 and 255.

get_code(-*Code***)** *[ISO]*

Read the current input stream and unify *Code* with the character code of the next character. *Code* is unified with -1 on end of file. See also `get_char/1`.

get_code(+*Stream, -Code***)** *[ISO]*

Read the next character-code from *Stream*.

get_char(-*Char***)** *[ISO]*

Read the current input stream and unify *Char* with the next character as a one-character-atom. See also `atom_chars/2`. On end-of-file, *Char* is unified to the atom `end_of_file`.

get_char(+*Stream, -Char***)** *[ISO]*

Unify *Char* with the next character from *Stream* as a one-character-atom. See also `get_char/2`, `get_byte/2` and `get_code/2`.

get0(-*Char***)** *[deprecated]*

Edinburgh version of the ISO `get_code/1` predicate. Note that Edinburgh prolog didn't support wide characters and therefore technically speaking `get0/1` should have been mapped to `get_byte/1`. The intention of `get0/1` however is to read character codes.

get0(+*Stream, -Char***)** *[deprecated]*

Edinburgh version of the ISO `get_code/2` predicate. See also `get0/1`.

get(-*Char***)** *[deprecated]*

Read the current input stream and unify the next non-blank character with *Char*. *Char* is unified with -1 on end of file. The predicate `get/1` operates on character *codes*. See also `get0/1`.

get(+*Stream, -Char***)** *[deprecated]*

Read the next non-blank character from *Stream*. See also `get/1`, `get0/1` and `get0/2`.

peek_byte(-*Byte***)** *[ISO]*

peek_byte(+*Stream, -Byte***)** *[ISO]*

peek_code(-*Code***)** *[ISO]*
peek_code(+*Stream*, -*Code***)** *[ISO]*
peek_char(-*Char***)** *[ISO]*
peek_char(+*Stream*, -*Char***)** *[ISO]*

Read the next byte/code/char from the input without removing it. These predicates do not modify the stream's position or end-of-file status. These predicates require a buffered stream (see `set_stream/2`) and raise a permission_error if the stream is unbuffered or the buffer is too small to hold the longest multibyte sequence that might need to be buffered.

skip(+*Code***)**

Read the input until *Char* or the end of the file is encountered. A subsequent call to `get_code/1` will read the first character after *Code*.

skip(+*Stream*, +*Code***)**

Skip input (as `skip/1`) on *Stream*.

get_single_char(-*Code***)**

Get a single character from input stream 'user' (regardless of the current input stream). Unlike `get_code/1` this predicate does not wait for a return. The character is not echoed to the user's terminal. This predicate is meant for keyboard menu selection etc. If SWI-Prolog was started with the `-tty` option this predicate reads an entire line of input and returns the first non-blank character on this line, or the character code of the newline (10) if the entire line consisted of blank characters.

at_end_of_stream *[ISO]*

Succeeds after the last character of the current input stream has been read. Also succeeds if there is no valid current input stream.

at_end_of_stream(+*Stream***)** *[ISO]*

Succeeds after the last character of the named stream is read, or *Stream* is not a valid input stream. The end-of-stream test is only available on buffered input stream (unbuffered input streams are rarely used, see `open/4`).

set_end_of_stream(+*Stream***)**

Sets the size of the file opened as *Stream* to the current file-position. This is typically used in combination with the open-mode `update`.

copy_stream_data(+*StreamIn*, +*StreamOut*, +*Len***)**

Copy *Len* codes from stream *StreamIn* to *StreamOut*. Note that the copy is done using the semantics of `get_code/2` and `put_code/2`, taking care of possibly recoding that needs take place between two text files. See section 2.18.1.

copy_stream_data(+*StreamIn*, +*StreamOut***)**

Copy data all (remaining) data from stream *StreamIn* to *StreamOut*.

read_pending_input(+*StreamIn*, -*Codes*, ?*Tail***)**

Read input pending in the input buffer of *StreamIn* and return it in the difference list *Codes-Tail*. I.e. the available characters codes are used to create the list *Codes* ending in the tail *Tail*. This predicate is intended for efficient unbuffered copying and filtering of input coming from network connections or devices.

The following code fragment realises efficient non-blocking copy of data from an input- to an output stream. The `at_end_of_stream/1` call checks for end-of-stream and fills the input buffer. Note that the use of a `get_code/2` and `put_code/2` based loop requires a `flush_output/1` call after *each* `put_code/2`. The `copy_stream_data/2` does not allow for inspection of the copied data and suffers from the same buffering issues.

```
copy(In, Out) :-
        repeat,
            (   at_end_of_stream(In)
            ->  !
            ;   read_pending_input(In, Chars, []),
                format(Out, '~s', [Chars]),
                flush_output(Out),
                fail
            ).
```

4.19 Term reading and writing

This section describes the basic term reading and writing predicates. The predicates `format/[1,2]` and `writef/2` provide formatted output. Writing to Prolog datastructures such as atoms or code-lists is supported by `with_output_to/2` and `format/3`.

Reading is sensitive to the Prolog flag `character_escapes`, which controls the interpretation of the `\` character in quoted atoms and strings.

write_term(+*Term*, +*Options***)** *[ISO]*

The predicate `write_term/2` is the generic form of all Prolog term-write predicates. Valid options are:

attributes(*Atom***)**

Define how attributed variables (see section 6.1) are written. The default is determined by the Prolog flag `write_attributes`. Defined values are `ignore` (ignore the attribute), `dots` (write the attributes as `{...}`), `write` (simply hand the attributes recursively to `write_term/2`) and `portray` (hand the attributes to `attr_portray_hook/2`).

backquoted_string(*Bool***)**

If `true`, write a string object (see section 4.23) as `` `...` ``. The default depends on the Prolog flag `backquoted_string`.

blobs(*Atom***)**

Define how non-text blobs are handled. By default, this is left to the write-handler specified with the blob-type. Using `portray`, `portray/1` is called for each blob encountered. See section 9.4.7.

character_escapes(*Bool***)**

If `true`, and `quoted`(*true*) is active, special characters in quoted atoms and strings are emitted as ISO escape-sequences. Default is taken from the reference module (see below).

cycles(*Bool*)

If `true` (default), cyclic terms are written as `@`(*Template, Subsitutions*), where *Subsitutions* is a list *Var* = *Value*. If `cycles` is `false`, `max_depth` is not given and *Term* is cyclic, `write_term/2` raises a `domain_error`.[41] See also the `cycles` option in `read_term/2`.

ignore_ops(*Bool*)

If `true`, the generic term-representation (⟨*functor*⟩(⟨*args*⟩ ...)) will be used for all terms, Otherwise (default), operators, list-notation and `{}`/1 will be written using their special syntax.

max_depth(*Integer*)

If the term is nested deeper than *Integer*, print the remainder as ellipses (...). A 0 (zero) value (default) imposes no depth limit. This option also delimits the number of printed for a list. Example:

```
?- write_term(a(s(s(s(s(0)))), [a,b,c,d,e,f]), [max_depth(3)]).
a(s(s(...)), [a, b|...])

Yes
```

Used by the top-level and debugger to limit screen output. See also the Prolog flags `toplevel_print_options` and `debugger_print_options`.

module(*Module*)

Define the reference module (default `user`). This defines the default value for the `character_escapes` option as well as the operator definitions to use. See also `op/3`.

numbervars(*Bool*)

If `true`, terms of the format `$VAR(N)`, where ⟨*N*⟩ is a positive integer, will be written as a variable name. If *N* is an atom it is written without quotes. This extension allows for writing variables with user-provided names. The default is `false`. See also `numbervars/3`.

partial(*Bool*)

If `true` (default `false`), do not reset the logic that inserts extra spaces that separate tokens where needed. This is intended to solve the problems with the code below. Calling `write_value(.)` writes `..`, which cannot be read. By adding `partial`(*true*) to the option, it correctly emits `. ..`. Similar problems appear when emitting operators using multiple calls to `write_term/3`.

```
write_value(Value) :-
        write_term(Value, [quoted(true)]),
        write('.'), nl.
```

portray(*Bool*)

If `true`, the hook `portray/1` is called before printing a term that is not a variable. If `portray/1` succeeds, the term is considered printed. See also `print/1`. The default is `false`. This option is an extension to the ISO write_term options.

---

[41] The cycles option and the cyclic term representation using the @-term are copied from SICStus Prolog. However, the default in SICStus is set to `false` and SICStus writes an infinite term if not protected by e.g., the `depth_limit` option.

portray_goal(*:Goal*)

Implies `portray`(*true*), but calls *Goal* rather than the predefined hook `portray/1`. *Goal* is called through `call/3`, where the first argument is *Goal*, the second is the term to be printed and the 3th argument is the current write-option list. The write option list is copied from the write_term call, but the list is guaranteed to hold an option `proprity` that reflects the current priority.

priority(*Integer*)

An integer between 0 and 1200 representing the 'context priority'. Default is 1200. Can be used to write partial terms appearing as the argument to an operator. For example:

```
        format('~w = ', [VarName]),
        write_term(Value, [quoted(true), priority(699)])
```

quoted(*Bool*)

If `true`, atoms and functors that needs quotes will be quoted. The default is `false`.

spacing(*+Spacing*)

Determines whether and where extra white-space is added to enhance readability. The default is `standard`, adding only space where needed for proper tokenization by `read_term/3`. Currently, the only other value is `next_argument`, adding a space after a comma used to separate arguments in a term or list.

write_term(*+Stream, +Term, +Options*) *[ISO]*

As `write_term/2`, but output is sent to *Stream* rather than the current output.

write_length(*+Term, -Length, +Options*) *[semidet]*

True when *Length* is the number of characters emited for *write_term*Term, Options. In addition to valid options for `write_term/2`, it processes the option

max_length(*+MaxLength*)

If provided, fail if *Length* would be larger than *MaxLength*. The implementation ensures that the runtime is limited when computing the length of a huge term with a bounded maximum.

write_canonical(*+Term*) *[ISO]*

Write *Term* on the current output stream using standard parenthesised prefix notation (i.e., ignoring operator declarations). Atoms that need quotes are quoted. Terms written with this predicate can always be read back, regardless of current operator declarations. Equivalent to `write_term/2` using the options `ignore_ops`, `quoted` and `numbervars` after `numbervars/4` using the `singletons` option.

Note that due to the use of `numbervars/4`, non-ground terms must be written using a *single* `write_canonical/1` call. This used to be the case anyhow, as garbage collection between multiple calls to one of the write predicates can change the `_G`⟨*NNN*⟩ identity of the variables.

write_canonical(*+Stream, +Term*) *[ISO]*

Write *Term* in canonical form on *Stream*.

write(*+Term*) *[ISO]*

Write *Term* to the current output, using brackets and operators where appropriate.

write(+*Stream*, +*Term***)** *[ISO]*

Write *Term* to *Stream*.

writeq(+*Term***)** *[ISO]*

Write *Term* to the current output, using brackets and operators where appropriate. Atoms that need quotes are quoted. Terms written with this predicate can be read back with `read/1` provided the currently active operator declarations are identical.

writeq(+*Stream*, +*Term***)** *[ISO]*

Write *Term* to *Stream*, inserting quotes.

print(+*Term***)**

Prints *Term* on the current output stream similar to `write/1`, but for each (sub)term of *Term* first the dynamic predicate `portray/1` is called. If this predicate succeeds *print* assumes the (sub)term has been written. This allows for user defined term writing. See also `portray_text`.

print(+*Stream*, +*Term***)**

Print *Term* to *Stream*.

portray(+*Term***)**

A dynamic predicate, which can be defined by the user to change the behaviour of `print/1` on (sub)terms. For each subterm encountered that is not a variable `print/1` first calls `portray/1` using the term as argument. For lists only the list as a whole is given to `portray/1`. If portray succeeds `print/1` assumes the term has been written.

read(-*Term***)** *[ISO]*

Read the next Prolog term from the current input stream and unify it with *Term*. On a syntax error `read/1` displays an error message, attempts to skip the erroneous term and fails. On reaching end-of-file *Term* is unified with the atom `end_of_file`.

read(+*Stream*, -*Term***)** *[ISO]*

Read *Term* from *Stream*.

read_clause(-*Term***)**

Equivalent to `read/1`, but warns the user for variables only occurring once in a term (singleton variables, see section 2.15.1) which do not start with an underscore if `style_check(singleton)` is active (default). Used to read Prolog source files (see `consult/1`). New code should use `read_term/2` with the option `singletons(warning)`.

read_clause(+*Stream*, -*Term***)**

Read a clause from *Stream*. See `read_clause/1`.

read_term(-*Term*, +*Options***)** *[ISO]*

Read a term from the current input stream and unify the term with *Term*. The reading is controlled by options from the list of *Options*. If this list is empty, the behaviour is the same as for `read/1`. The options are upward compatible with Quintus Prolog. The argument order is according to the ISO standard. Syntax-errors are always reported using exception-handling (see `catch/3`). Options:

backquoted_string(*Bool***)**
If `true`, read `` `...` `` to a string object (see section 4.23). The default depends on the Prolog flag `backquoted_string`.

character_escapes(*Bool***)**
Defines how to read \ escape-sequences in quoted atoms. See the Prolog flags `character_escapes`, `current_prolog_flag/2`. (SWI-Prolog).

comments(*-Comments***)**
Unify *Comments* with a list of *Position-Comment*, where *Position* is a stream-position object (see `stream_position_data/3`) indicating the start of a comment and *Comment* is a string-object containing the text including delimiters of a comment. It returns all comments from where the `read_term/2` call started up to the end of the term read.

cycles(*Bool***)**
If `true` (default `false`), re-instantiate templates as produced by the corresponding `write_term/2` option. Note that the default is `false` to avoid mis-interpretation of `@`(*Template, Substutions*), while the default of `write_term/2` is `true` because emitting cyclic terms without using the template construct produces an infinitely large term (read: it will generate an error after producing a huge amount of output).

double_quotes(*Atom***)**
Defines how to read ”...” strings. See the Prolog flag `double_quotes`. (SWI-Prolog).

module(*Module***)**
Specify *Module* for operators, `character_escapes` flag and `double_quotes` flag. The value of the latter two is overruled if the corresponding `read_term/3` option is provided. If no module is specified, the current ‘source-module’ is used. (SWI-Prolog).

singletons(*Vars***)**
As `variable_names`, but only reports the variables occurring only once in the *Term* read. Variables starting with an underscore (‘_’) are not included in this list. (ISO). If *Vars* is the constant `warning`, singleton variables are reported using `print_message/2`.

syntax_errors(*Atom***)**
If `error` (default), throw and exception on a syntax error. Other values are `fail`, which causes a message to be printed using `print_message/2`, after which the predicate fails, `quiet` which causes the predicate to fail silently and `dec10` which causes syntax errors to be printed, after which `read_term/[2,3]` continues reading the next term. Using `dec10`, `read_term/[2,3]` never fails. (Quintus, SICStus).

subterm_positions(*TermPos***)**
Describes the detailed layout of the term. The formats for the various types of terms is given below. All positions are character positions. If the input is related to a normal stream, these positions are relative to the start of the input, when reading from the terminal, they are relative to the start of the term.

From-*To*
Used for primitive types (atoms, numbers, variables).

string_position(*From, To***)**
Used to indicate the position of a string enclosed in double quotes (`"`).

brace_term_position(*From, To, Arg***)**
Term of the form `{...}`, as used in DCG rules. *Arg* describes the argument.

list_position(*From, To, Elms, Tail***)**
A list. *Elms* describes the positions of the elements. If the list specifies the tail as |⟨*TailTerm*⟩, *Tail* is unified with the term-position of the tail, otherwise with the atom `none`.

term_position(*From, To, FFrom, FTo, SubPos***)**
Used for a compound term not matching one of the above. *FFrom* and *FTo* describe the position of the functor. *SubPos* is a list, each element of which describes the term-position of the corresponding subterm.

term_position(*Pos***)**
Unifies *Pos* with the starting position of the term read. *Pos* if of the same format as use by `stream_property/2`.

variables(*Vars***)**
Unify *Vars* with a list of variables in the term. The variables appear in the order they have been read. See also `term_variables/2`. (ISO).

variable_names(*Vars***)**
Unify *Vars* with a list of '*Name* = *Var*', where *Name* is an atom describing the variable name and *Var* is a variable that shares with the corresponding variable in *Term*. (ISO).

read_term(*+Stream, -Term, +Options***)** *[ISO]*
Read term with options from *Stream*. See `read_term/2`.

read_history(*+Show, +Help, +Special, +Prompt, -Term, -Bindings***)**
Similar to `read_term/2` using the option `variable_names`, but allows for history substitutions. `read_history/6` is used by the top level to read the user's actions. *Show* is the command the user should type to show the saved events. *Help* is the command to get an overview of the capabilities. *Special* is a list of commands that are not saved in the history. *Prompt* is the first prompt given. Continuation prompts for more lines are determined by `prompt/2`. A `%w` in the prompt is substituted by the event number. See section 2.7 for available substitutions.

SWI-Prolog calls `read_history/6` as follows:

```
read_history(h, '!h', [trace], '%w ?- ', Goal, Bindings)
```

prompt(*-Old, +New***)**
Set prompt associated with `read/1` and its derivatives. *Old* is first unified with the current prompt. On success the prompt will be set to *New* if this is an atom. Otherwise an error message is displayed. A prompt is printed if one of the read predicates is called and the cursor is at the left margin. It is also printed whenever a newline is given and the term has not been terminated. Prompts are only printed when the current input stream is *user*.

prompt1(*+Prompt***)**
Sets the prompt for the next line to be read. Continuation lines will be read using the prompt defined by `prompt/2`.

4.20 Analysing and Constructing Terms

functor(*?Term, ?Name, ?Arity***)** *[ISO]*

True when *Term* is a term with functor *Name/Arity*. If Term is a variable it is unified with a new term whose arguments are all different variables (such a term is called a skeleton). If Term is atomic, Arity will be unified with the integer 0, and Name will be unified with Term. Raises `instantiation_error` if term is unbound and *Name/Arity* is insufficiently instantiated.

arg(*?Arg, +Term, ?Value***)** *[ISO]*

Term should be instantiated to a term, *Arg* to an integer between 1 and the arity of *Term*. *Value* is unified with the *Arg*-th argument of *Term*. *Arg* may also be unbound. In this case *Value* will be unified with the successive arguments of the term. On successful unification, *Arg* is unified with the argument number. Backtracking yields alternative solutions.[42] The predicate `arg/3` fails silently if $Arg = 0$ or $Arg > arity$ and raises the exception `domain_error(not_less_then_zero, Arg)` if $Arg < 0$.

?Term **=..** *?List* *[ISO]*

List is a list which head is the functor of *Term* and the remaining arguments are the arguments of the term. Each of the arguments may be a variable, but not both. This predicate is called ‘Univ’. Examples:

```
?- foo(hello, X) =.. List.

List = [foo, hello, X]

?- Term =.. [baz, foo(1)]

Term = baz(foo(1))
```

numbervars(*+Term, +Start, -End***)**

Unify the free variables of *Term* with a term `$VAR`(*N*), where *N* is the number of the variable. Counting starts at *Start*. *End* is unified with the number that should be given to the next variable. Example:

```
?- numbervars(foo(A, B, A), 0, End).
A = '$VAR'(0),
B = '$VAR'(1),
End = 2.
```

See also the `numbervars` option to `write_term/3` and `numbervars/4`.

numbervars(*+Term, +Start, -End, +Options***)**

As `numbervars/3`, but providing the following options:

functor_name(*+Atom***)**

Name of the functor to use instead of `$VAR`.

---

[42] The instantiation pattern (-, +, ?) is an extension to ‘standard’ Prolog. Some systems provide `genarg/3` that covers this pattern.

attvar(+*Action***)**
What to do if an attributed variable is encountered. Options are `skip`, which causes `numbervars/3` to ignore the attributed variable, `bind` which causes it to thread it as a normal variable and assign the next `'$VAR'`(N) term to it or (default) `error` which raises the a `type_error` exception.[43]

singletons(+*Bool***)**
If `true` (default `false`), `numbervars/4` does singleton detection. Singleton variables are unified with `'$VAR'('_')`, causing them to be printed as _ by `write_term/2` using the numbervars option. This option is exploited by `portray_clause/2` and `write_canonical/2`.[44]

var_number(@*Term*, -*VarNumber***)**
True if *Term* is numbered by `numbervars/3` and *VarNumber* is the number given to this variable. This predicate avoids the need for unification with `'$VAR'(X)` and opens the path for replacing this valid Prolog term by an internal representation that has no textual equivalent.

term_variables(+*Term*, -*List***)** *[ISO]*
Unify *List* with a list of variables, each sharing with a unique variable of *Term*.[45] The variables in *List* are ordered in order of appearance traversing *Term* depth-first and left-to-right. See also `term_variables/3`. For example:

```
?- term_variables(a(X, b(Y, X), Z), L).

L = [G367, G366, G371]
X = G367
Y = G366
Z = G371
```

term_variables(+*Term*, -*List*, ?*Tail***)**
Difference list version of `term_variables/2`. I.e. *Tail* is the tail of the variable-list *List*.

copy_term(+*In*, -*Out***)** *[ISO]*
Create a version if *In* with renamed (fresh) variables and unify it to *Out*. Attributed variables (see section 6.1) have their attributed copied. The implementation of `copy_term/2` can deal with infinite trees (cyclic terms). As pure Prolog cannot distinguish a ground term from another ground term with exactly the same structure, ground sub-terms are *shared* between *In* and *Out*. Sharing ground terms does affect `setarg/3`. SWI-Prolog provides `duplicate_term/2` to create a true copy of a term.

4.20.1 Non-logical operations on terms

Prolog is not capable to *modify* instantiated parts of a term. Lacking that capability makes that language much safer, but unfortunately there are problems that suffer severely in terms of time and/or

---

[43] This behaviour was decided after a long discussion between David Reitter, Richard O'Keefe, Bart Demoen and Tom Schrijvers.

[44] BUG: Currently this option is ignored for cyclic terms.

[45] This predicate used to be called `free_variables/2`. The name `term_variables/2` is more widely used. The old predicate is still available from the library `backcomp`.

memory usage. Always try hard to avoid the use of these primitives, but they can be a good alternative to using dynamic predicates. See also section 6.3, discussing the use of global variables.

setarg(+*Arg*, +*Term*, +*Value***)**

Extra-logical predicate. Assigns the *Arg*-th argument of the compound term *Term* with the given *Value*. The assignment is undone if backtracking brings the state back into a position before the `setarg/3` call. See also `nb_setarg/3`.

This predicate may be used for destructive assignment to terms, using them as an extra-logical storage bin. Always try hard to avoid the use of `setarg/3` as it is not supported by many Prolog systems and one has to be very careful about unexpected copying as well as unexpected not copying of terms. A good practice to improve somewhat on this situation is to make sure that terms whose arguments are subject to `setarg/3` have one unused and unshared variable in addition to the used arguments. This variable avoids unwanted sharing in e.g., `copy_term/2` and causes the term to be considered as non-ground.

nb_setarg(+*Arg*, +*Term*, +*Value***)**

Assigns the *Arg*-th argument of the compound term *Term* with the given *Value* as `setarg/3`, but on backtracking the assignment is *not* reversed. If *Value* is not atomic, it is duplicated using `duplicate_term/2`. This predicate uses the same technique as `nb_setval/2`. We therefore refer to the description of `nb_setval/2` for details on non-backtrackable assignment of terms. This predicate is compatible with GNU-Prolog `setarg`(*A,T,V,false*), removing the type-restriction on *Value*. See also `nb_linkarg/3`. Below is an example for counting the number of solutions of a goal. Note that this implementation is thread-safe, reentrant and capable of handling exceptions. Realising these features with a traditional implementation based on assert/retract or `flag/3` is much more complicated.

```
:- meta_predicate
        succeeds_n_times(0, -).

succeeds_n_times(Goal, Times) :-
        Counter = counter(0),
        (   Goal,
            arg(1, Counter, N0),
            N is N0 + 1,
            nb_setarg(1, Counter, N),
            fail
        ;   arg(1, Counter, Times)
        ).
```

nb_linkarg(+*Arg*, +*Term*, +*Value***)**

As `nb_setarg/3`, but like `nb_linkval/2` it does *not* duplicate *Value*. Use with extreme care and consult the documentation of `nb_linkval/2` before use.

duplicate_term(+*In*, -*Out***)**

Version of `copy_term/2` that also copies ground terms and therefore ensures destructive modification using `setarg/3` does not affect the copy. See also `nb_setval/2`, `nb_linkval/2`, `nb_setarg/3` and `nb_linkarg/3`.

same_term(@*T1*, @*T2***)** *[semidet]*

True if *T1* and *T2* are the equivalent and will remain the equivalent, even if `setarg/3` is used on either of them. This means *T1* and *T2* are the same variable, equivalent atomic data or a compound term allocated at the same address.

4.21 Analysing and Constructing Atoms

These predicates convert between Prolog constants and lists of character codes. The predicates `atom_codes/2`, `number_codes/2` and `name/2` behave the same when converting from a constant to a list of character codes. When converting the other way around, `atom_codes/2` will generate an atom, `number_codes/2` will generate a number or exception and `name/2` will return a number if possible and an atom otherwise.

The ISO standard defines `atom_chars/2` to describe the 'broken-up' atom as a list of one-character atoms instead of a list of codes. Up-to version 3.2.x, SWI-Prolog's `atom_chars/2` behaved, compatible with Quintus and SICStus Prolog, like atom_codes. As of 3.3.x SWI-Prolog `atom_codes/2` and `atom_chars/2` are compliant to the ISO standard.

To ease the pain of all variations in the Prolog community, all SWI-Prolog predicates behave as flexible as possible. This implies the 'list-side' accepts either a code-list or a char-list and the 'atom-side' accept all atomic types (atom, number and string).

atom_codes(*?Atom, ?String***)** *[ISO]*

Convert between an atom and a list of character codes. If *Atom* is instantiated, if will be translated into a list of character codes and the result is unified with *String*. If *Atom* is unbound and *String* is a list of character codes, it will *Atom* will be unified with an atom constructed from this list.

atom_chars(*?Atom, ?CharList***)** *[ISO]*

As `atom_codes/2`, but *CharList* is a list of one-character atoms rather than a list of character codes[46].

```
?- atom_chars(hello, X).

X = [h, e, l, l, o]
```

char_code(*?Atom, ?Code***)** *[ISO]*

Convert between character and character code for a single character.[47]

number_chars(*?Number, ?CharList***)** *[ISO]*

Similar to `atom_chars/2`, but converts between a number and its representation as a list of one-character atoms. Fails with a `syntax_error` if *Number* is unbound and *CharList* does

---

[46] Up-to version 3.2.x, `atom_chars/2` behaved as the current `atom_codes/2`. The current definition is compliant with the ISO standard

[47] This is also called `atom_char/2` in older versions of SWI-Prolog as well as some other Prolog implementations. The `atom_char/2` predicate is available from the library `backcomp.pl`

not describe a number. Following the ISO standard, it allows for *leading* white space (including newlines) and does not allow for *trailing* white space.[48]

number_codes(*?Number, ?CodeList***)** *[ISO]*

As `number_chars/2`, but converts to a list of character codes rather than one-character atoms. In the mode -, +, both predicates behave identically to improve handling of non-ISO source.

atom_number(*?Atom, ?Number***)**

Realises the popular combination of `atom_codes/2` and `number_codes/2` to convert between atom and number (integer or float) in one predicate, avoiding the intermediate list. Calling in mode +,- to convert numbers represented as atoms is often good style. Converting numbers to atoms, which in turn are assembled into larger units before communication them to the outside world is bad style. Consider using streams or `with_output_to/2` to reduce the number of expensive intermediate atoms.

name(*?AtomOrInt, ?String***)**

String is a list of character codes representing the same text as *Atom*. Each of the arguments may be a variable, but not both. When *String* is bound to an character code list describing an integer and *Atom* is a variable *Atom* will be unified with the integer value described by *String* (e.g., ‘`name(N, "300"), 400 is N + 100`’ succeeds). New code should consider using the ISO predicates `atom_codes/2` or `number_codes/2`.[49]

term_to_atom(*?Term, ?Atom***)**

True if *Atom* describes a term that unifies with *Term*. When *Atom* is instantiated *Atom* is converted and then unified with *Term*. If *Atom* has no valid syntax, a `syntax_error` exception is raised. Otherwise *Term* is “written” on *Atom* using `write_term/2` with the option `quoted`(*true*). See also `format/3` and `with_output_to/2`.

atom_to_term(*+Atom, -Term, -Bindings***)**

Use *Atom* as input to `read_term/2` using the option `variable_names` and return the read term in *Term* and the variable bindings in *Bindings*. *Bindings* is a list of $Name = Var$ couples, thus providing access to the actual variable names. See also `read_term/2`. If *Atom* has no valid syntax, a `syntax_error` exception is raised.

atom_concat(*?Atom1, ?Atom2, ?Atom3***)** *[ISO]*

Atom3 forms the concatenation of *Atom1* and *Atom2*. At least two of the arguments must be instantiated to atoms. This predicate also allows for the mode (-,-,+), non-deterministically splitting the 3-th argument into two parts (as `append/3` does for lists). SWI-Prolog allows for atomic arguments. Portable code must use `atomic_concat/3` if non-atom arguments are involved.

---

[48] ISO also allows for Prolog comments in leading white space. We -and most other implementations- believe this is incorrect. We also beleive it would have been better not to allow for white space, or to allow for both leading and trailing white space. Prolog-syntax based conversion can be achieved using `format/3` and `read_from_chars/2`.

[49] Unfortunately, the ISO predicates provide no neat way to check that a string can be interpreted as a number. The most sensible way is to use `catch/3` to catch the exception from `number_codes/2`, however this is both slow and cumbersome.

atomic_concat(+*Atomic1, +Atomic2, -Atom***)**

Atom represents the text after converting *Atomic1* and *Atomic2* to text and concatenating the result:

```
?- atomic_concat(name, 42, X).
X = name42.
```

atomic_list_concat(+*List, -Atom***)** *[commons]*

List is a list of atoms, integers or floating point numbers. Succeeds if *Atom* can be unified with the concatenated elements of *List*.

atomic_list_concat(+*List, +Separator, ?Atom***)** *[commons]*

Creates an atom just like `atomic_list_concat/2`, but inserts *Separator* between each pair of atoms. For example:

```
?- atomic_list_concat([gnu, gnat], ', ', A).

A = 'gnu, gnat'
```

The SWI-Prolog version of this predicate can also be used to split atoms by instantiating *Separator* and *Atom* as shown below. We kept this functionality to simplify porting old SWI-Prolog code where this predicate was called `concat_atom/3`.

```
?- atomic_list_concat(L, -, 'gnu-gnat').

L = [gnu, gnat]
```

atom_length(+*Atom, -Length***)** *[ISO]*

True if *Atom* is an atom of *Length* characters long. The SWI-Prolog version accepts all atomic types, as well as code-lists and character-lists. New code should avoid this feature and use `write_length/3` to get the number of characters that would be written if the argument was handed to `write_term/3`.

atom_prefix(+*Atom, +Prefix***)** *[deprecated]*

True if *Atom* starts with the characters from *Prefix*. Its behaviour is equivalent to `?- sub_atom(`*Atom*`, 0, _, _, `*Prefix*`)`. Deprecated.

sub_atom(+*Atom, ?Before, ?Len, ?After, ?Sub***)** *[ISO]*

ISO predicate for breaking atoms. It maintains the following relation: *Sub* is a sub-atom of *Atom* that starts at *Before*, has *Len* characters and *Atom* contains *After* characters after the match.

```
?- sub_atom(abc, 1, 1, A, S).

A = 1, S = b
```

The implementation minimises non-determinism and creation of atoms. This is a very flexible predicate that can do search, prefix- and suffix-matching, etc.

4.22 Character properties

SWI-Prolog offers two comprehensive predicates for classifying characters and character-codes. These predicates are defined as built-in predicates to exploit the C-character classification's handling of *locale* (handling of local character-sets). These predicates are fast, logical and deterministic if applicable.

In addition, there is the library `ctype` providing compatibility with some other Prolog systems. The predicates of this library are defined in terms of `code_type/2`.

char_type(*?Char, ?Type***)**

Tests or generates alternative *Type*s or *Char*s. The character-types are inspired by the standard C `<ctype.h>` primitives.

alnum

Char is a letter (upper- or lowercase) or digit.

alpha

Char is a letter (upper- or lowercase).

csym

Char is a letter (upper- or lowercase), digit or the underscore (_). These are valid C- and Prolog symbol characters.

csymf

Char is a letter (upper- or lowercase) or the underscore (_). These are valid first characters for C- and Prolog symbols

ascii

Char is a 7-bits ASCII character (0..127).

white

Char is a space or tab. E.i. white space inside a line.

cntrl

Char is an ASCII control-character (0..31).

digit

Char is a digit.

digit(*Weigth***)**

Char is a digit with value *Weigth*. I.e. `char_type(X, digit(6)` yields *X* = `'6'`. Useful for parsing numbers.

xdigit(*Weigth***)**

Char is a hexa-decimal digit with value *Weigth*. I.e. `char_type(a, xdigit(X)` yields *X* = `'10'`. Useful for parsing numbers.

graph

Char produces a visible mark on a page when printed. Note that the space is not included!

lower

Char is a lower-case letter.

lower(*Upper***)**

Char is a lower-case version of *Upper*. Only true if *Char* is lowercase and *Upper* uppercase.

to_lower(*Upper*)
Char is a lower-case version of *Upper*. For non-letters, or letter without case, *Char* and *Lower* are the same. See also `upcase_atom/2` and `downcase_atom/2`.

upper
Char is an upper-case letter.

upper(*Lower*)
Char is an upper-case version of *Lower*. Only true if *Char* is uppercase and *Lower* lowercase.

to_upper(*Lower*)
Char is an upper-case version of *Lower*. For non-letters, or letter without case, *Char* and *Lower* are the same. See also `upcase_atom/2` and `downcase_atom/2`.

punct
Char is a punctuation character. This is a `graph` character that is not a letter or digit.

space
Char is some form of layout character (tab, vertical-tab, newline, etc.).

end_of_file
Char is -1.

end_of_line
Char ends a line (ASCII: 10..13).

newline
Char is a the newline character (10).

period
Char counts as the end of a sentence (.,!,?).

quote
Char is a quote-character (`"`, `'`, `` ` ``).

paren(*Close*)
Char is an open-parenthesis and *Close* is the corresponding close-parenthesis.

code_type(*?Code, ?Type*)
As `char_type/2`, but uses character-codes rather than one-character atoms. Please note that both predicates are as flexible as possible. They handle either representation if the argument is instantiated and only will instantiate with an integer code or one-character atom depending of the version used. See also the Prolog flag `double_quotes`, `atom_chars/2` and `atom_codes/2`.

4.22.1 Case conversion

There is nothing in the Prolog standard for converting case in textual data. The SWI-Prolog predicates `code_type/2` and `char_type/2` can be used to test and convert individual characters. We have started some additional support:

downcase_atom(*+AnyCase, -LowerCase*)
Converts the characters of *AnyCase* into lowercase as `char_type/2` does (i.e. based on the defined *locale* if Prolog provides locale support on the hosting platform) and unifies the lowercase atom with *LowerCase*.

upcase_atom(+*AnyCase*, -*UpperCase***)**

Converts, similar to `downcase_atom/2`, an atom to upper-case.

4.22.2 White space normalization

normalize_space(-*Out*, +*In***)**

Normalize white space in *In*. All leading and trailing white space is removed. All non-empty sequences for Unicode white space characters are replaces by a single space (`\u0020`) character. *Out* uses the same conventions as `with_output_to/2` and `format/3`.

4.22.3 Language specific comparison

This section deals with predicates for language specific string comparison operations.

collation_key(+*Atom*, -*Key***)**

Create a *Key* from *Atom* for locale specific comparison. The key is defined such that if the key of atom A precedes the key of atom B in the standard order of terms, A is alphabetically smaller than B using the sort order of the current locale.

The predicate `collation_key/2` is used by `locale_sort/2` from library(sort). Please examine the implementation of `locale_sort/2` as an example of using this call.

The *Key* is an implementation defined and generally unreadable string. On systems that do not support locale-handling, *Key* is simply unified with *Atom*.

locale_sort(+*List*, -*Sorted***)**

Sort a list of atoms using the current locale. *List* is a list of atoms or string objects (see section 4.23). *Sorted* is unified with a list containing all atoms of *List*, sorted to the rules of the current locale. See also `collation_key/2` and `setlocale/3`.

4.23 Representing text in strings

SWI-Prolog supports the data type *string*. Strings are a time and space efficient mechanism to handle text in Prolog. Strings are stored as a byte array on the global (term) stack and thus destroyed on backtracking and reclaimed by the garbage collector.

Strings were added to SWI-Prolog based on an early draft of the ISO standard, offering a mechanism to represent temporary character data efficiently. As SWI-Prolog strings can handle 0-bytes, they are frequently used through the foreign language interface (section 9) for storing arbitrary byte-sequences.

Starting with version 3.3, SWI-Prolog offers garbage collection on the atom-space as well as representing 0-bytes in atoms. Although strings and atoms still have different features, new code should consider using atoms to avoid too many representations for text as well as for compatibility with other Prolog implementations. Below are some of the differences:

- *creation*

  Creating strings is fast, as the data is simply copied to the global stack. Atoms are unique and therefore more expensive in terms of memory and time to create. On the other hand, if the same text has to be represented multiple times, atoms are more efficient.

- *destruction*
  Backtracking destroys strings at no cost. They are cheap to handle by the garbage collector, but it should be noted that extensive use of strings will cause many garbage collections. Atom garbage collection is generally faster.

String objects by default have no lexical representation and thus can only be created using the predicates below or through the foreign language interface (See chapter 9. There are two ways to make `read/1` read text into strings, both controlled through Prolog flags. One is by setting the `double_quotes` flag to `string` and the other is by setting the `backquoted_string` flag to `true`. In latter case, `` `Hello world` `` is read into a string and `write_term/2` prints strings between back-quotes if `quoted` is `true`. This flag provides compatibility with LPA Prolog string handling.

string_to_atom(*?String, ?Atom***)**
Logical conversion between a string and an atom. At least one of the two arguments must be instantiated. *Atom* can also be an integer or floating point number.

string_to_list(*?String, ?List***)**
Logical conversion between a string and a list of character codes characters. At least one of the two arguments must be instantiated.

string_length(*+String, -Length***)**
Unify *Length* with the number of characters in *String*. This predicate is functionally equivalent to `atom_length/2` and also accepts atoms, integers and floats as its first argument.

string_concat(*?String1, ?String2, ?String3***)**
Similar to `atom_concat/3`, but the unbound argument will be unified with a string object rather than an atom. Also, if both *String1* and *String2* are unbound and *String3* is bound to text, it breaks *String3*, unifying the start with *String1* and the end with *String2* as append does with lists. Note that this is not particularly fast on long strings as for each redo the system has to create two entirely new strings, while the list equivalent only creates a single new list-cell and moves some pointers around.

sub_string(*+String, ?Start, ?Length, ?After, ?Sub***)**
Sub is a substring of *String* starting at *Start*, with length *Length* and *String* has *After* characters left after the match. See also `sub_atom/5`.

4.24 Operators

Operators are defined to improve the readability of source-code. For example, without operators, to write `2*3+4*5` one would have to write `+(*(2,3),*(4,5))`. In Prolog, a number of operators have been predefined. All operators, except for the comma (,) can be redefined by the user.

Some care has to be taken before defining new operators. Defining too many operators might make your source 'natural' looking, but at the same time lead to hard to understand the limits of your syntax. To ease the pain, as of SWI-Prolog 3.3.0, operators are local to the module in which they are defined. Operators can be exported from modules using a term `op`(*Precedence, Type, Name*) in the export list as specified by `module/2`. This is an extension specific to SWI-Prolog and the recommended mechanism if portability is not an important concern.

The module-table of the module `user` acts as default table for all modules and can be modified explicitly from inside a module to achieve compatibility with other Prolog systems:

```
:- module(prove,
          [ prove/1
          ]).

:- op(900, xfx, user:(=>)).
```

Unlike what many users think, operators and quoted atoms have no relation: defining an atom as an operator does **not** influence parsing characters into atoms and quoting an atom does **not** stop it from acting as an operator. To stop an atom acting as an operator, enclose it in braces like this: (myop).

op(*+Precedence, +Type, :Name***)** *[ISO]*

Declare *Name* to be an operator of type *Type* with precedence *Precedence*. *Name* can also be a list of names, in which case all elements of the list are declared to be identical operators. *Precedence* is an integer between 0 and 1200. Precedence 0 removes the declaration. *Type* is one of: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `fy` or `fx`. The '`f`' indicates the position of the functor, while `x` and `y` indicate the position of the arguments. '`y`' should be interpreted as "on this position a term with precedence lower or equal to the precedence of the functor should occur". For '`x`' the precedence of the argument must be strictly lower. The precedence of a term is 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator. A term enclosed in brackets `(...)` has precedence 0.

The predefined operators are shown in table 4.2. Operators can be redefined, unless prohibited by one of the limitations below. Applications must be careful with (re-)defining operators because changing operators may cause (other) files to be interpreted **differently**. Often this will lead to a syntax error. In other cases, text is read silently into a different term which may lead to subtle and difficult to track errors.

- It is not allowed to redefine the comma (`','`).
- The bar (`|`) can only be (re-)defined as infix operator with priority not less than 1001.
- It is not allowed to define the empty list (`[]`) or the curly-bracket-pair (`{}`) as operators.

In SWI-Prolog, operators are *local* to a module (see also section 5.8). Keeping operators in modules and using controlled import/export of operators as described with the `module/2` directive keep the issues manageable. The module `system` provides the operators from table 4.2 and these operators cannot be modified. Files that are loaded from the SWI-Prolog directories resolve operators and predicates from this `system` module rather than `user` which makes the semantics of the library and development system modules independent from operator changes to the `user` module.

current_op(*?Precedence, ?Type, ?:Name***)** *[ISO]*

True if *Name* is currently defined as an operator of type *Type* with precedence *Precedence*. See also `op/3`.

1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	dynamic, discontiguous, initialization, meta_predicate, module_transparent, multifile, thread_local, volatile
1100	xfy	;, |
1050	xfy	->, op*->
1000	xfy	,
900	fy	\+
900	fx	~
700	xfx	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600	xfy	:
500	yfx	+, -, /\, \/, xor
500	fx	?
400	yfx	*, /, //, rdiv, <<, >>, mod, rem
200	xfx	**
200	xfy	^
200	fy	+, -, \

Table 4.2: System operators

4.25 Character Conversion

Although I wouldn't really know for what you would like to use these features, they are provided for ISO compliance.

char_conversion(+*CharIn*, +*CharOut***)** *[ISO]*

Define that term-input (see `read_term/3`) maps each character read as *CharIn* to the character *CharOut*. Character conversion is only executed if the Prolog flag `char_conversion` is set to `true` and not inside quoted atoms or strings. The initial table maps each character onto itself. See also `current_char_conversion/2`.

current_char_conversion(*?CharIn, ?CharOut***)** *[ISO]*

Queries the current character conversion-table. See `char_conversion/2` for details.

4.26 Arithmetic

Arithmetic can be divided into some special purpose integer predicates and a series of general predicates for integer, floating point and rational arithmetic as appropriate. The general arithmetic predicates all handle *expressions*. An expression is either a simple number or a *function*. The arguments of a function are expressions. The functions are described in section 4.26.2.

4.26.1 Special purpose integer arithmetic

The predicates in this section provide more logical operations between integers. They are not covered by the ISO standard, although they are 'part of the community' and found as either library or built-in in many other Prolog systems.

between(_+Low, +High, ?Value_**)**
: *Low* and *High* are integers, $High \geq Low$. If *Value* is an integer, $Low \leq Value \leq High$. When *Value* is a variable it is successively bound to all integers between *Low* and *High*. If *High* is `inf` or `infinite`[50] `between/3` is true iff $Value \geq Low$, a feature that is particularly interesting for generating integers from a certain value.

succ(_?Int1, ?Int2_**)**
: True if $Int2 = Int1 + 1$ and $Int1 \geq 0$. At least one of the arguments must be instantiated to a natural number. This predicate raises the domain-error `not_less_than_zero` if called with a negative integer. E.g. `succ`(*X, 0*) fails silently and `succ`(*X, -1*) raises a domain-error.[51]

plus(_?Int1, ?Int2, ?Int3_**)**
: True if $Int3 = Int1 + Int2$. At least two of the three arguments must be instantiated to integers.

4.26.2 General purpose arithmetic

The general arithmetic predicates are optionally compiled (see `set_prolog_flag/2` and the `-O` command line option). Compiled arithmetic reduces global stack requirements and improves performance. Unfortunately compiled arithmetic cannot be traced, which is why it is optional.

+Expr1 **>** *+Expr2* *[ISO]*
: True if expression *Expr1* evaluates to a larger number than *Expr2*.

+Expr1 **<** *+Expr2* *[ISO]*
: True if expression *Expr1* evaluates to a smaller number than *Expr2*.

+Expr1 **=<** *+Expr2* *[ISO]*
: True if expression *Expr1* evaluates to a smaller or equal number to *Expr2*.

+Expr1 **>=** *+Expr2* *[ISO]*
: True if expression *Expr1* evaluates to a larger or equal number to *Expr2*.

+Expr1 **=\=** *+Expr2* *[ISO]*
: True if expression *Expr1* evaluates to a number non-equal to *Expr2*.

+Expr1 **=:=** *+Expr2* *[ISO]*
: True if expression *Expr1* evaluates to a number equal to *Expr2*.

-Number **is** *+Expr* *[ISO]*
: True when *Number* is the value to which *Expr* evaluates. Typically, `is/2` should be used with unbound left operand. If equality is to be tested, `=:=/2` should be used. For example:

`?- 1 is sin(pi/2).`	Fails!. sin(pi/2) evaluates to the float 1.0, which does not unify with the integer 1.
`?- 1 =:= sin(pi/2).`	Succeeds as expected.

---

[50] We prefer `infinite`, but some other Prolog systems already use `inf` for infinity we accept both for the time being.

[51] The behaviour to deal with natural numbers only was defined by Richard O'Keefe to support the common count-down-to-zero in a natural way. Up-to 5.1.8 `succ/2` also accepted negative integers.

Arithmetic types

SWI-Prolog defines the following numeric types:

- *integer*
  If SWI-Prolog is built using the *GNU multiple precision arithmetic library* (GMP), integer arithmetic is *unbounded*, which means that the size of integers is limited by available memory only. Without GMP, SWI-Prolog integers are 64-bits, regardless of the native integer size of the platform. The type of integer support can be detected using the Prolog flags `bounded`, `min_integer` and `max_integer`. As the use of GMP is default, most of the following descriptions assume unbounded integer arithmetic.

  Internally, SWI-Prolog has three integer representations. Small integers (defined by the Prolog flag `max_tagged_integer`) are encoded directly. Larger integers are represented as 64-bit value on the global stack. Integers that do not fit in 64-bit are represented as serialised GNU MPZ structures on the global stack.

- *rational number*
  Rational numbers (Q) are quotients of two integers. Rational arithmetic is only provided if GMP is used (see above). Rational numbers are currently not supported by a Prolog type. They are represented by the compound term `rdiv`(*N*,*M*). Rational numbers that are returned from `is/2` are *canonical*, which means M is positive and N and M have no common divisors. Rational numbers are introduced in the computation using the `rational/1`, `rationalize/1` or the `rdiv/2` (rational division) function. Using the same functor for rational division and representing rational numbers allow for passing rational numbers between computations as well as to `format/3` for printing.

  On the long term it is likely that rational numbers will become *atomic* as well as subtype of *number*. User code that creates or inspects the `rdiv`(*M*,*N*) terms will not be portable to future versions. Rationals are created using one of the functions mentioned above and inspected using `rational/3`.

- *float*
  Floating point numbers are represented using the C-type `double`

A is 2 rdiv 6	A = 1 rdiv 3
A is 4 rdiv 3 + 1	A = 7 rdiv 3
A is 4 rdiv 3 + 1.5	A = 2.83333
A is 4 rdiv 3 + rational(1.5)	A = 17 rdiv 6

Note that floats cannot represent all decimal numbers exactly. The function `rational/1` creates an *exact* equivalent of the float, while `rationalize/1` creates a rational number that is within the float rounding error from the original float. Please check the documentation of these functions for details and examples.

Rational numbers can be printed as decimal numbers with arbitrary precision using the `format/3` floating point conversion:

```
?- A is 4 rdiv 3 + rational(1.5),
   format('~50f~n', [A]).
2.83333333333333333333333333333333333333333333333333

A = 17 rdiv 6
```

Arithmetic Functions

Arithmetic functions are terms which are evaluated by the arithmetic predicates described in section 4.26.2. There are four types of arguments to functions:

Expr	Arbitrary expression, returning either a floating point value or an integer.
IntExpr	Arbitrary expression that must evaluate into an integer.
RatExpr	Arbitrary expression that must evaluate into a rational number.
FloatExpr	Arbitrary expression that must evaluate into a floating point.

For systems using bounded integer arithmetic (default is unbounded, see section 4.26.2 for details), integer operations that would cause overflow automatically convert to floating point arithmetic.

- +*Expr* *[ISO]*

$Result = -Expr$

+ +*Expr* *[ISO]*

$Result = Expr$. Note that if + is followed by a number the parser discards the +. I.e. `?- integer(+1)` succeeds.

+*Expr1* **+** +*Expr2* *[ISO]*

$Result = Expr1 + Expr2$

+*Expr1* **-** +*Expr2* *[ISO]*

$Result = Expr1 - Expr2$

+*Expr1* ***** +*Expr2* *[ISO]*

$Result = Expr1 \times Expr2$

+Expr1 **/** *+Expr2* *[ISO]*

$Result = \frac{Expr1}{Expr2}$ The the flag `iso` is `true`, both arguments are converted to float and the return value is a float. Otherwise (default), if both arguments are integers the operation returns an integer if the division is exact. If at least one of the arguments is rational and the other argument is integer, the operation returns a rational number. In all other cases the return value is a float. See also `///2` and `rdiv/2`.

+IntExpr1 **mod** *+IntExpr2* *[ISO]*

Modulo, defined as *Result* = *IntExpr1* - (*IntExpr1* div *IntExpr2*) × *IntExpr2*, where `div` is *floored* division.

+IntExpr1 **rem** *+IntExpr2* *[ISO]*

Remainder of integer division. Behaves as if defined by *Result* is *IntExpr1* - (*IntExpr1* // *IntExpr2*) × *IntExpr2*

+IntExpr1 **//** *+IntExpr2* *[ISO]*

Integer division, defined as *Result* is rnd_I(*Expr1*/*Expr2*). The function rnd_I is the default rounding used by the C-compiler and available through the Prolog flag `integer_rounding_function`. In the C99 standard, C-rounding is defined as `towards_zero`.[52]

div(*+IntExpr1, +IntExpr2***)** *[ISO]*

Integer division, defined as *Result* is $(IntExpr1 - IntExpr1 mod IntExpr2) == IntExpr2$. In other words, this is integer division that rounds towards -infinity. This function guarantees behaviour that is consistent with `mod/2`, i.e., the following holds for every pair of integers $X;Y$ where `Y =\= 0`.

```
        Q is div(X, Y),
        M is mod(X, Y),
        X =:= Y*Q+M.
```

+RatExpr **rdiv** *+RatExpr*

Rational number division. This function is only available if SWI-Prolog has been compiled with rational number support. See section 4.26.2 for details.

+IntExpr1 **gcd** *+IntExpr2*

Result is the greatest common divisor of *IntExpr1*, *IntExpr2*.

abs(*+Expr***)** *[ISO]*

Evaluate *Expr* and return the absolute value of it.

sign(*+Expr***)** *[ISO]*

Evaluate to -1 if $Expr < 0$, 1 if $Expr > 0$ and 0 if $Expr = 0$.

max(*+Expr1, +Expr2***)** *[ISO]*

Evaluates to the largest of both *Expr1* and *Expr2*. Both arguments are compared after converting to the same type, but the return value is in the original type. For example, max(2.5, 3) compares the two values after converting to float, but returns the integer 3.

---

[52] Future versions might guarantee rounding towards zero.

min(+*Expr1, +Expr2***)** *[ISO]*

Evaluates to the smallest of both *Expr1* and *Expr2*. See `max/2` for a description of type-handling.

.(+*Int, []***)**

A list of one element evaluates to the element. This implies `"a"` evaluates to the character code of the letter 'a' (97). This option is available for compatibility only. It will not work if '`style_check(+string)`' is active as `"a"` will then be transformed into a string object. The recommended way to specify the character code of the letter 'a' is `0'a`.

random(+*IntExpr***)**

Evaluates to a random integer *i* for which $0 \leq i < IntExpr$. The system has two implementations. If it is compiled with support for unbounded arithmetic (default) it uses the GMP-library random functions. In this case, each thread keeps its own random state. The default algorithm is the *Mersenne Twister* algorithm. The seed is set when the first random number in a thread is generated. If available, it is set from `/dev/random`. Otherwise it is set from the system clock. If unbounded arithmetic is not supported, random numbers are shared between threads and the seed is initialised from the clock when SWI-Prolog was started. The predicate `set_random/1` can be used to control the random number generator.

random_float

Evaluates to a random float I for which $0{:}0 < i < 1{:}0$. This function shares the random state with `random/1`. All remarks with the function `random/1` also apply for `random_float/0`. Note that both sides of the domain are *open*. This avoids evaluation errors on e.g., `log/1` or `//2` while no practical application can expect 0.0.[53]

round(+*Expr***)** *[ISO]*

Evaluates *Expr* and rounds the result to the nearest integer.

integer(+*Expr***)**

Same as `round/1` (backward compatibility).

float(+*Expr***)** *[ISO]*

Translate the result to a floating point number. Normally, Prolog will use integers whenever possible. When used around the 2nd argument of `is/2`, the result will be returned as a floating point number. In other contexts, the operation has no effect.

rational(+*Expr***)**

Convert the *Expr* to a rational number or integer. The function returns the input on integers and rational numbers. For floating point numbers, the returned rational number *exactly* represents the float. As floats cannot exactly represent all decimal numbers the results may be surprising. In the examples below, doubles can represent 0.25 and the result is as expected, in contrast to the result of `rational`(*0.1*). The function `rationalize/1` remedies this. See section 4.26.2 for more information on rational number support.

---

[53] Richard O'Keefe said: "If you *are* generating IEEE doubles with the claimed uniformity, then 0 has a 1 in $2^{53} = 1in9,007,199,254,740,992$ chance of turning up. No program that expects [0.0,1.0) is going to be surprised when 0.0 fails to turn up in a few millions of millions of trials, now is it? But a program that expects (0.0,1.0) could be devastated if 0.0 did turn up."

```
?- A is rational(0.25).

A is 1 rdiv 4
?- A is rational(0.1).
A = 3602879701896397 rdiv 36028797018963968
```

rationalize(+*Expr*)

Convert the *Expr* to a rational number or integer. The function is similar to `rational/1`, but the result is only accurate within the rounding error of floating point numbers, generally producing a much smaller denominator.[54]

```
?- A is rationalize(0.25).

A = 1 rdiv 4
?- A is rationalize(0.1).

A = 1 rdiv 10
```

float_fractional_part(+*Expr*) *[ISO]*

Fractional part of a floating-point number. Negative if *Expr* is negative, rational if *Expr* is rational and 0 if *Expr* is integer. The following relation is always true: $X isfloat_{f}ractional_{p}art(X) + float_{i}nteger_{p}art(X)$.

float_integer_part(+*Expr*) *[ISO]*

Integer part of floating-point number. Negative if *Expr* is negative, *Expr* if *Expr* is integer.

truncate(+*Expr*) *[ISO]*

Truncate *Expr* to an integer. If $Expr \geq 0$ this is the same as `floor`(*Expr*). For $Expr < 0$ this is the same as `ceil`(*Expr*). E.i. truncate rounds towards zero.

floor(+*Expr*) *[ISO]*

Evaluates *Expr* and returns the largest integer smaller or equal to the result of the evaluation.

ceiling(+*Expr*) *[ISO]*

Evaluates *Expr* and returns the smallest integer larger or equal to the result of the evaluation.

ceil(+*Expr*)

Same as `ceiling/1` (backward compatibility).

+IntExpr **>>** *+IntExpr* *[ISO]*

Bitwise shift *IntExpr1* by *IntExpr2* bits to the right. The operation performs *arithmetic shift*, which implies that the inserted most significant bits are copies of the original most significant bit.

+IntExpr **<<** *+IntExpr* *[ISO]*

Bitwise shift *IntExpr1* by *IntExpr2* bits to the left.

---

[54] The names `rational/1` and `rationalize/1` as well as their semantics are inspired by Common Lisp.

+IntExpr **\/** *+IntExpr* *[ISO]*

Bitwise 'or' *IntExpr1* and *IntExpr2*.

+IntExpr **/** *+IntExpr* *[ISO]*

Bitwise 'and' *IntExpr1* and *IntExpr2*.

+IntExpr **xor** *+IntExpr* *[ISO]*

Bitwise 'exclusive or' *IntExpr1* and *IntExpr2*.

\ *+IntExpr* *[ISO]*

Bitwise negation. The returned value is the one's complement of *IntExpr*.

sqrt(*+Expr***)** *[ISO]*

$Result = \sqrt{Expr}$

sin(*+Expr***)** *[ISO]*

$Result = \sin Expr$. *Expr* is the angle in radians.

cos(*+Expr***)** *[ISO]*

$Result = \cos Expr$. *Expr* is the angle in radians.

tan(*+Expr***)** *[ISO]*

$Result = \tan Expr$. *Expr* is the angle in radians.

asin(*+Expr***)** *[ISO]*

$Result = \arcsin Expr$. *Result* is the angle in radians.

acos(*+Expr***)** *[ISO]*

$Result = \arccos Expr$. *Result* is the angle in radians.

atan(*+Expr***)** *[ISO]*

$Result = \arctan Expr$. *Result* is the angle in radians.

atan2(*+YExpr, +XExpr***)** *[ISO]*

$Result = \arctan \frac{YExpr}{XExpr}$. *Result* is the angle in radians. The return value is in the range $[- \; ::: \;]$. Used to convert between rectangular and polar coordinate system.

atan(*+YExpr, +XExpr***)**

Same as `atan2/2` (backward compatibility).

log(*+Expr***)** *[ISO]*

Natural logarithm. $Result = \ln Expr$

log10(*+Expr***)**

Base-10 logarithm. $Result = \lg Expr$

exp(*+Expr***)** *[ISO]*

$Result = e^{Expr}$

+Expr1 ****** *+Expr2* *[ISO]*

$Result = Expr1^{Expr2}$. With unbounded integers and integer values for *Expr1* and a non-negative integer *Expr2*, the result is always integer. The integer expressions 0^I, 1^I and -1^I are guaranteed to work for any integer I. Other integer base values generate a `resource` error if the result does not fit in memory.

powm(+*IntExprBase*, +*IntExprExp*, +*IntExprMod***)**

$Result = (IntExprBase^{IntExprExp})$ modulo *IntExprMod*. Only available when compiled with unbounded integer support. This formula is required for Diffie-Hellman key-exchange, a technique where two parties can establish a secret key over a public network.

+*Expr1* ^ +*Expr2*

Same as `**/2` (backward compatibility).

pi *[ISO]*

Evaluates to the mathematical constant (3.14159...).

e

Evaluates to the mathematical constant e (2.71828...).

epsilon

Evaluates to the the difference between the float 1.0 and the first larger floating point number.

cputime

Evaluates to a floating point number expressing the CPU time (in seconds) used by Prolog up till now. See also `statistics/2` and `time/1`.

eval(+*Expr***)**

Evaluate *Expr*. Although ISO standard dictates that A=1+2, B is A works and unifies B to 3, it is widely felt that source-level variables in arithmetic expressions should have been limited to numbers. In this view the eval function can be used to evaluate arbitrary expressions.[55]

Bitvector functions The functions below are not covered by the standard. The `msb/1` function is compatible with hProlog. The others are private extensions that improve handling of —unbounded— integers as bit-vectors.

msb(+*IntExpr***)**

Return the largest integer N such that `(IntExpr >> N) /\ 1 =:= 1`. This is the (zero-origin) index of the most significant 1 bit in the value of *IntExpr*, which must evaluate to a positive integer. Errors for 0, negative integers, and non-integers.

lsb(+*IntExpr***)**

Return the smallest integer N such that `(IntExpr >> N) /\ 1 =:= 1`. This is the (zero-origin) index of the least significant 1 bit in the value of IntExpr, which must evaluate to a positive integer. Errors for 0, negative integers, and non-integers.

popcount(+*IntExpr***)**

Return the number of 1s in the binary representation of the non-negative integer *IntExpr*.

---

[55] The `eval/1` function was first introduced by ECLiPSe and is under consideration for YAP.

4.27 Misc arithmetic support predicates

set_random(+*Option*)

Controls the random number generator that accessible through the *functions* `random/1` and `random_float/0`. Note that the library `random` provides an alternative API to the same random primitives.

seed(+*Seed*)

Set the seed of the random generator for this thread. *Seed* is an integer or the atom `random`. If `random`, repeat the initialization procedure described with the function `random/1`. Here is an example:

```
?- set_random(seed(111)), A is random(6).
A = 5.
?- set_random(seed(111)), A is random(6).
A = 5.
```

state(+*State*)

Set the generator to a state fetched using the state property of `random_property/1`. Using other values may lead to undefined behaviour.[56]

random_property(*?Option*)

True when *Option* is a current property of the random generator. Currently, this predicate provides access to the state. This predicate is not present on systems where the state is inaccessible.

state(*-State*)

Describes the current state of the random generator. State is a normal Prolog term that can be asserted or written to a file. Applications should make no other assumptions about its representation. The only meaningful operation is to use as argument to `set_random/1` using the `state`(*State*) option.[57]

current_arithmetic_function(*?Head*)

True when *Head* is an evaluable function. For example:

```
?- current_arithmetic_function(sin(_)).
true.
```

4.28 Built-in list operations

Most list operations are defined in the library `lists` described in section A.12. Some that are implemented with more low-level primitives are built-in and described here.

---

[56] The limitations of the underlying (GMP) library are unknown, which makes it impossible to validate the *State*.

[57] BUG: GMP provides no portable mechanism to fetch and restore the state. The current implementation works, but the state depends on the platform. I.e., it is generally not possible to reuse the state with another version of GMP or on a CPU with different datasizes or endian-ness.

is_list(+*Term***)**

True if *Term* is bound to the empty list (`[]`) or a term with functor '`.`' and arity 2 and the second argument is a list.[58] This predicate acts as if defined by the definition below on *acyclic* terms. The implementation *fails* safely if *Term* represents a cyclic list.

```
is_list(X) :-
        var(X), !,
        fail.
is_list([]).
is_list([_|T]) :-
        is_list(T).
```

memberchk(?*Elem*, +*List***)**

Same as `once(member(`*Elem*`, `*List*`))`. See `member/2`.

length(?*List*, ?*Int***)**

True if *Int* represents the number of elements of list *List*. This predicate is a true relation and can be used to find the length of a list or produce a list (holding variables) of length *Int*. The predicate is non-deterministic, producing lists of increasing length if *List* is a *partial list* and *Int* is unbound. It raises errors if *List* is not a list or partial list or *Int* is not an integer or unbound.

sort(+*List*, -*Sorted***)** *[ISO]*

True if *Sorted* can be unified with a list holding the elements of *List*, sorted to the standard order of terms (see section 4.7). Duplicates are removed. The implementation is in C, using *natural merge sort*.[59] The `sort/2` predicate can sort a cyclic list, returning a non-cyclic version with the same elements.

msort(+*List*, -*Sorted***)**

Equivalent to `sort/2`, but does not remove duplicates. Raises a `type_error` if *List* is a cyclic list or not a list.

i(*2***)** *[ISO]*

+List, -Sorted List is a proper list whose elements are *Key−Value*, that is, terms whose principal functor is (-)/2, whose first argument is the sorting key, and whose second argument is the satellite data to be carried along with the key. `keysort/2` sorts *List* like `msort/2`, but only compares the keys. It is used to sort terms not on standard order, but on any criterion that can be expressed on a multi-dimensional scale. Sorting on more than one criterion can be done using terms as keys, putting the first criterion as argument 1, the second as argument 2, etc. The order of multiple elements that have the same *Key* is not changed. The implementation is in C, using *natural merge sort*. Fails with a `type_error` if *List* is a cyclic list or not a list or one of the elements of *List* is not a *pair*.

predsort(+*Pred*, +*List*, -*Sorted***)**

Sorts similar to `sort/2`, but determines the order of two terms by calling

---

[58] In versions before 5.0.1, `is_list/1` just checked for `[]` or `[_|_]` and `proper_list/1` had the role of the current `is_list/1`. The current definition conforms to the de-facto standard. Assuming proper coding standards, there should only be very few cases where a quick-and-dirty `is_list/1` is a good choice. Richard O'Keefe pointed at this issue.

[59] Contributed by Richard O'Keefe.

Pred(*-Delta*, *+E1*, *+E2*). This call must unify *Delta* with one of <, > or =. If built-in predicate `compare/3` is used, the result is the same as `sort/2`. See also `keysort/2`.[60]

4.29 Finding all Solutions to a Goal

findall(*+Template, :Goal, -Bag***)** *[ISO]*

Creates a list of the instantiations *Template* gets successively on backtracking over *Goal* and unifies the result with *Bag*. Succeeds with an empty list if *Goal* has no solutions. `findall/3` is equivalent to `bagof/3` with all free variables bound with the existential operator (`^`), except that `bagof/3` fails when goal has no solutions.

findall(*+Template, :Goal, -Bag, +Tail***)**

As `findall/3`, but returns the result as the difference-list *Bag*-*Tail*. The 3-argument version is defined as

```
findall(Templ, Goal, Bag) :-
        findall(Templ, Goal, Bag, [])
```

bagof(*+Template, :Goal, -Bag***)** *[ISO]*

Unify *Bag* with the alternatives of *Template*, if *Goal* has free variables besides the one sharing with *Template* bagof will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Template*. The construct *+Var*`^`*Goal* tells bagof not to bind *Var* in *Goal*. `bagof/3` fails if *Goal* has no solutions.

The example below illustrates `bagof/3` and the `^` operator. The variable bindings are printed together on one line to save paper.

```
2 ?- listing(foo).

foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).

Yes
3 ?- bagof(C, foo(A, B, C), Cs).

A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g] ;

No
4 ?- bagof(C, A^foo(A, B, C), Cs).
```

---

[60] Please note that the semantics have changed between 3.1.1 and 3.1.2

```
A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g] ;

No
5 ?-
```

setof(+*Template*, +*Goal*, -*Set***)** *[ISO]*

Equivalent to `bagof/3`, but sorts the result using `sort/2` to get a sorted list of alternatives without duplicates.

4.30 Forall

forall(:*Cond*, :*Action***)** *[semidet]*

For all alternative bindings of *Cond Action* can be proven. The example verifies that all arithmetic statements in the list *L* are correct. It does not say which is wrong if one proves wrong.

```
?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 * 2]),
                 Result =:= Formula).
```

4.31 Formatted Write

The current version of SWI-Prolog provides two formatted write predicates. The first is `writef/[1,2]`, which is compatible with Edinburgh C-Prolog. The second is `format/[1,2]`, which is compatible with Quintus Prolog. We hope the Prolog community will once define a standard formatted write predicate. If you want performance use `format/[1,2]` as this predicate is defined in C. Otherwise compatibility reasons might tell you which predicate to use.

4.31.1 Writef

writeln(+*Term***)**

Equivalent to `write(Term), nl.`

writef(+*Atom***)**

Equivalent to `writef(Atom, []).`

writef(+*Format*, +*Arguments***)**

Formatted write. *Format* is an atom whose characters will be printed. *Format* may contain certain special character sequences which specify certain formatting and substitution actions. *Arguments* then provides all the terms required to be output.

Escape sequences to generate a single special character:

`\n`	Output a newline character (see also `nl/[0,1]`)
`\l`	Output a line separator (same as `\n`)
`\r`	Output a carriage-return character (ASCII 13)
`\t`	Output the ASCII character TAB (9)
`\\`	The character \ is output
`\%`	The character `%` is output
`\nnn`	where ⟨*nnn*⟩ is an integer (1-3 digits) the character with character code ⟨*nnn*⟩ is output (NB : ⟨*nnn*⟩ is read as **decimal**)

Note that `\l`, `\nnn` and `\\` are interpreted differently when character-escapes are in effect. See section 2.15.1.

Escape sequences to include arguments from *Arguments*. Each time a % escape sequence is found in *Format* the next argument from *Arguments* is formatted according to the specification.

`%t`	`print/1` the next item (mnemonic: term)
`%w`	`write/1` the next item
`%q`	`writeq/1` the next item
`%d`	Write the term, ignoring operators. See also `write_term/2`. Mnemonic: old Edinburgh `display/1`.
`%p`	`print/1` the next item (identical to `%t`)
`%n`	Put the next item as a character (i.e., it is a character code)
`%r`	Write the next item N times where N is the second item (an integer)
`%s`	Write the next item as a String (so it must be a list of characters)
`%f`	Perform a `ttyflush/0` (no items used)
`%Nc`	Write the next item Centered in *N* columns.
`%Nl`	Write the next item Left justified in *N* columns.
`%Nr`	Write the next item Right justified in *N* columns. *N* is a decimal number with at least one digit. The item must be an atom, integer, float or string.

swritef(-*String*, +*Format*, +*Arguments***)**

Equivalent to `writef/2`, but "writes" the result on *String* instead of the current output stream. Example:

```
?- swritef(S, '%15L%w', ['Hello', 'World']).

S = "Hello          World"
```

swritef(-*String*, +*Format***)**

Equivalent to `swritef(String, Format, []).`

4.31.2 Format

The format-family of predicates is the most versatile and portable[61] way to produce textual output.

format(+*Format***)**

Defined as ‘`format(Format) :- format(Format, []).`’

format(+*Format, :Arguments***)**

Format is an atom, list of character codes, or a Prolog string. *Arguments* provides the arguments required by the format specification. If only one argument is required and this single argument is not a list the argument need not be put in a list. Otherwise the arguments are put in a list.

Special sequences start with the tilde (`~`), followed by an optional numeric argument, followed by a character describing the action to be undertaken. A numeric argument is either a sequence of digits, representing a positive decimal number, a sequence `` ` ``⟨*character*⟩, representing the character code value of the character (only useful for `~t`) or a asterisk (`*`), in which case the numeric argument is taken from the next argument of the argument list, which should be a positive integer. E.g., the following three examples all pass 46 (`.`) to `~t`:

```
?- format('~w ~46t ~w~72|~n', ['Title', 'Page']).
?- format('~w ~`.t ~w~72|~n', ['Title', 'Page']).
?- format('~w ~*t ~w~72|~n', ['Title', 46, 'Page']).
```

Numeric conversion (`d`, `D`, `e`, `E`, `f`, `g` and `G`) accept an arithmetic expression as argument. This is introduced to handle rational numbers transparently (see section 4.26.2. The floating point conversions allow for unlimited precision for printing rational numbers in decimal form. E.g., the following will write as many 3-s as you want by changing the ‘70’.

```
?- format('~70f', [10 rdiv 3]).
3.3333333333333333333333333333333333333333333333333333333333333333333333
```

- `~` Output the tilde itself.
- `a` Output the next argument, which must be an atom. This option is equivalent to **w**, except for that it requires the argument to be an atom.
- `c` Interpret the next argument as an character code and add it to the output. This argument should be an integer in the range [0, . . . , 255] (including 0 and 255).
- `d` Output next argument as a decimal number. It should be an integer. If a numeric argument is specified a dot is inserted *argument* positions from the right (useful for doing fixed point arithmetic with integers, such as handling amounts of money).
- `D` Same as **d**, but makes large values easier to read by inserting a comma every three digits left to the dot or right.
- `e` Output next argument as a floating point number in exponential notation. The numeric argument specifies the precision. Default is 6 digits. Exact representation depends on the C library function printf(). This function is invoked with the format `%.`⟨*precision*⟩`e`.

---

[61] Unfortunately not coverted by any standard.

- `E` Equivalent to **e**, but outputs a capital E to indicate the exponent.
- `f` Floating point in non-exponential notation. See C library function printf().
- `g` Floating point in **e** or **f** notation, whichever is shorter.
- `G` Floating point in **E** or **f** notation, whichever is shorter.
- `i` Ignore next argument of the argument list. Produces no output.
- `k` Give the next argument to (`write_canonical/1`).
- `n` Output a newline character.
- `N` Only output a newline if the last character output on this stream was not a newline. Not properly implemented yet.
- `p` Give the next argument to `print/1`.
- `q` Give the next argument to `writeq/1`.
- `r` Print integer in radix the numeric argument notation. Thus `~16r` prints its argument hexadecimal. The argument should be in the range $[2; : : : ; 36]$. Lower case letters are used for digits above 9.
- `R` Same as **r**, but uses upper case letters for digits above 9.
- `s` Output text from a list of character codes or a string (see `string/1` and section 4.23) from the next argument.[62]
- `@` Interpret the next argument as a goal and execute it. Output written to the `current_output` stream is inserted at this place. Goal is called in the module calling `format/3`. This option is not present in the original definition by Quintus, but supported by some other Prolog systems.
- `t` All remaining space between 2 tab stops is distributed equally over `~t` statements between the tab stops. This space is padded with spaces by default. If an argument is supplied this is taken to be the character code of the character used for padding. This can be used to do left or right alignment, centering, distributing, etc. See also `~|` and `~+` to set tab stops. A tab stop is assumed at the start of each line.
- `|` Set a tab stop on the current position. If an argument is supplied set a tab stop on the position of that argument. This will cause all `~t`'s to be distributed between the previous and this tab stop.
- `+` Set a tab stop relative to the current position. Further the same as `~|`.
- `w` Give the next argument to `write/1`.
- `W` Give the next two argument to `write_term/2`. E.g. `format(' W', [Term, [numbervars(true)]])`. This option is SWI-Prolog specific.

Example:

```
simple_statistics :-
    <obtain statistics>          % left to the user
    format('~tStatistics~t~72|~n~n'),
    format('Runtime: ~`.t ~2f~34|  Inferences: ~`.t ~D~72|~n',
```

---

[62] The **s** modifier also accept an atom for compatibility. This is deprecated due to the ambiguity of `[]`.

```
                                                   [RunT, Inf]),
    ....
```

Will output

```
                             Statistics

Runtime: .................. 3.45  Inferences: .......... 60,345
```

format(+*Output*, +*Format*, :*Arguments***)**

As `format/2`, but write the output on the given *Output*. The de-facto standard only allows *Output* to be a stream. The SWI-Prolog implementation allows all valid arguments for `with_output_to/2`.[63] For example:

```
?- format(atom(A), ’~D’, [1000000]).
A = ’1,000,000’
```

4.31.3 Programming Format

format_predicate(+*Char*, +*Head***)**

If a sequence `~c` (tilde, followed by some character) is found, the format derivatives will first check whether the user has defined a predicate to handle the format. If not, the built in formatting rules described above are used. *Char* is either an ASCII value, or a one character atom, specifying the letter to be (re)defined. *Head* is a term, whose name and arity are used to determine the predicate to call for the redefined formatting character. The first argument to the predicate is the numeric argument of the format command, or the atom `default` if no argument is specified. The remaining arguments are filled from the argument list. The example below redefines `~n` to produce *Arg* times return followed by linefeed (so a (Grr.) DOS machine is happy with the output).

```
:- format_predicate(n, dos_newline(_Arg)).

dos_newline(default) :- !,
        dos_newline(1).
dos_newline(N) :-
        (   N > 0
        ->  write(’\r\n’),
            N2 is N - 1,
            dos_newline(N2)
        ;   true
        ).
```

---

[63] Earlier versions defined `sformat/3`. These predicates have been moved to the library `backcomp`.

current_format_predicate(*?Code, ?:Head***)**

Enumerates all user-defined format predicates. *Code* is the character code of the format character. *Head* is unified with a term with the same name and arity as the predicate. If the predicate does not reside in module `user`, *Head* is qualified with the definition module of the predicate.

4.32 Terminal Control

The following predicates form a simple access mechanism to the Unix termcap library to provide terminal independent I/O for screen terminals. These predicates are only available on Unix machines. The SWI-Prolog Windows consoles accepts the ANSI escape sequences.

tty_get_capability(*+Name, +Type, -Result***)**

Get the capability named *Name* from the termcap library. See termcap(5) for the capability names. *Type* specifies the type of the expected result, and is one of `string`, `number` or `bool`. String results are returned as an atom, number result as an integer and bool results as the atom `on` or `off`. If an option cannot be found this predicate fails silently. The results are only computed once. Successive queries on the same capability are fast.

tty_goto(*+X, +Y***)**

Goto position (*X*, *Y*) on the screen. Note that the predicates `line_count/2` and `line_position/2` will not have a well defined behaviour while using this predicate.

tty_put(*+Atom, +Lines***)**

Put an atom via the termcap library function tputs(). This function decodes padding information in the strings returned by `tty_get_capability/3` and should be used to output these strings. *Lines* is the number of lines affected by the operation, or 1 if not applicable (as in almost all cases).

set_tty(*-OldStream, +NewStream***)**

Set the output stream, used by `tty_put/2` and `tty_goto/2` to a specific stream. Default is user_output.

tty_size(*-Rows, -Columns***)**

Determine the size of the terminal. Platforms:

Unix If the system provides *ioctl* calls for this, these are used and `tty_size/2` properly reh [(a5r3 100)-23361

Status is unified with the exit status of the command.

On *Win32* systems, `shell/[1,2]` executes the command using the CreateProcess() API and waits for the command to terminate. If the command ends with a `&` sign, the command is handed to the WinExec() API, which does not wait for the new task to terminate. See also `win_exec/2` and `win_shell/2`. Please note that the CreateProcess() API does **not** imply the Windows command interpreter (`command.exe` on Windows 95/98 and `cmd.exe` on Windows-NT) and therefore commands built-in to the command-interpreter can only be activated using the command interpreter. For example: `'command.exe /C copy file1.txt file2.txt'`

shell(+*Command***)**
Equivalent to ‘`shell(Command, 0)`’.

shell
Start an interactive Unix shell. Default is `/bin/sh`, the environment variable `SHELL` overrides this default. Not available for Win32 platforms.

win_exec(+*Command*, +*Show***)**
Win32 systems only. Spawns a Windows task without waiting for its completion. *Show* is one of the Win32 `SW_*` constants written in lowercase without the `SW_*`: `hide` `maximize` `minimize` `restore` `show` `showdefault` `showmaximized` `showminimized` `showminnoactive` `showna` `shownoactive` `shownormal`. In addition, `iconic` is a synonym for `minimize` and `normal` for `shownormal`

win_shell(+*Operation*, +*File*, +*Show***)**
Win32 systems only. Opens the document *File* using the windows shell-rules for doing so. *Operation* is one of `open`, `print` or `explore` or another operation registered with the shell for the given document-type. On modern systems it is also possible to pass a URL as *File*, opening the URL in Windows default browser. This call interfaces to the Win32 API ShellExecute(). The *Show* argument determines the initial state of the opened window (if any). See `win_exec/2` for defined values.

win_shell(+*Operation*, +*File***)**
Same as `win_shell`(*Operation, File, normal*)

win_registry_get_value(+*Key*, +*Name*, -*Value***)**
Win32 systems only. Fetches the value of a Win32 registry key. *Key* is an atom formed as a path-name describing the desired registry key. *Name* is the desired attribute name of the key. *Value* is unified with the value. If the value is of type `DWORD`, the value is returned as an integer. If the value is a string it is returned as a Prolog atom. Other types are currently not supported. The default ‘root’ is `HKEY_CURRENT_USER`. Other roots can be specified explicitly as `HKEY_CLASSES_ROOT`, `HKEY_CURRENT_USER`, `HKEY_LOCAL_MACHINE` or `HKEY_USERS`. The example below fetches the extension to use for Prolog files (see `README.TXT` on the Windows version):

```
?- win_registry_get_value('HKEY_LOCAL_MACHINE/Software/SWI/Prolog',
                          fileExtension,
                          Ext).

Ext = pl
```

win_folder(*?Name, -Directory***)**

Is true if *Name* is the Windows 'CSIDL' of *Directory*. If *Name* is unbound all known Windows special paths are generated. *Name* is the CSIDL after deleting the leading `CSIDL_` and mapping the constant to lowercase. Check the Windows documentation for the function SHGetSpecialFolderPath() for a description of the defined constants. This example extracts the 'My Documents' folder:

```
?- win_folder(personal, MyDocuments).

MyDocuments = 'C:/Documents and Settings/jan/My Documents'
```

getenv(*+Name, -Value***)**

Get environment variable. Fails silently if the variable does not exist. Please note that environment variable names are case-sensitive on Unix systems and case-insensitive on Windows.

setenv(*+Name, +Value***)**

Set an environment variable. *Name* and *Value* must be instantiated to atoms or integers. The environment variable will be passed to `shell/[0-2]` and can be requested using `getenv/2`. They also influence `expand_file_name/2`. Environment variables are shared between threads. Depending on the underlying C library, `setenv/2` and `unsetenv/1` may not be thread-safe and may cause memory leaks. Only changing the environment once and before starting threads is safe in all versions of SWI-Prolog.

unsetenv(*+Name***)**

Remove an environment variable from the environment. Some systems lack the underlying unsetenv() library function. On these systems `unsetenv/1` sets the variable to the empty string.

setlocale(*+Category, -Old, +New***)**

Set/Query the *locale* setting which tells the C-library how to interpret text-files, write numbers, dates, etc. Category is one of `all`, `collate`, `ctype`, `messages`, `monetary`, `numeric` or `time`. For details, please consult the C-library locale documentation. See also section 2.18.1. Please note that the locale is shared between all threads and thread-safe usage of `setlocale/3` is in general not possible. Do locale operations before starting threads or thoroughly study threading aspects of locale support in your environment before use in multi-threaded environments. Locale settings are used by `format_time/3`, `collation_key/2` and `locale_sort/2`.

unix(*+Command***)**

This predicate comes from the Quintus compatibility library and provides a partial implementation thereof. It provides access to some operating system features and unlike the name suggests, is not operating system specific. Defined *Command*'s are below.

system(*+Command***)**

Equivalent to calling `shell/1`. Use for compatibility only.

shell(*+Command***)**

Equivalent to calling `shell/1`. Use for compatibility only.

shell
Equivalent to calling `shell/0`. Use for compatibility only.

cd
Equivalent to calling `working_directory/2` to the expansion (see `expand_file_name/2`) of ~. For compatibility only.

cd(+*Directory***)**
Equivalent to calling `working_directory/2`. Use for compatibility only.

argv(-*Argv***)**
Unify *Argv* with the list of command-line arguments provides to this Prolog run. Please note that Prolog system-arguments and application arguments are separated by `--`. Integer arguments are passed as Prolog integers, float arguments and Prolog floating point numbers and all other arguments as Prolog atoms. New applications should use the Prolog flag `argv`. See also prolog Prolog flag `argv`.

A stand-alone program could use the following skeleton to handle command-line arguments. See also section 2.10.2.

```
main :-
        current_prolog_flag(argv, Argv),
        append(_PrologArgs, [--|AppArgs], Argv), !,
        main(AppArgs).
```

4.33.1 Dealing with time and date

Representing time in a computer system is surprisingly complicated. There are a large number of time representations in use and the correct choice depends on factors such as compactness, resolution and desired operations. Humans tend to think about time in hours, days, months, years or centuries. Physicists think about time in seconds. But, a month does not have a defined number of seconds. Even a day does not have a defined number of seconds as sometimes a leap-second is introduced to synchronise properly with our earth's rotation. At the same time, resolution demands range from better then pico-seconds to millions of years. Finally, civilizations have a wide range of calendars. Although there exist libraries dealing with most if this complexity, our desire to keep Prolog clean and lean stops us from fully supporting these.

For human-oriented tasks, time can be broken into years, months, days, hours, minutes, seconds and a timezone. Physicists prefer to have time in an arithmetic type representing seconds or fraction thereof, so basic arithmetic deal with comparison and durations. An additional advantage of the physicists approach is that it requires much less space. For these reasons, SWI-Prolog uses an arithmetic type as its prime time representation.

Many C libraries deal with time using fixed-point arithmetic, dealing with a large but finite time interval at constant resolution. In our opinion using a floating point number is a more natural choice as we can use a natural unit and the interface does not need to be changed if a higher resolution is required in the future. Our unit of choice is the second as it is the scientific unit.[64] We have placed our origin at 1970-1-1T0:0:0Z for compatibility with the POSIX notion of time as well as with older time support provided by SWI-Prolog.

---

[64] Using Julian days is a choice made by the Eclipse team. As conversion to dates is needed for a human readable notation of time and Julian days cannot deal naturally with leap seconds, we decided for second as our unit.

Where older versions of SWI-Prolog relied on the POSIX conversion functions, the current implementation uses libtai to realise conversion between time-stamps and calendar dates for a period of 10 million years.

Time and date data-structures

We use the following time representations

TimeStamp

A TimeStamp is a floating point number expression the time in seconds since the Epoch at 1970-1-1.

date(*Y,M,D,H,Mn,S,Off,TZ,DST***)**

We call this term a *date-time* structure. The first 5 fields are integers expressing the year, month (1..12), day (1..31), hour (0..23), Minute (0..59). The *S* field holds the seconds as a floating point number between 0.0 and 60.0. *Off* is an integer representing the offset relative to UTC in seconds where positive values are west of Greenwich. If converted from local time (see `stamp_date_time/3`, *TZ* holds the name of the local timezone. If the timezone is not known *TZ* is the atom `-`. *DST* is `true` if daylight saving time applies to the current time, `false` if daylight saving time is relevant but not effective and `-` if unknown or the timezone has no daylight saving time.

date(*Y,M.D***)**

Date using the same values as described above. Extracted using `date_time_value/3`.

time(*H,Mn,S***)**

Time using the same values as described above. Extracted using `date_time_value/3`.

Time and date predicates

get_time(*-TimeStamp***)**

Return the current time as a *TimeStamp*. The granularity is system dependent. See section 4.33.1.

stamp_date_time(*+TimeStamp, -DateTime, +TimeZone***)**

Convert a *TimeStamp* to a *DateTime* in the given time zone. See section 4.33.1 for details on the data-types. *TimeZone* describes the timezone for the conversion. It is one of `local` to extract the local time, `'UTC'` to extract at UTC time or an integer describing the seconds west of Greenwich.

date_time_stamp(*+DateTime, -TimeStamp***)**

Compute the timestamp from a date/9 term. Values for month, day, hour, minute or second need not be normalized. This flexibility allows for easy computation of the time at any given number of these units from a given timestamp. Normalization can be achieved following this call with `stamp_date_time/3`. This example computes the date 200 days after 2006-7-14:

```
?- date_time_stamp(date(2006,7,214,0,0,0,0,-,-), Stamp),
   stamp_date_time(Stamp, D, 0),
   date_time_value(date, D, Date).
Date = date(2007, 1, 30)
```

date_time_value(*?Key, +DateTime, ?Value***)**

Extract values from a date/9 term. Provided keys are:

key	**value**
`year`	Calendar year as an integer
`month`	Calendar month as an integer 1..12
`day`	Calendar day as an integer 1..31
`hour`	Clock hour as an integer 0..23
`minute`	Clock minute as an integer 0..59
`second`	Clock second as a float 0.0..60.0
`utc_offset`	Offset to UTC in seconds (positive is west)
`time_zone`	Name of timezone; fails if unknown
`daylight_saving`	Bool (true) if dst is effective
`date`	Term `date`(*Y,M,D*)
`time`	Term `time`(*H,M,S*)

format_time(*+Out, +Format, +StampOrDateTime***)**

Modelled after POSIX strftime(), using GNU extensions. *Out* is a destination as specified with `with_output_to/2`. *Format* is an atom or string with the following conversions. Conversions start with a tilde (%) character.[65] *StampOrDateTime* is either a (numeric time-stamp, a term `date`(*Y,M,D,H,M,S,O,TZ,DST*) or a term `date`(*Y,M,D*).

- `a` The abbreviated weekday name according to the current locale. Use `format_time/4` for POSIX locale.
- `A` The full weekday name according to the current locale. Use `format_time/4` for POSIX locale.
- `b` The abbreviated month name according to the current locale. Use `format_time/4` for POSIX locale.
- `B` The full month name according to the current locale. Use `format_time/4` for POSIX locale.
- `c` The preferred date and time representation for the current locale.
- `C` The century number (year/100) as a 2-digit integer.
- `d` The day of the month as a decimal number (range 01 to 31).
- `D` Equivalent to %m/%d/%y. (Yecch for Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in international context this format is ambiguous and should not be used.)
- `e` Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space.
- `E` Modifier. Not implemented.
- `f` Number of microseconds. The `f` can be prefixed by an integer to print the desired number of digits. E.g., `%3f` prints milliseconds. This format is not covered by any standard, but available with different format-specifiers in various incarnations of the strftime() function.

---

[65] Descriptions taken from Linux Programmer's Manual

`F`	Equivalent to %Y-%m-%d (the ISO 8601 date format).
`g`	Like %G, but without century, i.e., with a 2-digit year (00-99).
`G`	The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see %V). This has the same format and value as %y, except that if the ISO week number belongs to the previous or next year, that year is used instead.
`V`	The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also %U and %W.
`h`	Equivalent to %b.
`H`	The hour as a decimal number using a 24-hour clock (range 00 to 23).
`I`	The hour as a decimal number using a 12-hour clock (range 01 to 12).
`j`	The day of the year as a decimal number (range 001 to 366).
`k`	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also %H.)
`l`	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also %I.)
`m`	The month as a decimal number (range 01 to 12).
`M`	The minute as a decimal number (range 00 to 59).
`n`	A newline character.
`O`	Modifier to select locale-specific output. Not implemented.
`p`	Either 'AM' or 'PM' according to the given time value, or the corresponding strings for the current locale. Noon is treated as 'pm' and midnight as 'am'.
`P`	Like %p but in lowercase: 'am' or 'pm' or a corresponding string for the current locale.
`r`	The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to '%I:%M:%S %p'.
`R`	The time in 24-hour notation (%H:%M). For a version including the seconds, see %T below.
`s`	The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.
`S`	The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for occasional leap seconds.)
`t`	A tab character.
`T`	The time in 24-hour notation (%H:%M:%S).
`u`	The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w.
`U`	The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also %V and %W.
`w`	The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.
`W`	The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
`x`	The preferred date representation for the current locale without the time.

`X` The preferred time representation for the current locale without the date.

`y` The year as a decimal number without a century (range 00 to 99).

`Y` The year as a decimal number including the century.

`z` The time-zone as hour offset from GMT using the format HHmm. Required to emit RFC822-conforming dates (using `'%a, %d %b %Y %T %z'`). Our implementation supports `%:z`, which modifies the output to HH:mm as required by XML-Schema. Note that both notations are valid in ISO8601. The sequence `%:z` is compatible to the GNU date(1) command.

`Z` The time zone or name or abbreviation.

`+` The date and time in date(1) format.

`%` A literal '%' character.

The table below, gives some format strings for popular time representations. RFC1123 is used by HTTP. The full implementation of `http_timestamp/2` as available from `http/http_header` is here.

```
http_timestamp(Time, Atom) :-
        stamp_date_time(Time, Date, 'UTC'),
        format_time(atom(Atom),
                    '%a, %d %b %Y %T GMT',
                    Date, posix).
```

Standard	Format string
xsd	`'%FT%T%:z'`
ISO8601	`'%FT%T%z'`
RFC822	`'%a, %d %b %Y %T %z'`
RFC1123	`'%a, %d %b %Y %T GMT'`

format_time(+*Out*, +*Format*, +*StampOrDateTime*, +*Locale***)**

Format time given a specified *Locale*. This predicate is a work-around for lacking proper portable and thread-safe time and locale handling in current C libraries. In its current implementation the only value allowed for *Locale* is `posix`, which currently only modifies the behaviour or the `a`, `A`, `b` and `B` format specifiers. The predicate is used to be able to emit POSIX locale week and month names for emitting standardised time-stamps such as RFC1123.

parse_time(+*Text*, -*Stamp***)**

Same as `parse_time`(*Text*, *_Format*, *Stamp*). See `parse_time/3`.

parse_time(+*Text*, ?*Format*, -*Stamp***)**

Parse a textual time representation, producing a time-stamp. Supported formats for *Text* are in the table below. If the format is known, it may be given to reduce parse-time and avoid ambiguities. Otherwise, *Format* is unified with the format encountered.

Name	Example
rfc_1123	`Fri, 08 Dec 2006 15:29:44 GMT`

iso_8601	`2006-12-08T17:29:44+02:00`
	`20061208T172944+0200`
	`2006-12-08T15:29Z`
	`2006-12-08`
	`20061208`
	`2006-12`
	`2006-W49-5`
	`2006-342`

day_of_the_week(+*Date*,-*DayOfTheWeek***)**

Computes the day of the week for a given date. *Date* `=` `date(`*Year*`,`*Month*`,`*Day*`)`, Days of the week are numbered from one to seven: monday = 1, tuesday = 2, . . . , sunday = 7.

4.33.2 Controlling the `swipl-win.exe` console window

The Windows executable `swipl-win.exe` console has a number of predicates to control the appearance of the console. Being totally non-portable, we do not advice using it for your own application, but use XPCE or another portable GUI platform instead. We give the predicates for reference here.

window_title(-*Old*, +*New***)**

Unify *Old* with the title displayed in the console and change the title to *New*.[66]

win_window_pos(+*ListOfOptions***)**

Interface to the MS-Windows SetWindowPos() function, controlling size, position and stacking order of the window. *ListOfOptions* is a list that may hold any number of the terms below.

size(*W*, *H***)**

Change the size of the window. *W* and *H* are expressed in character-units.

position(*X*, *Y***)**

Change the top-left corner of the window. The values are expressed in pixel units.

zorder(*ZOrder***)**

Change the location in the window stacking order. Values are `bottom`, `top`, `topmost` and `notopmost`. *Topmost* windows are displayed above all other windows.

show(*Bool***)**

If `true`, show the window, if `false` hide the window.

activate

If present, activate the window.

win_has_menu

True if `win_insert_menu/2` and `win_insert_menu_item/4` are present.

win_insert_menu(+*Label*, +*Before***)**

Insert a new entry (pulldown) in the menu. If the menu already contains this entry, nothing is

---

[66] BUG: This predicate should have been called `win_window_title` for consistent naming.

done. The *Label* is the label and using the Windows conventions, a letter prefixed with `&` is underlined and defines the associated accelerator key. *Before* is the label before which this one must be inserted. Using `-` adds the new entry at the end (right). For example, the call below adds a Application entry just before the Help menu.

```
win_insert_menu('&Application', '&Help')
```

win_insert_menu_item(+*Pulldown*, +*Label*, +*Before*, :*Goal***)**
Add an item to the named *Pulldown* menu. *Label* and *Before* are handled as in `win_insert_menu/2`, but the label `-` inserts a *separator*. *Goal* is called if the user selects the item.

4.34 File System Interaction

access_file(+*File*, +*Mode***)**
True if *File* exists and can be accessed by this prolog process under mode *Mode*. *Mode* is one of the atoms `read`, `write`, `append`, `exist`, `none` or `execute`. *File* may also be the name of a directory. Fails silently otherwise. `access_file(File, none)` simply succeeds without testing anything.

If 'Mode' is `write` or `append`, this predicate also succeeds if the file does not exist and the user has write-access to the directory of the specified location.

exists_file(+*File***)**
True if *File* exists and is a regular file. This does not imply the user has read and/or write permission for the file.

file_directory_name(+*File*, -*Directory***)**
Extracts the directory-part of *File*. The returned *Directory* name does not end in `/`. There are two special cases. The directory-name of `/` is `/` itself and the directory-name if *File* does not contain any `/` characters is `.`.

file_base_name(+*File*, -*BaseName***)**
Extracts the filename part from a path specification. If *File* does not contain any directory separators, *File* is returned.

same_file(+*File1*, +*File2***)**
True if both filenames refer to the same physical file. That is, if *File1* and *File2* are the same string or both names exist and point to the same file (due to hard or symbolic links and/or relative vs. absolute paths).

exists_directory(+*Directory***)**
True if *Directory* exists and is a directory. This does not imply the user has read, search and or write permission for the directory.

delete_file(+*File***)**
Remove *File* from the file system.

rename_file(+*File1, +File2***)**

Rename *File1* into *File2*. The semantics is compatible to the POSIX semantics of the rename() system call as far as the operating system allows. if *File2* exists, the operation succeeds (except for possible permission errors) and is *atomic* (meaning there is no window where *File2* does not exist).

size_file(+*File, -Size***)**

Unify *Size* with the size of *File* in bytes.

time_file(+*File, -Time***)**

Unify the last modification time of *File* with *Time*. *Time* is a floating point number expressing the seconds elapsed since Jan 1, 1970. See also `convert_time/[2,8]` and `get_time/1`.

absolute_file_name(+*File, -Absolute***)**

Expand a local file-name into an absolute path. The absolute path is canonised: references to `.` and `..` are deleted. This predicate ensures that expanding a file-name it returns the same absolute path regardless of how the file is addressed. SWI-Prolog uses absolute file names to register source files independent of the current working directory. See also `absolute_file_name/3`. See also `absolute_file_name/3` and `expand_file_name/2`.

absolute_file_name(+*Spec, -Absolute, +Options***)**

Converts the given file specification into an absolute path. *Spec* is a term Alias(Relative) (e.g., `(library(lists))`, a relative file-name or an absolute file name. The primary intention of this predicate is to resolve files specified as Alias(Relative). *Option* is a list of options to guide the conversion:

extensions(*ListOfExtensions***)**

List of file-extensions to try. Default is `''`. For each extension, `absolute_file_name/3` will first add the extension and then verify the conditions imposed by the other options. If the condition fails, the next extension of the list is tried. Extensions may be specified both as `..ext` or plain `ext`.

relative_to(+*FileOrDir***)**

Resolve the path relative to the given directory or directory the holding the given file. Without this option, paths are resolved relative to the working directory (see `working_directory/2`) or, if *Spec* is atomic and `absolute_file_name/[2,3]` is executed in a directive, it uses the current source-file as reference.

access(*Mode***)**

Imposes the condition access_file(*File*, *Mode*). *Mode* is on of `read`, `write`, `append`, `execute`, `exist` or `none`. See also `access_file/2`.

file_type(*Type***)**

Defines extensions. Current mapping: `txt` implies `['']`, `prolog` implies `['.pl', '']`, `executable` implies `['.so', '']`, `qlf` implies `['.qlf', '']` and `directory` implies `['']`. The file-type `source` is an alias for `prolog` for compatibility with SICStus Prolog. See also `prolog_file_type/2`. This predicate only returns non-directories, unless the option `file_type(`*directory*`)` is specified.

file_errors(*fail/error***)**
If `error` (default), throw and `existence_error` exception if the file cannot be found. If `fail`, stay silent.[67]

solutions(*first/all***)**
If `first` (default), the predicates leaves no choice-point. Otherwise a choice-point will be left and backtracking may yield more solutions.

expand(*true/false***)** be lefty.9091Tf 68.5(8.5d2)]TJ/ET dica3 -4Prolo-25d 0t/al3ag.9091 Tı
If `true` (default is `false`) and *Spec*

be lefUniETq-289091 Tf 37.581 0 Td (3050def)10(El)ri-289091 T7 37.581 0 Td (30rst)]TJ/F

Before expanding wildcards, the construct `$`*var* is expanded to the value of the environment variable *var* and a possible leading ~ character is expanded to the user's home directory.[69].

prolog_to_os_filename(*?PrologPath, ?OsPath***)**
Converts between the internal Prolog pathname conventions and the operating-system pathname conventions. The internal conventions are Unix and this predicates is equivalent to =/2 (unify) on Unix systems. On DOS systems it will change the directory-separator, limit the filename length map dots, except for the last one, onto underscores.

read_link(*+File, -Link, -Target***)**
If *File* points to a symbolic link, unify *Link* with the value of the link and *Target* to the file the link is pointing to. *Target* points to a file, directory or non-existing entry in the file system, but never to a link. Fails if *File* is not a link. Fails always on systems that do not support symbolic links.

tmp_file(*+Base, -TmpName***)** *[deprecated]*
Create a name for a temporary file. *Base* is an identifier for the category of file. The *TmpName* is guaranteed to be unique. If the system halts, it will automatically remove all created temporary files. *Base* is used as part of the final filename. Portable applications should limit themselves to alphanumerical characters.

Because it is possible to guess the generated filename, attackers may create the filesystem entry as a link and possibly create a security issue. New code should use `tmp_file_stream/3`.

tmp_file_stream(*+Encoding, -FileName, -Stream***)**
Create a temporary file name *FileName* and open it for writing in the given *Encoding*. *Encoding* is a text-encoding name or `binary`. *Stream* is the output stream. If the OS supports it, the created file is only accessible to the current user. If the OS supports it, the file is created using the open()-flag `O_EXCL`, which guarantees that the file did not exist before this call. This predicate is a safe replacement of `tmp_file/2`. Note that in those cases where the temporary file is needed to store output from an external command, the file must be closed first. E.g., the following downloads a file from a URL to a temporary file and opens the file for reading (On Unix systems you can delete the file after opening it for reading for cleanup):

```
open_url(URL, In) :-
        tmp_file_stream(text, File, Stream),
        close(Stream),
        process_create(curl, ['-o', File, URL], []),
        open(File, read, In),
        delete_file(File).              % Unix-only
```

Temporary files created using this call are removed if the Prolog process terminates. Calling `delete_file/1` using *FileName* removes the file and removes the entry from the administration of files-to-be-deleted.

---

[69] On Windows, the home directory is determined as follows: if the environment variable `HOME` exists, this is used. If the variables `HOMEDRIVE` and `HOMEPATH` exist (Windows-NT), these are used. At initialisation, the system will set the environment variable `HOME` to point to the SWI-Prolog home directory if neither `HOME` nor `HOMEPATH` and `HOMEDRIVE` are defined

make_directory(+*Directory***)**

Create a new directory (folder) on the filesystem. Raises an exception on failure. On Unix systems, the directory is created with default permissions (defined by the process *umask* setting).

delete_directory(+*Directory***)**

Delete directory (folder) from the filesystem. Raises an exception on failure. Please note that in general it will not be possible to delete a non-empty directory.

working_directory(-*Old*, +*New***)**

Unify *Old* with an absolute path to the current working directory and change working directory to *New*. Use the pattern `working_directory`(*CWD, CWD*) to get the current directory. See also `absolute_file_name/2` and `chdir/1`.[70] Note that the working directory is shared between all threads.

chdir(+*Path***)**

Compatibility predicate. New code should use `working_directory/2`.

4.35 User Top-level Manipulation

break

Recursively start a new Prolog top level. This Prolog top level has its own stacks, but shares the heap with all break environments and the top level. Debugging is switched off on entering a break and restored on leaving one. The break environment is terminated by typing the system's end-of-file character (control-D). If the `-t` *`toplevel`* command line option is given this goal is started instead of entering the default interactive top level (`prolog/0`).

abort

Abort the Prolog execution and restart the top level. If the `-t` *`toplevel`* command line options is given this goal is started instead of entering the default interactive top level.

Aborting is implemented by throwing the reserved exception `$aborted`. This exception can be caught using `catch/3`, but the recovery goal is wrapped with a predicate that prunes the choice-points of the recovery goal (i.e., as `once/1`) and re-throws the exception. This is illustrated in the example below, where we press control-C and 'a'.

```
?- catch((repeat,fail), E, true).
^CAction (h for help) ? abort
% Execution Aborted
```

halt *[ISO]*

Terminate Prolog execution. Open files are closed and if the command line option `-tty` is not active the terminal status (see Unix stty(1)) is restored. Hooks may be registered both in Prolog and in foreign code. Prolog hooks are registered using `at_halt/1`. `halt/0` is equivalent to `halt(0)`.[71]

---

[70] BUG: Some of the file-I/O predicates use local filenames. Changing directory while file-bound streams are open causes wrong results on `telling/1`, `seeing/1` and `current_stream/3`

[71] BUG: In the multi-threaded version, `halt/0` does not work when not called from the *main* thread. In the current system a `permission_error` exception is raised. Future versions may enable `halt/0` from any thread.

halt(+*Status***)** *[ISO]*

Terminate Prolog execution with given status. Status is an integer. See also `halt/0`.

prolog

This goal starts the default interactive top level. Queries are read from the stream `user_input`. See also the Prolog flag `history`. The `prolog/0` predicate is terminated (succeeds) by typing the end-of-file character (typically control-D).

The following two hooks allow for expanding queries and handling the result of a query. These hooks are used by the top-level variable expansion mechanism described in section 2.8.

expand_query(+*Query, -Expanded, +Bindings, -ExpandedBindings***)**

Hook in module `user`, normally not defined. *Query* and *Bindings* represents the query read from the user and the names of the free variables as obtained using `read_term/3`. If this predicate succeeds, it should bind *Expanded* and *ExpandedBindings* to the query and bindings to be executed by the top-level. This predicate is used by the top-level (`prolog/0`). See also `expand_answer/2` and `term_expansion/2`.

expand_answer(+*Bindings, -ExpandedBindings***)**

Hook in module `user`, normally not defined. Expand the result of a successfully executed top-level query. *Bindings* is the query ⟨*Name*⟩ = ⟨*Value*⟩ binding list from the query. *ExpandedBindings* must be unified with the bindings the top-level should print.

4.36 Creating a Protocol of the User Interaction

SWI-Prolog offers the possibility to log the interaction with the user on a file.[72] All Prolog interaction, including warnings and tracer output, are written on the protocol file.

protocol(+*File***)**

Start protocolling on file *File*. If there is already a protocol file open then close it first. If *File* exists it is truncated.

protocola(+*File***)**

Equivalent to `protocol/1`, but does not truncate the *File* if it exists.

noprotocol

Stop making a protocol of the user interaction. Pending output is flushed on the file.

protocolling(-*File***)**

True if a protocol was started with `protocol/1` or `protocola/1` and unifies *File* with the current protocol output file.

4.37 Debugging and Tracing Programs

This section is a reference to the debugger interaction predicates. A more use-oriented overview of the debugger is in section 2.9.

If you have installed XPCE, you can use the graphical front-end of the tracer. This front-end is installed using the predicate `guitracer/0`.

---

[72] A similar facility was added to Edinburgh C-Prolog by Wouter Jansweijer.

trace

Start the tracer. `trace/0` itself cannot be seen in the tracer. Note that the Prolog top-level treats `trace/0` special; it means 'trace the next goal'.

tracing

True if the tracer is currently switched on. `tracing/0` itself can not be seen in the tracer.

notrace

Stop the tracer. `notrace/0` itself cannot be seen in the tracer.

guitracer

Installs hooks (see `prolog_trace_interception/4`) into the system that redirects tracing information to a GUI front-end providing structured access to variable-bindings, graphical overview of the stack and highlighting of relevant source-code.

noguitracer

Reverts back to the textual tracer.

trace(+*Pred***)**

Equivalent to `trace(`*Pred*`, +all).`

trace(+*Pred*, +*Ports***)**

Put a trace-point on all predicates satisfying the predicate specification *Pred*. *Ports* is a list of port names (`call`, `redo`, `exit`, `fail`). The atom `all` refers to all ports. If the port is preceded by a `-` sign the trace-point is cleared for the port. If it is preceded by a `+` the trace-point is set.

The predicate `trace/2` activates debug mode (see `debug/0`). Each time a port (of the 4-port model) is passed that has a trace-point set the goal is printed as with `trace/0`. Unlike `trace/0` however, the execution is continued without asking for further information. Examples:

`?- trace(hello).`	Trace all ports of hello with any arity in any module.
`?- trace(foo/2, +fail).`	Trace failures of foo/2 in any module.
`?- trace(bar/1, -all).`	Stop tracing bar/1.

The predicate `debugging/0` shows all currently defined trace-points.

notrace(:*Goal***)**

Call *Goal*, but suspend the debugger while *Goal* is executing. The current implementation cuts the choice-points of *Goal* after successful completion. See `once/1`. Later implementations may have the same semantics as `call/1`.

debug

Start debugger. In debug mode, Prolog stops at spy- and trace-points, disables last-call optimisation and aggressive destruction of choice points to make debugging information accessible. Implemented by the Prolog flag `debug`.

Note that the `min_free` parameters of all stacks is enlarged to 8 K-cells if debugging is switched off to avoid excessive GC. GC complicates tracing because it renames the _$G\langle NNN\rangle$ variables and replaces unreachable variables with the atom `\bnfmeta{garbage_collected}`. Calling `nodebug/0` does *not* reset the initial free-margin because several parts of the toplevel and debugger disable debugging of system code-regions. See also `set_prolog_stack/2`.

nodebug

Stop debugger. Implemented by the Prolog flag `debug`. See also `debug/0`.

debugging

Print debug status and spy points on current output stream. See also the Prolog flag `debug`.

spy(+*Pred***)**

Put a spy point on all predicates meeting the predicate specification *Pred*. See section 4.5.

nospy(+*Pred***)**

Remove spy point from all predicates meeting the predicate specification *Pred*.

nospyall

Remove all spy points from the entire program.

leash(*?Ports***)**

Set/query leashing (ports which allow for user interaction). *Ports* is one of +*Name*, -*Name*, *?Name* or a list of these. +*Name* enables leashing on that port, -*Name* disables it and *?Name* succeeds or fails according to the current setting. Recognised ports are: `call`, `redo`, `exit`, `fail` and `unify`. The special shorthand `all` refers to all ports, `full` refers to all ports except for the unify port (default). `half` refers to the `call`, `redo` and `fail` port.

visible(+*Ports***)**

Set the ports shown by the debugger. See `leash/1` for a description of the port specification. Default is `full`.

unknown(-*Old*, +*New***)**

Edinburgh-prolog compatibility predicate, interfacing to the ISO prolog flag `unknown`. Values are `trace` (meaning `error`) and `fail`. If the `unknown` flag is set to `warning`, `unknown/2` reports the value as `trace`.

style_check(+*Spec***)**

Set style checking options. *Spec* is either +⟨*option*⟩, −⟨*option*⟩, `?(`⟨*option*⟩`)`[73] or a list of such options. +⟨*option*⟩ sets a style checking option, −⟨*option*⟩ clears it and `?(`⟨*option*⟩`)` succeeds or fails according to the current setting. `consult/1` and derivatives resets the style checking options to their value before loading the file. If—for example—a file containing long atoms should be loaded the user can start the file with:

```
:- style_check(-atom).
```

Currently available options are:

---

[73] In older versions '`?`' was a prefix operator. Inversions after 5.5.13, explicit brackets are needed.

Name	Default	Description
`singleton`	on	`read_clause/1` (used by `consult/1`) warns on variables only appearing once in a term (clause) which have a name not starting with an underscore. See section 2.15.1 for details on variable handling and warnings.
`atom`	on	`read/1` and derivatives will produce an error message on quoted atoms or strings longer than 5 lines.
`discontiguous`	on	Warn if the clauses for a predicate are not together in the same source file.
`string`	off	Backward compatibility. See the Prolog flag `double_quotes` (`current_prolog_flag/2`).
`charset`	off	Warn on atoms and variables holding non-ASCII characters that are not quoted. See also section 2.15.1.

4.38 Obtaining Runtime Statistics

statistics(+*Key*, -*Value***)**

Unify system statistics determined by *Key* with *Value*. The possible keys are given in the table 4.3. The last part of the table contains keys for compatibility with other Prolog implementations (Quintus) for improved portability. Note that the ISO standard does not define methods to collect system statistics. Space unit is bytes[74]. Times are in seconds, represented as a floating point number. The Quintus compatibility keys express times in milliseconds.

statistics

Display a table of system statistics on the current output stream.

time(:*Goal***)**

Execute *Goal* just like `call/1` and print used time, number of logical inferences and the average number of *lips* (logical inferences per second). Note that SWI-Prolog counts the actual executed number of inferences rather than the number of passes through the call- and redo ports of the theoretical 4-port model. If *Goal* is non-deterministic prints statistics for each solution, where the reported values are relative to the previous answer.

4.39 Execution profiling

This section describes the hierarchical execution profiler introduced in SWI-Prolog 5.1.10. This profiler is based on ideas from `gprof` described in [Graham *et al.*, 1982]. The profiler consists of two parts: the information-gathering is built into the kernel,[75] and a presentation component which is defined in the `statistics` library. The latter can be hooked, which is used by the XPCE module `swi/pce_profile` to provide an interactive graphical representation of results.

---

[74] This may change to 'cells' to enhance compatibility between the 32- and 64-bit versions.

[75] There are two implementations; one based on setitimer() using the `SIGPROF` signal and one using Windows Multi Media (MM) timers. On other systems the profiler is not provided.

agc	Number of atom garbage-collections performed
agc_gained	Number of atoms removed
agc_time	Time spent in atom garbage-collections
process_cputime	(User) CPU time since Prolog was started in seconds
cputime	(User) CPU time since thread was started in seconds
inferences	Total number of passes via the call and redo ports since Prolog was started.
heapused	Bytes heap in use by Prolog (0 if not maintained).
heap_gc	Number of heap garbage-collections performed. Only provided if SWI-Prolog is configured with Boehm-GC. See also `garbage_collect_heap/0`.
c_stack	System (C-) stack limit. 0 if not known.
stack	Total memoryQBT/F37 10.9091 Tf 21i.9 522s0(spent)-250(inll0(since)-250(seconds)]TJE

[illegible] Td own.

4.39.1 Profiling predicates

Currently, the interface is kept compatible with the old profiler. As experience grows, it is likely that the old interface is replaced with one that better reflects the new capabilities. Feel free to examine the internal interfaces and report useful application thereof.

profile(*:Goal***)**

Execute *Goal* just like `time/1`, collecting profiling statistics and call `show_profile`(*plain, 25*). With XPCE installed this opens a graphical interface to the collected profiling data.

profile(*:Goal, +Style, +Number***)**

Execute *Goal* just like `time/1`. Collect profiling statistics and show the top *Number* procedures on the current output stream (see `show_profile/1`) using *Style*. The results are kept in the database until `reset_profiler/0` or `profile/3` is called and can be displayed again with `show_profile/1`. The `profile/1` predicate is a backward compatibility interface to `profile/1`. The other predicates in this section are low-level predicates for special cases.

show_profile(*+Style, +Number***)**

Show the collected results of the profiler. It shows the top *Number* predicates according the percentage CPU-time used. If *Style* is `plain` the time spent in the predicates itself is displayed. If *Style* is `cumulative` the time spent in its siblings (callees) is added to the predicate.

This predicate first calls prolog:show_profile_hook/2. If XPCE is loaded this hook is used to activate a GUI interface to visualise the profile results.

show_profile(*+Number***)**

Compatibility. Same as `show_profile`(*plain, Number*).

profiler(*-Old, +New***)**

Query or change the status of the profiler. The status is a boolean (`true` or `false`) stating whether or not the profiler is collecting data. It can be used to enable or disable profiling certain parts of the program.

reset_profiler

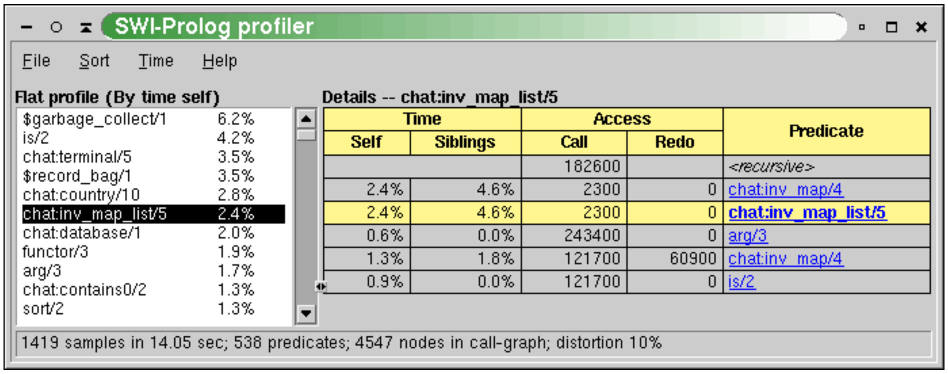
Switches the profiler to `false` and clears all collected statistics.

noprofile(*+Name/+Arity, ...***)**

Declares the predicate *Name/Arity* to be invisible to the profiler. The time spend in the named predicate is added to the caller and the callees are linked directly to the caller. This is particularly useful for simple meta-predicates such as `call/1`, `ignore/1`, `catch/3`, etc.

4.39.2 Visualizing profiling data

Browsing the annotated call-tree as described in section 4.39.3 itself is not very attractive. Therefore, the results are combined per predicate, collecting all *callers* and and *callees* as well as the propagation of time and activations in both directions. Figure 4.1 illustrates this. The central yellowish line is the 'current' predicate with counts for time spent in the predicate ('Self'), time spent in its children ('Siblings'), activations through the call and redo ports. Above that are the *callers*. Here, the two time fields indicate how much time is spent serving each of the callers. The columns sum to the time in the yellowish line. The caller <*recursive*> are the number of recursive calls. Below the yellowish lines



Flat profile (By time self)

$garbage_collect/1	6.2%
is/2	4.2%
chat:terminal/5	3.5%
$record_bag/1	3.5%
chat:country/10	2.8%
chat:inv_map_list/5	2.4%
chat:database/1	2.0%
functor/3	1.9%
arg/3	1.7%
chat:contains0/2	1.3%
sort/2	1.3%

Details -- chat:inv_map_list/5

Time		Access		Predicate
Self	Siblings	Call	Redo	
		182600		<recursive>
2.4%	4.6%	2300	0	chat:inv_map/4
2.4%	4.6%	2300	0	chat:inv_map_list/5
0.6%	0.0%	243400	0	arg/3
1.3%	1.8%	121700	60900	chat:inv_map/4
0.9%	0.0%	121700	0	is/2

1419 samples in 14.05 sec; 538 predicates; 4547 nodes in call-graph; distortion 10%

Figure 4.1: Execution profiler showing the activity of the predicate chat:inv_map_list/5.

are the callees, with the time spent in the callee itself for serving the current predicate and the time spent in the callees of the callee ('Siblings'), so the whole time-block adds up to the 'Siblings' field of the current predicate. The 'Access' fields show how many times the current predicate accesses each of the callees.

The predicates have a menu that allows changing the view of the detail window to the given caller or callee, showing the documentation (if it is a built-in) and/or jumping to the source.

The statistics shown in the report-field of figure 4.1 show the following information:

- *samples*
  Number of times the call-tree was sampled for collecting time statistics. On most hardware the resolution of `SIGPROF` is 1/100 second. This number must be sufficiently large to get reliable timing figures. The Time menu allows viewing time as samples, relative time or absolute time.

- *sec*
  Total user CPU time with the profiler active.

- *predicates*
  Total count of predicates that have been called at least one time during the profile.

- *nodes*
  Number of nodes in the call-tree.

- *distortion*
  How much of the time is spend building the call-tree as a percentage of the total execution time. Timing samples while the profiler is building the call-tree are not added to the call-tree.

4.39.3 Information gathering

While the program executes under the profiler, the system builds a *dynamic* call-tree. It does this using three hooks from the kernel: one that starts a new goal (*profCall*), one the tells the system which goal is resumed after an *exit* (*profExit*) and one that tells the system which goal is resumed after a *fail* (i.e. which goal is used to *retry* (*profRedo*)). The profCall() function finds or creates the subnode for the argument predicate below the current node, increments the call-count of this link and returns the subnode which is recorded in the Prolog stack-frame. Choice-points are marked with the current profiling node. profExit() and profRedo() pass the profiling node where execution resumes.

Just using the above algorithm would create a much too big tree due to recursion. For this reason the system performs detection of recursion. In the simplest case, recursive procedures increment the 'recursive' count on the current node. Mutual recursion however is not easily detected. For example, `call/1` can call a predicate that uses `call/1` itself. This can be viewed as a recursive invocation, but this is generally not desirable. Recursion is currently assumed if the same predicate *with the same parent* appears higher in the call-graph. Early experience with a some arbitrary non-trivial programs are promising.

The last part of the profiler collects statistics on the CPU-time used in each node. On systems providing setitimer() with `SIGPROF`, it 'ticks' the current node of the call-tree each time the timer fires. On Windows a MM-timer in a separate thread checks 100 times per second how much time is spent in the profiled thread and adds this to the current node. See section 4.39.3 for details.

Profiling in the Windows Implementation

Profiling in the Windows version is similar but as profiling is a statistical process it is good to be aware of the implementation[76] for proper interpretation of the results.

Windows does not provide timers that fire asynchronously, frequent and proportional to the CPU time used by the process. Windows does provide multi-media timers that can run at high frequency. Such timers however run in a separate thread of execution and they are fired on the wall-clock rather than the amount of CPU time used. The profiler installs such a timer running, for saving CPU time, rather inaccurately at about 100 Hz. Each time it is fired, it determines the milliseconds CPU time used by Prolog since the last time it was fired. If this value is non-zero, active predicates are incremented with this value.

4.40 Memory Management

garbage_collect

Invoke the global- and trail stack garbage collector. Normally the garbage collector is invoked automatically if necessary. Explicit invocation might be useful to reduce the need for garbage collections in time critical segments of the code. After the garbage collection `trim_stacks/0` is invoked to release the collected memory resources.

garbage_collect_atoms

Reclaim unused atoms. Normally invoked after `agc_margin` (a Prolog flag) atoms have been created. On multi-threaded versions the actual collection is delayed until there there are no threads performing normal garbage collection. In this case `garbage_collect_atoms/0` returns immediately. Note this implies there is no guarantee it will *ever* happen as there may always be threads performing garbage collection.

trim_stacks

Release stack memory resources that are not in use at this moment, returning them to the operating system. It can be used to release memory resources in a backtracking loop, where the iterations require typically seconds of execution time and very different, potentially large, amounts of stack space. Such a loop can be written as follows:

---

[76] We hereby acknowledge Lionel Fourquaux, who suggested the design described here after a newsnet enquiry.

```
loop :-
        generator,
            trim_stacks,
            potentially_expensive_operation,
        stop_condition, !.
```

The prolog top level loop is written this way, reclaiming memory resources after every user query.

set_prolog_stack(+*Stack*, +*KeyValue***)**

Set a parameter for one of the Prolog runtime stacks. *Stack* is one of `local`, `global`, `trail` or `argument`. The table below describes the *Key*(argValue) pairs. *Value* can be an arithmetic integer expression. E.g., to specify a 2Gb limit for the global stack one can use:

```
?- set_prolog_stack(global, limit(2*10**9)).
```

Current settings can be retrieved with `prolog_stack_property/2`.

limit(+*Bytes***)**

Set the limit to which the stack is allowed to grow. If the specified value is lower than the current usage a permission_error is raised. If the limit is larger than supported, the system silently reduces the requested limit to the system limit.

min_free(+*Cells***)**

Minimum amount of free space after trimming or shifting the stack. Setting this value higher can reduce the number of garbage collections and stack-shifts at the cost of higher memory usage. The spare stack amount is reported and specified in 'cells'. A cell is 4 bytes in the 32-bit version and 8-bytes on the 64-bit version. See `address_bits`. See also `trim_stacks/0` and `debug/0`.

spare(+*Cells***)**

All stacks trigger overflow before actually reaching the limit, so the resulting error can be handled gracefully. The spare stack is used for `print_message/2` from the garbage collector and for handling exceptions. The default suffices, unless the user redefines related hooks. Do **not** specify large values for this because it reduces the amount of memory available for your real task.

Related hooks are: `message_hook/3` (redefining GC messages), `prolog_trace_interception/4` and `prolog_exception_hook/4`.

prolog_stack_property(?*Stack*, ?*KeyValue***)**

True if *KeyValue* is a current property of *Stack*. See `set_prolog_stack/2` for defined properties.

4.41 Windows DDE interface

The predicates in this section deal with MS-Windows 'Dynamic Data Exchange' or DDE protocol.[77] A Windows DDE conversation is a form of interprocess communication based on sending reserved window-events between the communicating processes.

Failing DDE operations raise an error of the structure below, where *Operation* is the name of the (partial) operation that failed and *Message* is a translation of the operator error code. For some errors, *Context* provides additional comments.

```
        error(dde_error(Operation, Message), Context)
```

4.41.1 DDE client interface

The DDE client interface allows Prolog to talk to DDE server programs. We will demonstrate the use of the DDE interface using the Windows PROGMAN (Program Manager) application:

```
1 ?- open_dde_conversation(progman, progman, C).

C = 0
2 ?- dde_request(0, groups, X)

--> Unifies X with description of groups

3 ?- dde_execute(0, '[CreateGroup("DDE Demo")]').

Yes

4 ?- close_dde_conversation(0).

Yes
```

For details on interacting with `progman`, use the SDK online manual section on the Shell DDE interface. See also the Prolog `library(progman)`, which may be used to write simple Windows setup scripts in Prolog.

open_dde_conversation(+*Service*, +*Topic*, -*Handle***)**
Open a conversation with a server supporting the given service name and topic (atoms). If successful, *Handle* may be used to send transactions to the server. If no willing server is found this predicate fails silently.

close_dde_conversation(+*Handle***)**
Close the conversation associated with *Handle*. All opened conversations should be closed when they're no longer needed, although the system will close any that remain open on process termination.

---

[77] This interface is contributed by Don Dwiggins.

dde_request(+*Handle*, +*Item*, -*Value***)**
Request a value from the server. *Item* is an atom that identifies the requested data, and *Value* will be a string (`CF_TEXT` data in DDE parlance) representing that data, if the request is successful.

dde_execute(+*Handle*, +*Command***)**
Request the DDE server to execute the given command-string. Succeeds if the command could be executed and fails with error message otherwise.

dde_poke(+*Handle*, +*Item*, +*Command***)**
Issue a `POKE` command to the server on the specified *Item*. Command is passed as data of type `CF_TEXT`.

4.41.2 DDE server mode

The `library(dde)` defines primitives to realise simple DDE server applications in SWI-Prolog. These features are provided as of version 2.0.6 and should be regarded prototypes. The C-part of the DDE server can handle some more primitives, so if you need features not provided by this interface, please study `library(dde)`.

dde_register_service(+*Template*, +*Goal***)**
Register a server to handle DDE request or DDE execute requests from other applications. To register a service for a DDE request, *Template* is of the form:

+Service(+Topic, +Item, +Value)

Service is the name of the DDE service provided (like `progman` in the client example above). *Topic* is either an atom, indicating *Goal* only handles requests on this topic or a variable that also appears in *Goal*. *Item* and *Value* are variables that also appear in *Goal*. *Item* represents the request data as a Prolog atom.[78]

The example below registers the Prolog `current_prolog_flag/2` predicate to be accessible from other applications. The request may be given from the same Prolog as well as from another application.

```
?- dde_register_service(prolog(current_prolog_flag, F, V),
                        current_prolog_flag(F, V)).

?- open_dde_conversation(prolog, current_prolog_flag, Handle),
   dde_request(Handle, home, Home),
   close_dde_conversation(Handle).

Home = '/usr/local/lib/pl-2.0.6/'
```

Handling DDE `execute` requests is very similar. In this case the template is of the form:

+Service(+Topic, +Item)

---

[78] Up-to version 3.4.5 this was a list of character codes. As recent versions have atom garbage collection there is no need for this anymore.

Passing a *Value* argument is not needed as execute requests either succeed or fail. If *Goal* fails, a ‘not processed’ is passed back to the caller of the DDE request.

dde_unregister_service(+*Service***)**
Stop responding to *Service*. If Prolog is halted, it will automatically call this on all open services.

dde_current_service(-*Service, -Topic***)**
Find currently registered services and the topics served on them.

dde_current_connection(-*Service, -Topic***)**
Find currently open conversations.

4.42 Miscellaneous

dwim_match(+*Atom1, +Atom2***)**
True if *Atom1* matches *Atom2* in ‘Do What I Mean’ sense. Both *Atom1* and *Atom2* may also be integers or floats. The two atoms match if:

- They are identical
- They differ by one character (spy ≡ spu)
- One character is inserted/deleted (debug ≡ deug)
- Two characters are transposed (trace ≡ tarce)
- ‘Sub-words’ are glued differently (existsfile ≡ existsFile ≡ exists_file)
- Two adjacent sub words are transposed (existsFile ≡ fileExists)

dwim_match(+*Atom1, +Atom2, -Difference***)**
Equivalent to `dwim_match/2`, but unifies *Difference* with an atom identifying the difference between *Atom1* and *Atom2*. The return values are (in the same order as above): `equal`, `mismatched_char`, `inserted_char`, `transposed_char`, `separated` and `transposed_word`.

wildcard_match(+*Pattern, +String***)**
True if *String* matches the wildcard pattern *Pattern*. *Pattern* is very similar the Unix csh pattern matcher. The patterns are given below:

`?`	Matches one arbitrary character.
`*`	Matches any number of arbitrary characters.
`[...]`	Matches one of the characters specified between the brackets. ⟨*char1*⟩-⟨*char2*⟩ indicates a range.
`{...}`	Matches any of the patterns of the comma separated list between the braces.

Example:

```
?- wildcard_match('[a-z]*.{pro,pl}[%~]', 'a_hello.pl%').

Yes
```

sleep(+*Time***)**

Suspend execution *Time* seconds. *Time* is either a floating point number or an integer. Granularity is dependent on the system's timer granularity. A negative time causes the timer to return immediately. On most non-realtime operating systems we can only ensure execution is suspended for **at least** *Time* seconds.

On Unix systems the `sleep/1` predicate is realised —in order of preference— by nanosleep(), usleep(), select() if the time is below 1 minute or sleep(). On Windows systems Sleep() is used.

5

Modules

A Prolog module is a collection of predicates which defines a public interface by means of a set of provided predicates and operators. Prolog modules are defined by an ISO standard. Unfortunately, the standard is considered a failure and, as far as we are aware, not implemented by any concrete Prolog implementation. The SWI-Prolog module system is derived from the Quintus Prolog module system. The Quintus module system has been the starting point for the module systems of a number of mainstream Prolog systems, such as SICStus, Ciao and YAP.

This chapter motivates and describes the SWI-Prolog module system. Novices can start using the module system after reading section 5.2 and section 5.3. The primitives defined in these sections suffice for basic usage until one needs to export predicates that call or manage other predicates dynamically (e.g., use `call/1`, `assert/1`, etc.). Such predicates are called *meta predicates* and are discussed in section 5.4. Section 5.5 to section 5.8 describe more advanced issues. Starting with section 5.9, we discuss more low-level aspects of the SWI-Prolog module systems that are used to implement the visible module system, and can be used to build other code reuse mechanisms.

5.1 Why Use Modules?

In classic Prolog systems, all predicates are organised in a single namespace and any predicate can call any predicate. Because each predicate in a file can be called from anywhere in the program, it becomes very hard to find the dependencies and enhance the implementation of a predicate without risking to break the overall application. This is true for any language, but even worse for Prolog due to its frequent need for 'helper predicates'.

A Prolog module encapsulates a set of predicates and defines an *interface*im-3115(x(are)344.132]TJ 0 -13.549 T314

```
:- module(reverse, [reverse/2]).

reverse(List1, List2) :-
        rev(List1, [], List2).

rev([], List, List).
rev([Head|List1], List2, List3) :-
        rev(List1, [Head|List2], List3).
```

The module is named `reverse`. Typically, the name of a module is the same as the name of the file by which it is defined without the filename extension, but this naming is not enforced. Modules are organised in a single and flat namespace and therefore module names must be chosen with some care to avoid conflicts. As we will see, typical applications of the module system rarely use the name of a module explicitly in the source text.

:- module(+*Module*, +*PublicList***)**

This directive can only be used as the first term of a source file. It declares the file to be a *module file*, defining a module named *Module*. Note that a module name is an atom. The module exports the predicates of *PublicList*. *PublicList* is a list of predicate indicators (name/arity or name//arity pairs) or operator declarations using the format `op`(*Precedence, Type, Name*). Operators defined in the export list are available inside the module as well as to modules importing this module. See also section 4.24.

Compatible to Ciao Prolog, if *Module* is unbound, it is unified with the basename without extension of the file being loaded.

5.3 Importing Predicates into a Module

Predicates can be added to a module by *importing* them from another module. Importing adds predicates to the namespace of a module. An imported predicate can be called exactly the same as a locally defined predicate, although its implementation remains part of the module in which it has been defined.

Importing the predicates from another module is achieved using the directives `use_module/1` or `use_module/2`. Note that both directives take *file name(s)* as arguments. I.e., modules are imported based on their file name rather than their module name.

use_module(+*Files***)**

Load the file(s) specified with *File* just like `ensure_loaded/1`. The files must all be module files. All exported predicates from the loaded files are imported into the module from which this predicate is called. This predicate is equivalent to `ensure_loaded/1`, except that it raises an error if *File* is not a module file.

use_module(+*File*, +*ImportList***)**

Load *File*, which must be a module file and import the predicates as specified by *ImportList*. *ImportList* is a list of predicate indicators specifying the predicates that will be imported from the loaded module. *ImportList* also allows for renaming or import-everything-except. See also `import` option of `load_files/2`. The first example below loads `member/2` from the `lists` library and `append/2` under the name `list_concat`, which is how this predicate is named in YAP. The second example loads all exports from library `option`, except for `meta_options/3`. These renaming facilities are generally used to deal with portability issues with as few as possible changes to the actual code. See also section C and section 5.7.

```
:- use_module(library(lists), [ member/2,
                                 append/2 as list_concat
                               ]).
:- use_module(library(option), except([meta_options/3])).
```

The `module/2` directive, `use_module/1` and `use_module/2` are sufficient to partition a simple Prolog program into modules. The SWI-Prolog graphical cross-referencing tool `gxref/0` can be used to analyse the dependencies between non-module files and propose module declarations for each file.

5.4 Defining a meta-predicate

A meta-predicate is a predicate that calls other predicates dynamically, modifies a predicate or reasons about properties of a predicate. Such predicates use either a compound term or a *predicate indicator* to describe the predicate they address, e.g., `assert(name(jan))` or `abolish(name/1)`. With modules, this simple schema no longer works as each module defines its own mapping from name+arity to predicate. This is resolved by wrapping the original description in a term ⟨*module*⟩:⟨*term*⟩, e.g., `assert(person:name(jan))` or `abolish(person:name/1)`.

Of course, calling `assert/1` from inside a module, we expect to assert to a predicate local to this module. In other words, we do not wish to provide this `:/2` wrapper by hand. The `meta_predicate/1` directive tells the compiler that certain arguments are terms that will be used to lookup a predicate and thus need to be wrapped (qualified) with ⟨*module*⟩:⟨*term*⟩, unless they are already wrapped.

In the example below, we use this to define `maplist/3` inside a module. The argument '2' in the meta_predicate declaration means that the argument is module sensitive and refers to a predicate with an arity that is two more than the term that is passed in. The compiler only distinguishes the values 0..9 and `:`, which denote module-sensitive arguments, from `+`, `-` and `?`, which denote *modes*. The values 0..9 are used by the *cross-referencer* and syntax highlighting. Note that the helper predicate `maplist_/3` does not need to be declared as a meta-predicate because the `maplist/3` wrapper already ensures that *Goal* is qualified as ⟨*module*⟩:*Goal*. See the description of `meta_predicate/1` for details.

```
:- module(maplist, [maplist/3]).
:- meta_predicate maplist(2, ?, ?).

%%      maplist(:Goal, +List1, ?List2)
%
%       True if Goal can successfully be applied to all
%       successive pairs of elements from List1 and List2.
```

```
maplist(Goal, L1, L2) :-
        maplist_(L1, L2, Goal).

maplist_([], [], _).
maplist_([H0|T0], [H|T], Goal) :-
        call(Goal, H0, H),
        maplist_(T0, T, Goal).
```

meta_predicate +*Head, ...*

Define the predicates referenced by the comma-separated list *Head* as *meta-predicates*. Each argument of each head is a *meta argument specifier*. Defined specifiers are given below. Only 0..9 and `:` are interpreted; the mode declarations `+`, `-` and `?` are ignored.

0..9

The argument is a term that is used to reference a predicate with N more arguments than the given argument term. For example: `call(0)` or `maplist(1, +)`.

:

The argument is module sensitive, but does not directly refer to a predicate. For example: `consult(:)`.

-

The argument is not module sensitive and unbound on entry.

?

The argument is not module sensitive and the mode is unspecified.

+

The argument is not module sensitive and bound (i.e., nonvar) on entry.

^

This extension is used to denote the possibly ^-annotated goal of `setof/3`, `bagof/3`, `aggregate/3` and `aggregate/4`. It is processed similar to '0', but leaving the ^/2 intact.

Each argument that is module sensitive (i.e., marked 0..9, `:` or ^) is qualified with the context module of the caller if it is not already qualified. The implementation ensures that the argument is passed as ⟨*module*⟩:⟨*term*⟩, where ⟨*module*⟩ is an atom denoting the name of a module and ⟨*term*⟩ itself is not a `:/2` term where the first argument is an atom. Below is a simple declaration and a number of queries.

```
:- meta_predicate
        meta(0, +).

meta(Module:Term, _Arg) :-
        format('Module=~w, Term = ~q~n', [Module, Term]).
```

```
?- meta(test, x).
Module=user, Term = test
```

```
?- meta(m1:test, x).
Module=m1, Term = test
?- m2:meta(test, x).
Module=m2, Term = test
?- m1:meta(m2:test, x).
Module=m2, Term = test
?- meta(m1:m2:test, x).
Module=m2, Term = test
?- meta(m1:42:test, x).
Module=42, Term = test
```

The `meta_predicate/1` declaration is the portable mechanism for defining meta-predicates and replaces the old SWI-Prolog specific mechanism provided by the deprecated predicates `module_transparent/1`, `context_module/1` and `strip_module/3`. See also section 5.15.

5.5 Overruling Module Boundaries

The module system described so far is sufficient to distribute programs over multiple modules. There are, however, cases in which we would like to be able to overrule this schema and explicitly call a predicate in some module or assert explicitly into some module. Calling in a particular module is useful for debugging from the user's top-level or to access multiple implementations of the same interface that reside in multiple modules. Accessing the same interface from multiple modules cannot be achieved using importing because importing a predicate with the same name and arity from two modules results in a name conflict. Asserting in a different module can be used to create models dynamically in a new module. See section 5.12.

Direct addressing of modules is achieved using a `:/2` explicitly in a program and relies on the module qualification mechanism described in section 5.4. Here are a few examples:

```
?- assert(world:done).  % asserts done/0 into module world
?- world:assert(done).  % the same
?- world:done.          % calls done/0 in module world
```

5.6 Interacting with modules from the toplevel

Debugging often requires interaction with predicates that reside in modules: running them, setting spy-points on them, etc. This can be achieved using the ⟨*module*⟩:⟨*term*⟩ construct explicitly as described above. In SWI-Prolog, you may also wish to omit the module qualification. Setting a spy-point (`spy/1`) on a plain predicate sets a spy-point on any predicate with that name in any module. Editing (`edit/1`) or calling an unqualified predicate invokes the DWIM (Do What I Mean) mechanism, which generally suggests the correct qualified query.

Mainly for compatibility, we provide `module/1` to switch the module with which the interactive toplevel interacts:

module(+*Module***)**

The call `module(`*Module*`)` may be used to switch the default working module for the interactive toplevel (see `prolog/0`). This may be used when debugging a module. The example below lists the clauses of `file_of_label/2` in the module `tex`.

```
1 ?- module(tex).

Yes
tex: 2 ?- listing(file_of_label/2).
...
```

5.7 Composing modules from other modules

The predicates in this section are intended to create new modules from the content of other modules. Below is an example to define a *composite* module. The example exports all public predicates of `module_1`, `module_2` and `module_3`, `pred/1` from `module_4`, all predicates from `module_5` except `do_not_use/1` and all predicates from `module_6` while renaming `pred/1` into `mypred/1`.

```
:- module(my_composite, []).
:- reexport([ module_1,
              module_2,
              module_3
            ]).
:- reexport(module_4, [ pred/1 ]).
:- reexport(module_5, except([do_not_use/1])).
:- reexport(module_6, except([pred/1 as mypred])).
```

reexport(+*Files***)**

Load and import predicates as `use_module/1` and re-export all imported predicates. The reexport declarations must immediately follow the module declaration.

reexport(+*File*, +*Import***)**

Import from *File* as `use_module/2` and re-export the imported predicates. The reexport declarations must immediately follow the module declaration.

5.8 Operators and modules

Operators (section 4.24) are local to modules, where the initial table behaves as if it is copied from the module `user` (see section 5.10). A specific operator can be disabled inside a module using `:- op(0, Type, Name).` Inheritance from the public table can be restored using `:- op(-1, Type, Name).`

In addition to using the `op/3` directive, operators can be declared in the `module/2` directive as shown below. Such operator declarations are visible inside the module, and importing such a module makes the operators visible in the target module. Exporting operators is typically used by modules that implement sub-languages such as chr (see chapter 7). The example below is copied from the library `clpfd`.

```
:- module(clpfd,
          [ op(760, yfx, #<==>),
            op(750, xfy, #==>),
            op(750, yfx, #<==),
            op(740, yfx, #\/),
            ...
            (#<==>)/2,
            (#==>)/2,
            (#<==)/2,
            (#\/)/2,
            ...
          ]).
```

5.9 Dynamic importing using import modules

Until now we discussed the public module interface that is, at least to some extent, portable between Prolog implementations with a module system that is derived from Quintus Prolog. The remainder of this chapter describes the underlying mechanisms that can be used to emulate other module systems or implement other code-reuse mechanisms.

In addition to built-in predicates, imported predicates and locally defined predicates, SWI-Prolog modules can also call predicates from its *import modules*. Each module has a (possibly empty) list of import modules. In the default setup, each new module has a single import module, which is `user` for all normal user modules and `system` for all system library modules. Module `user` imports from `system` where all built-in predicates reside. These special modules are described in more detail in section 5.10.

The list of import modules can be manipulated and queried using the following predicates, as well as using `set_module/1`.

import_module(+*Module*, -*Import***)** *[nondet]*

True if *Module* inherits directly from *Import*. All normal modules only import from `user`, which imports from `system`. The predicates `add_import_module/3` and `delete_import_module/2` can be used to manipulate the import list. See also `default_module/2`.

default_module(+*Module*, -*Default***)** *[multi]*

True if predicates and operators in *Default* are visible in *Module*. Modules are returned in the same search order used for predicates and operators. That is, *Default* is first unified with *Module*, followed by the depth-first transitive closure of `import_module/2`.

add_import_module(+*Module*, +*Import*, +*StartOrEnd***)**

If *Import* is not already an import module for *Module*, add it to this list at the `start` or `end` depending on *StartOrEnd*. See also `import_module/2` and `delete_import_module/2`.

delete_import_module(+*Module*, +*Import***)**

Delete *Import* from the list of import modules for *Module*. Fails silently if *Import* is not in the list.

One usage scenario of import modules is to define a module that is a copy of another, but where one or more predicates have an alternative definition.

5.10 Reserved Modules and using the ‘user’ module

As mentioned above, SWI-Prolog contains two special modules. The first one is the module `system`. This module contains all built-in predicates. Module `system` has no import module. The second special module is the module `user`. This module forms the initial working space of the user. Initially it is empty. The import module of module `user` is `system`, making all built-in predicates available.

All other modules import from the module `user`. This implies they can use all predicates imported into `user` without explicitly importing them. If an application loads all modules from the `user` module using `use_module/1`, one achieves a scoping system similar to the C-language, where every module can access all exported predicates without any special precautions.

5.11 An alternative import/export interface

The `use_module/1` predicate from section 5.3 defines import and export relations based on the filename from which a module is loaded. If modules are created differently, such as by asserting predicates into a new module as described in section 5.12, this interface cannot be used. The interface below provides for import/export from modules that are not created using a module file.

export(+*PredicateIndicator, . . .***)**

Add predicates to the public list of the context module. This implies the predicate will be imported into another module if this module is imported with `use_module/[1,2]`. Note that predicates are normally exported using the directive `module/2`. `export/1` is meant to handle export from dynamically created modules.

import(+*PredicateIndicator, . . .***)**

Import predicates *PredicateIndicator* into the current context module. *PredicateIndicator* must specify the source module using the ⟨*module*⟩:⟨*pi*⟩ construct. Note that predicates are normally imported using one of the directives `use_module/[1,2]`. The `import/1` alternative is meant for handling imports into dynamically created modules. See also `export/1` and `export_list/2`.

5.12 Dynamic Modules

So far, we discussed modules that were created by loading a module file. These modules have been introduced to facilitate the development of large applications. The modules are fully defined at load-time of the application and normally will not(e)-dev the directi5rom5(Heen)]Tdirecti5ron a39 normallypredicates into a

Dynamic modules can easily be created. Any built-in predicate that tries to locate a predicate in a specific module will create this module as a side-effect if it did not yet exist. For example:

```
?- assert(world_a:consistent),
   world_a:set_prolog_flag(unknown, fail).
```

These calls create a module called 'world_a' and make the call 'world_a:consistent' succeed. Undefined predicates will not raise an exception for this module (see `unknown`).

Import and export from a dynamically created world can be achieved using `import/1` and `export/1` or by specifying the import module as described in section 5.9.

```
?- world_b:export(solve/2).          % exports solve/2 from world_b
?- world_c:import(world_b:solve/2).  % and import it to world_c
```

5.13 Transparent predicates: definition and context module

The 'module-transparent' mechanism is still underlying the actual implementation. Direct usage by programmers is deprecated. Please use `meta_predicate/1` *to deal with meta-predicates.*

The qualification of module-sensitive arguments described in section 5.4 is realised using *transparent* predicates. It is now deprecated to use this mechanism directly. However, studying the underlying mechanism helps to understand SWI-Prolog's modules. In some respect, the transparent mechanism is more powerful than meta-predicate declarations.

Each predicate of the program is assigned a module, called its *definition module*. The definition module of a predicate is always the module in which the predicate was originally defined. Each active goal in the Prolog system has a *context module* assigned to it.

The context module is used to find predicates for a Prolog term. By default, the context module is the definition module of the predicate running the goal. For transparent predicates, however, this is the context module of the goal inherited from the parent goal. Below, we implement `maplist/3` using the transparent mechanism. The code of `maplist/3` and `maplist_/3` is the same as in section 5.4, but now we must declare both the main predicate and the helper as transparent to avoid changing the context module when calling the helper.

```
:- module(maplist, maplist/3).

:- module_transparent
        maplist/3,
        maplist_/3.

maplist(Goal, L1, L2) :-
        maplist_(L1, L2, G).

maplist_([], [], _).
maplist_([H0|T0], [H|T], Goal) :-
        call(Goal, H0, H),
        maplist_(T0, T, Goal).
```

Note that *any* call that translates terms into predicates is subject to the transparent mechanism, not just the terms passed to module-sensitive arguments. For example, the module below counts the number of unique atoms returned as bindings for a variable. It works as expected. If we use the directive `:- module_transparent count_atom_results/3.` instead, `atom_result/2` is called wrongly in the module *calling* `count_atom_results/3`. This can be solved using `strip_module/3` to create a qualified goal and a non-transparent helper predicate that is defined in the same module.

```
:- module(count_atom_results,
          [ count_atom_results/3
          ]).
:- meta_predicate count_atom_results(-,0,-).

count_atom_results(A, Goal, Count) :-
        setof(A, atom_result(A, Goal), As), !,
        length(As, Count).
count_atom_results(_, _, 0).

atom_result(Var, Goal) :-
        call(Goal),
        atom(Var).
```

The following predicates support the module-transparent interface:

:- module_transparent(*+Preds***)**

Preds is a comma-separated list of name/arity pairs (like `dynamic/1`). Each goal associated with a transparent-declared predicate will inherit the *context module* from its parent goal.

context_module(*-Module***)**

Unify *Module* with the context module of the current goal. `context_module/1` itself is, of course, transparent.

strip_module(*+Term, -Module, -Plain***)**

Used in module-transparent predicates or meta-predicates to extract the referenced module and plain term. If *Term* is a module-qualified term, i.e. of the format *Module*:*Plain*, *Module* and *Plain* are unified to these values. Otherwise, *Plain* is unified to *Term* and *Module* to the context module.

5.14 Module properties

The following predicates can be used to query the module system for reflexive programming:

current_module(*?Module***)** *[nondet]*

True if *Module* is a currently defined module. This predicate enumerates all modules, whether loaded from a file or created dynamically. Note that modules cannot be destroyed in the current version of SWI-Prolog.

module_property(*?Module, ?Property***)**
True if *Property* is a property of *Module*. Defined properties are:

class(*-Class***)**
True when *Class* is the class of the module. Defined classes are

user
Default for user-defined modules.

system
Module `system` and modules from ⟨*home*⟩`/boot`.

library
Other modules from the system directories.

test
Modules that create tests.

development
Modules that only support the development environment.

file(*?File***)**
True if *Module* was loaded from *File*.

line_count(*-Line***)**
True if *Module* was loaded from the N-th line of file.

exports(*-ListOfPredicateIndicators***)**
True if *Module* exports the given predicates. Predicate indicators are in canonical form (i.e., always using name/arity and never the DCG form name//arity). Future versions may also use the DCG form and include public operators. See also `predicate_property/2`.

exported_operators(*-ListOfOperators***)**
True if *Module* exports the given operators. Each exported operator is represented as a term `op`(*Pri,Assoc,Name*).

set_module(*:Property***)**
Modify properties of the module. Currently, the following properties may be modified:

base(*+Base***)**
Set the default import module of the current module to *Module*. Typically, *Module* is one of `user` or `system`. See section 5.9.

class(*+Class***)**
Set the class of the module. See `module_property/2`.

5.15 Compatibility of the Module System

The SWI-Prolog module system is largely derived from the Quintus Prolog module system, which is also adopted by SICStus, Ciao and YAP. Originally, the mechanism for defining meta-predicates in SWI-Prolog was based on the `module_transparent/1` directive and `strip_module/3`. Since 5.7.4 it supports the de-facto standard `meta_predicate/1` directive for implementing meta-predicates, providing much better compatibility.

The support for the `meta_predicate/1` mechanism, however, is considerably different. On most systems, the *caller* of a meta-predicate is compiled differently to provide the required ⟨*module*⟩:⟨*term*⟩ qualification. This implies that the meta-declaration must be available to the compiler when compiling code that calls a meta-predicate. In practice, this implies that other systems pose the following restrictions on meta-predicates:

- Modules that provide meta-predicates for a module to be compiled must be loaded explicitly by that module.
- The meta-predicate directives of exported predicates must follow the `module/2` directive immediately.
- After changing a meta-declaration, all modules that *call* the modified predicates need to be recompiled.

In SWI-Prolog, meta-predicates are also *module-transparent*, and qualifying the module-sensitive arguments is done inside the meta-predicate. As a result, the caller need not be aware that it is calling a meta-predicate and none of the above restrictions hold for SWI-Prolog. However, code that aims at portability must obey the above rules.

Other differences are listed below.

- If a module does not define a predicate, it is searched for in the *import modules*. By default, the import module of any user-defined module is the `user` module. In turn, the `user` module imports from the module `system` that provides all built-in predicates. The auto-import hierarchy can be changed using `add_import_module/3` and `delete_import_module/2`.

  This mechanism can be used to realise a simple object-oriented system or hierarchical module system.

- Operator declarations are local to a module and may be exported. In Quintus and SICStus all operators are global. YAP and Ciao also use local operators. SWI-Prolog provides global operator declarations from within a module by explicitly qualifying the operator name with the `user` module.

  ```
  :- op(precedence, type, user:(operatorname)).
  ```

Special Variables and Coroutining

6

This chapter deals with extensions primarily designed to support constraint logic programming (CLP). The low-level attributed variable interface defined in section 6.1 is not intended for the typical Prolog programmer. Instead, the typical Prolog programmer should use the coroutining predicates and the various constraint solvers built on top of attributed variables. CHR (chapter 7) provides a general purpose constraint handling language.

As a rule of thumb, constraint programming reduces the search space by reordering goals and joining goals based on domain knowledge. A typical example is constraint reasoning over integer domains. Plain Prolog has no efficient means to deal with (integer) $X > 0$ and $X < 3$. At best it could translate $X > 0$ with uninstantiated X to `between`(*1, infinite, X*) and a similar primitive for $X < 3$. If the two are combined it has no choice but to generate and test over this infinite two-dimensional space. Instead, a constraint system will *delay* an uninstantiated goal to $X > 0$. If, later, it finds a value for *X* it will execute the test. If it finds $X < 3$ it will combine this knowledge to infer that X is in 1..2 (see below). If it never finds a concrete value for *X* it can be asked to *label X* and produce 1 and 2 on backtracking. See section A.7.

```
1 ?- [library(clpfd)].
...
true.

2 ?- X #> 0, X #< 3.
X in 1..2.
```

Using constraints generally makes your program more *declarative*. There are some caveats though:

- Constraints and cuts do not merge well. A cut after a goal that is delayed prunes the search space before the condition is true.
- Term-copying operations (`assert/1`, `retract/2`, `findall/3`, `copy_term/2`, etc.) generally also copy constraints. The effect varies from ok, silent copying of huge constraint networks to violations of the internal consistency of constraint networks. As a rule of thumb, copying terms holding attributes must be deprecated.

6.1 Attributed variables

Attributed variables provide a technique for extending the Prolog unification algorithm [Holzbaur, 1992] by hooking the binding of attributed variables. There is no consensus in the Prolog community on the exact definition and interface to attributed variables. The SWI-Prolog interface is identical to the one realised by Bart Demoen for hProlog [Demoen, 2002]. This interface is simple and available on all Prolog systems that can run the Leuven CHR system (see chapter 7 and the Leuven CHR page).

Binding an attributed variable schedules a goal to be executed at the first possible opportunity. In the current implementation the hooks are executed immediately after a successful unification of the clause-head or successful completion of a foreign language (built-in) predicate. Each attribute is associated to a module, and the hook (`attr_unify_hook/2`) is executed in this module. The example below realises a very simple and incomplete finite domain reasoner:

```
:- module(domain,
          [ domain/2                    % Var, ?Domain
          ]).
:- use_module(library(ordsets)).

domain(X, Dom) :-
        var(Dom), !,
        get_attr(X, domain, Dom).
domain(X, List) :-
        list_to_ord_set(List, Domain),
        put_attr(Y, domain, Domain),
        X = Y.

%       An attributed variable with attribute value Domain has been
%       assigned the value Y

attr_unify_hook(Domain, Y) :-
        (   get_attr(Y, domain, Dom2)
        ->  ord_intersection(Domain, Dom2, NewDomain),
            (   NewDomain == []
            ->  fail
            ;   NewDomain = [Value]
            ->  Y = Value
            ;   put_attr(Y, domain, NewDomain)
            )
        ;   var(Y)
        ->  put_attr( Y, domain, Domain )
        ;   ord_memberchk(Y, Domain)
        ).

%       Translate attributes from this module to residual goals

attribute_goals(X) -->
        { get_attr(X, domain, List) },
        [domain(X, List)].
```

Before explaining the code we give some example queries:

```
?- domain(X, [a,b]), X = c                  fail
?- domain(X, [a,b]), domain(X, [a,c]).      X = a
?- domain(X, [a,b,c]), domain(X, [a,c]).    domain(X, [a, c])
```

The predicate `domain/2` fetches (first clause) or assigns (second clause) the variable a *domain*, a set of values the variable can be unified with. In the second clause, `domain/2` first associates the domain with a fresh variable (Y) and then unifies X to this variable to deal with the possibility that X already has a domain. The predicate `attr_unify_hook/2` (see below) is a hook called after a variable with a domain is assigned a value. In the simple case where the variable is bound to a concrete value, we simply check whether this value is in the domain. Otherwise we take the intersection of the domains and either fail if the intersection is empty (first example), assign the value if there is only one value in the intersection (second example), or assign the intersection as the new domain of the variable (third example). The nonterminal `attribute_goals/3` is used to translate remaining attributes to user-readable goals that, when executed, reinstate these attributes.

6.1.1 Attribute manipulation predicates

attvar(@*Term***)**

Succeeds if *Term* is an attributed variable. Note that `var/1` also succeeds on attributed variables. Attributed variables are created with `put_attr/3`.

put_attr(+*Var, +Module, +Value***)**

If *Var* is a variable or attributed variable, set the value for the attribute named *Module* to *Value*. If an attribute with this name is already associated with *Var*, the old value is replaced. Backtracking will restore the old value (i.e., an attribute is a mutable term; see also `setarg/3`). This predicate raises a representation error if *Var* is not a variable and a type error if *Module* is not an atom.

get_attr(+*Var, +Module, -Value***)**

Request the current *value* for the attribute named *Module*. If *Var* is not an attributed variable or the named attribute is not associated to *Var* this predicate fails silently. If *Module* is not an atom, a type error is raised.

del_attr(+*Var, +Module***)**

Delete the named attribute. If *Var* loses its last attribute it is transformed back into a traditional Prolog variable. If *Module* is not an atom, a type error is raised. In all other cases this predicate succeeds regardless of whether or not the named attribute is present.

6.1.2 Attributed variable hooks

Attribute names are linked to modules. This means that certain operations on attributed variables cause *hooks* to be called in the module whose name matches the attribute name.

attr_unify_hook(+*AttValue, +VarValue***)**

A hook that must be defined in the module to which an attributed variable refers. It is called *after* the attributed variable has been unified with a non-var term, possibly another attributed variable. *AttValue* is the attribute that was associated to the variable in this module and *VarValue* is the new value of the variable. Normally this predicate fails to veto binding the variable to *VarValue*, forcing backtracking to undo the binding. If *VarValue* is another attributed variable the hook often combines the two attributes and associates the combined attribute with *VarValue* using `put_attr/3`.

attr_portray_hook(+*AttValue*, +*Var***)**

Called by `write_term/2` and friends for each attribute if the option `attributes`(*portray*) is in effect. If the hook succeeds the attribute is considered printed. Otherwise `Module = ...` is printed to indicate the existence of a variable. New infrastructure dealing with communicating attribute values must be based on `copy_term/3` and its hook `attribute_goals//1`.

attribute_goals(+*Var***) //**

This nonterminal, if it is defined in a module, is used by `copy_term/3` to project attributes of that module to residual goals. It is also used by the toplevel to obtain residual goals after executing a query.

6.1.3 Operations on terms with attributed variables

copy_term(+*Term*, -*Copy*, -*Gs***)**

Create a regular term *Copy* as a copy of *Term* (without any attributes), and a list *Gs* of goals that represents the attributes. The goal maplist(call,*Gs*) recreates the attributes for *Copy*. The nonterminal `attribute_goals//1`, as defined in the modules the attributes stem from, is used to convert attributes to lists of goals.

This building block is used by the toplevel to report pending attributes in a portable and understandable fashion. This predicate is the preferred way to reason about and communicate terms with constraints.

copy_term_nat(+*Term*, -*Copy***)**

As `copy_term/2`. Attributes, however, are *not* copied but replaced by fresh variables.

term_attvars(+*Term*, -*AttVars***)**

AttVars is a list of all attributed variables in *Term* and its attributes. That is, `term_attvars/2` works recursively through attributes. This predicate is cycle-safe. The goal `term_attvars`(*Term, []*) in an efficient test that *Term* has *no* attributes; scanning the term is aborted after the first attributed variable is found.

6.1.4 Special purpose predicates for attributes

Normal user code should deal with `put_attr/3`, `get_attr/3` and `del_attr/2`. The routines in this section fetch or set the entire attribute list of a variable. Use of these predicates is anticipated to be restricted to printing and other special purpose operations.

get_attrs(+*Var*, -*Attributes***)**

Get all attributes of *Var*. *Attributes* is a term of the form `att`(*Module, Value, MoreAttributes*), where *MoreAttributes* is `[]` for the last attribute.

put_attrs(+*Var*, -*Attributes***)**

Set all attributes of *Var*. See `get_attrs/2` for a description of *Attributes*.

del_attrs(+*Var***)**

If *Var* is an attributed variable, delete *all* its attributes. In all other cases, this predicate succeeds without side-effects.

6.2 Coroutining

Coroutining deals with having Prolog goals scheduled for execution as soon as some conditions are fulfilled. In Prolog the most commonly used condition is the instantiation (binding) of a variable. Scheduling a goal to execute immediately after a variable is bound can be used to avoid instantiation errors for some built-in predicates (e.g. arithmetic), do work *lazy*, prevent the binding of a variable to a particular value, etc. Using `freeze/2` for example we can define a variable that can only be assigned an even number:

```
?- freeze(X, X mod 2 =:= 0), X = 3

No
```

freeze(+*Var, :Goal***)**

Delay the execution of *Goal* until *Var* is bound (i.e. is not a variable or attributed variable). If *Var* is bound on entry `freeze/2` is equivalent to `call/1`. The `freeze/2` predicate is realised using an attributed variable associated with the module `freeze`. Use `frozen(Var, Goal)` to find out whether and which goals are delayed on *Var*.

frozen(@*Var, -Goal***)**

Unify *Goal* with the goal or conjunction of goals delayed on *Var*. If no goals are frozen on *Var*, *Goal* is unified to `true`.

when(@*Condition, :Goal***)**

Execute *Goal* when *Condition* becomes true. *Condition* is one of `?=`(*X, Y*), `nonvar`(*X*), `ground`(*X*), `,`(*Cond1, Cond2*) or `;`(*Cond1, Cond2*). See also `freeze/2` and `dif/2`. The implementation can deal with cyclic terms in *X* and *Y*.

The `when/2` predicate is realised using attributed variables associated with the module `when`. It is defined in the autoload library `when`.

dif(@*A*, @*B***)**

The `dif/2` predicate provides a constraint stating that *A* and *B* are different terms. If *A* and *B* can never unify, `dif/2` succeeds deterministically. If *A* and *B* are identical it fails immediately, and finally, if *A* and *B* can unify, goals are delayed that prevent *A* and *B* to become equal. The `dif/2` predicate behaves as if defined by `dif(X, Y) :- when(?=(X, Y), X \== Y)`. See also `?=/2`. The implementation can deal with cyclic terms.

The `dif/2` predicate is realised using attributed variables associated with the module `dif`. It is defined in the autoload library `dif`.

call_residue_vars(*:Goal, -Vars***)**

Find residual attributed variables left by *Goal*. This predicate is intended for debugging programs using coroutining or constraints. Consider a program that poses contradicting constraints on a variable. Such programs should fail, but sometimes succeed because the constraint solver is too weak to detect the contradiction. Ideally, delayed goals and constraints are all executed at the end of the computation. The meta predicate `call_residue_vars/2` finds variables that are given attribute variables or whose attributes are modified[1] by *Goal*, regardless of whether or not these variables are reachable from the arguments of *Goal*.

The predicate has considerable implications. During the execution of *Goal*, the garbage collector does not reclaim attributed variables. This causes some degradation of GC performance. In a well-behaved program there are no such variables, so the space impact is generally minimal. The actual collection of *Vars* is implemented using a scan of the trail and global stacks.

6.3 Global variables

Global variables are associations between names (atoms) and terms. They differ in various ways from storing information using `assert/1` or `recorda/3`.

- The value lives on the Prolog (global) stack. This implies that lookup time is independent of the size of the term. This is particularly interesting for large data structures such as parsed XML documents or the CHR global constraint store.
- They support both global assignment using `nb_setval/2` and backtrackable assignment using `b_setval/2`.
- Only one value (which can be an arbitrary complex Prolog term) can be associated to a variable at a time.
- Their value cannot be shared among threads. Each thread has its own namespace and values for global variables.
- Currently global variables are scoped globally. We may consider module scoping in future versions.

Both `b_setval/2` and `nb_setval/2` implicitly create a variable if the referenced name does not already refer to a variable.

Global variables may be initialised from directives to make them available during the program lifetime, but some considerations are necessary for saved states and threads. Saved states do not store global variables, which implies they have to be declared with `initialization/1` to recreate them after loading the saved state. Each thread has its own set of global variables, starting with an empty set. Using `thread_initialization/1` to define a global variable it will be defined, restored after reloading a saved state and created in all threads that are created *after* the registration. Finally, global variables can be initialised using the exception hook `exception/3`. The latter technique is used by CHR (see chapter 7).

b_setval(+*Name*, +*Value***)**

Associate the term *Value* with the atom *Name* or replace the currently associated value with *Value*. If *Name* does not refer to an existing global variable, a variable with initial value `[]` is created (the empty list). On backtracking the assignment is reversed.

---

[1] Tracking modifications is currently not complete and this feature may be dropped completely in future versions.

b_getval(+*Name*, -*Value***)**

Get the value associated with the global variable *Name* and unify it with *Value*. Note that this unification may further instantiate the value of the global variable. If this is undesirable the normal precautions (double negation or `copy_term/2`) must be taken. The `b_getval/2` predicate generates errors if *Name* is not an atom or the requested variable does not exist.

nb_setval(+*Name*, +*Value***)**

Associates a copy of *Value* created with `duplicate_term/2` with the atom *Name*. Note that this can be used to set an initial value other than `[]` prior to backtrackable assignment.

nb_getval(+*Name*, -*Value***)**

The `nb_getval/2` predicate is a synonym for `b_getval/2`, introduced for compatibility and symmetry. As most scenarios will use a particular global variable using either non-backtrackable or backtrackable assignment, using `nb_getval/2` can be used to document that the variable is non-backtrackable.

nb_linkval(+*Name*, +*Value***)**

Associates the term *Value* with the atom *Name* without copying it. This is a fast special-purpose variation of `nb_setval/2` intended for expert users only because the semantics on backtracking to a point before creating the link are poorly defined for compound terms. The principal term is always left untouched, but backtracking behaviour on arguments is undone if the original assignment was *trailed* and left alone otherwise, which implies that the history that created the term affects the behaviour on backtracking. Consider the following example:

```
demo_nb_linkval :-
        T = nice(N),
        (   N = world,
            nb_linkval(myvar, T),
            fail
        ;   nb_getval(myvar, V),
            writeln(V)
        ).
```

nb_current(?*Name*, ?*Value***)**

Enumerate all defined variables with their value. The order of enumeration is undefined.

nb_delete(+*Name***)**

Delete the named global variable.

6.3.1 Compatibility of SWI-Prolog Global Variables

Global variables have been introduced by various Prolog implementations recently. The implementation of them in SWI-Prolog is based on hProlog by Bart Demoen. In discussion with Bart it was decided that the semantics of hProlog `nb_setval/2`, which is equivalent to `nb_linkval/2`, is not acceptable for normal Prolog users as the behaviour is influenced by how built-in predicates that construct terms (`read/1`, =../2, etc.) are implemented.

GNU-Prolog provides a rich set of global variables, including arrays. Arrays can be implemented easily in SWI-Prolog using `functor/3` and `setarg/3` due to the unrestricted arity of compound terms.

CHR: Constraint Handling Rules

7

This chapter is written by Tom Schrijvers, K.U. Leuven, and adjustments by Jan Wielemaker.

The CHR system of SWI-Prolog is the *K.U.Leuven CHR system*. The runtime environment is written by Christian Holzbaur and Tom Schrijvers while the compiler is written by Tom Schrijvers. Both are integrated with SWI-Prolog and licensed under compatible conditions with permission from the authors.

The main reference for the K.U.Leuven CHR system is:

- T. Schrijvers, and B. Demoen, *The K.U.Leuven CHR System: Implementation and Application*, First Workshop on Constraint Handling Rules: Selected Contributions (Frühwirth, T. and Meister, M., eds.), pp. 1–5, 2004.

On the K.U.Leuven CHR website (`http://dtai.cs.kuleuven.be/CHR/`) you can find more related papers, references and example programs.

7.1 Introduction

Constraint Handling Rules (CHR) is a committed-choice rule-based language embedded in Prolog. It is designed for writing constraint solvers and is particularly useful for providing application-specific constraints. It has been used in many kinds of applications, like scheduling, model checking, abduction, and type checking, among many others.

CHR has previously been implemented in other Prolog systems (SICStus, Eclipse, Yap), Haskell and Java. This CHR system is based on the compilation scheme and runtime environment of CHR in SICStus.

In this documentation we restrict ourselves to giving a short overview of CHR in general and mainly focus on elements specific to this implementation. For a more thorough review of CHR we refer the reader to [Frühwirth, 1998]. More background on CHR can be found at [Frühwirth,].

In section 7.2 we present the syntax of CHR in Prolog and explain informally its operational semantics. Next, section 7.3 deals with practical issues of writing and compiling Prolog programs containing CHR. Section 7.4 explains the (currently primitive) CHR debugging facilities. Section 7.4.3 provides a few useful predicates to inspect the constraint store, and section 7.5 illustrates CHR with two example programs. Section 7.6 describes some compatibility issues with older versions of this system and SICStus' CHR system. Finally, section 7.7 concludes with a few practical guidelines for using CHR.

7.2 Syntax and Semantics

7.2.1 Syntax

The syntax of CHR rules is the following:

```
rules --> rule, rules.
rules --> [].

rule --> name, actual_rule, pragma, [atom('.')].

name --> atom, [atom('@')].
name --> [].

actual_rule --> simplification_rule.
actual_rule --> propagation_rule.
actual_rule --> simpagation_rule.

simplification_rule --> head, [atom('<=>')], guard, body.
propagation_rule --> head, [atom('==>')], guard, body.
simpagation_rule --> head, [atom('\')], head, [atom('<=>')],
                     guard, body.

head --> constraints.

constraints --> constraint, constraint_id.
constraints --> constraint, constraint_id, [atom(',')], constraints.

constraint --> compound_term.

constraint_id --> [].
constraint_id --> [atom('#')], variable.
constraint_id --> [atom('#')], [atom('passive')] .

guard --> [].
guard --> goal, [atom('|')].

body --> goal.

pragma --> [].
pragma --> [atom('pragma')], actual_pragmas.

actual_pragmas --> actual_pragma.
actual_pragmas --> actual_pragma, [atom(',')], actual_pragmas.

actual_pragma --> [atom('passive(')], variable, [atom(')')].
```

Note that the guard of a rule may not contain any goal that binds a variable in the head of the rule with a non-variable or with another variable in the head of the rule. It may, however, bind variables that do not appear in the head of the rule, e.g. an auxiliary variable introduced in the guard.

7.2.2 Semantics

In this subsection the operational semantics of CHR in Prolog are presented informally. They do not differ essentially from other CHR systems.

When a constraint is called, it is considered an active constraint and the system will try to apply the rules to it. Rules are tried and executed sequentially in the order they are written.

A rule is conceptually tried for an active constraint in the following way. The active constraint is matched with a constraint in the head of the rule. If more constraints appear in the head, they are looked for among the suspended constraints, which are called passive constraints in this context. If the necessary passive constraints can be found and all match with the head of the rule and the guard of the rule succeeds, then the rule is committed and the body of the rule executed. If not all the necessary passive constraints can be found, or the matching or the guard fails, then the body is not executed and the process of trying and executing simply continues with the following rules. If for a rule there are multiple constraints in the head, the active constraint will try the rule sequentially multiple times, each time trying to match with another constraint.

This process ends either when the active constraint disappears, i.e. it is removed by some rule, or after the last rule has been processed. In the latter case the active constraint becomes suspended.

A suspended constraint is eligible as a passive constraint for an active constraint. The other way it may interact again with the rules is when a variable appearing in the constraint becomes bound to either a non-variable or another variable involved in one or more constraints. In that case the constraint is triggered, i.e. it becomes an active constraint and all the rules are tried.

Rule Types There are three different kinds of rules, each with its specific semantics:

- *simplification*
  The simplification rule removes the constraints in its head and calls its body.
- *propagation*
  The propagation rule calls its body exactly once for the constraints in its head.
- *simpagation*
  The simpagation rule removes the constraints in its head after the \ and then calls its body. It is an optimization of simplification rules of the form:

  $$\mathit{constraints}_1; \mathit{constraints}_2 <=> \mathit{constraints}_1; \mathit{body}$$

  Namely, in the simpagation form:

  $$\mathit{constraints}_1 \backslash \mathit{constraints}_2 <=> \mathit{body}$$

  The $constraints_1$ constraints are not called in the body.

Rule Names Naming a rule is optional and has no semantic meaning. It only functions as documentation for the programmer.

Pragmas The semantics of the pragmas are:

passive(*Identifier***)**
The constraint in the head of a rule *Identifier* can only match a passive constraint in that rule. There is an abbreviated syntax for this pragma. Instead of:

```
                    ..., c # Id, ... <=> ... pragma passive(Id)
```

you can also write

```
                    ..., c # passive, ... <=> ...
```

Additional pragmas may be released in the future.

:- chr_option(+*Option*, +*Value***)**

It is possible to specify options that apply to all the CHR rules in the module. Options are specified with the `chr_option/2` declaration:

```
:- chr_option(Option,Value).
```

and may appear in the file anywhere after the first constraints declaration.

Available options are:

check_guard_bindings

This option controls whether guards should be checked for (illegal) variable bindings or not. Possible values for this option are `on` to enable the checks, and `off` to disable the checks. If this option is on, any guard fails when it binds a variable that appears in the head of the rule. When the option is off (default), the behavior of a binding in the guard is undefined.

optimize

This option controls the degree of optimization. Possible values are `full` to enable all available optimizations, and `off` (default) to disable all optimizations. The default is derived from the SWI-Prolog flag `optimise`, where `true` is mapped to `full`. Therefore the command-line option `-O` provides full CHR optimization. If optimization is enabled, debugging must be disabled.

debug

This option enables or disables the possibility to debug the CHR code. Possible values are `on` (default) and `off`. See section 7.4 for more details on debugging. The default is derived from the Prolog flag `generate_debug_info`, which is `true` by default. See `-nodebug`. If debugging is enabled, optimization must be disabled.

7.3 CHR in SWI-Prolog Programs

7.3.1 Embedding in Prolog Programs

The CHR constraints defined in a `.pl` file are associated with a module. The default module is `user`. One should never load different `.pl` files with the same CHR module name.

7.3.2 Constraint declaration

:- chr_constraint(+*Specifier***)**

Every constraint used in CHR rules has to be declared with a `chr_constraint/1` declaration by the *constraint specifier*. For convenience multiple constraints may be declared at once with the same `chr_constraint/1` declaration followed by a comma-separated list of constraint specifiers.

A constraint specifier is, in its compact form, F/A where F and A are respectively the functor name and arity of the constraint, e.g.:

```
:- chr_constraint foo/1.
:- chr_constraint bar/2, baz/3.
```

In its extended form, a constraint specifier is $c(A_1, \ldots, A_n)$ where c is the constraint's functor, n its arity and the A_i are argument specifiers. An argument specifier is a mode, optionally followed by a type. E.g.

```
:- chr_constraint get_value(+,?).
:- chr_constraint domain(?int, +list(int)),
                  alldifferent(?list(int)).
```

Modes A mode is one of:

-

The corresponding argument of every occurrence of the constraint is always unbound.

+

The corresponding argument of every occurrence of the constraint is always ground.

?

The corresponding argument of every occurrence of the constraint can have any instantiation, which may change over time. This is the default value.

Types A type can be a user-defined type or one of the built-in types. A type comprises a (possibly infinite) set of values. The type declaration for a constraint argument means that for every instance of that constraint the corresponding argument is only ever bound to values in that set. It does not state that the argument necessarily has to be bound to a value.

The built-in types are:

int

The corresponding argument of every occurrence of the constraint is an integer number.

dense_int

The corresponding argument of every occurrence of the constraint is an integer that can be used as an array index. Note that if this argument takes values in $[0; n]$, the array takes $O(n)$ space.

float

. . . a floating point number.

number
. . . a number.

natural
. . . a positive integer.

any
The corresponding argument of every occurrence of the constraint can have any type. This is the default value.

:- chr_type(+*TypeDeclaration***)**
User-defined types are algebraic data types, similar to those in Haskell or the discriminated unions in Mercury. An algebraic data type is defined using `chr_type/1`:

```
:- chr_type type ---> body.
```

If the type term is a functor of arity zero (i.e. one having zero arguments), it names a monomorphic type. Otherwise, it names a polymorphic type; the arguments of the functor must be distinct type variables. The body term is defined as a sequence of constructor definitions separated by semi-colons.

Each constructor definition must be a functor whose arguments (if any) are types. Discriminated union definitions must be transparent: all type variables occurring in the body must also occur in the type.

Here are some examples of algebraic data type definitions:

```
:- chr_type color ---> red ; blue ; yellow ; green.

:- chr_type tree --->  empty ; leaf(int) ; branch(tree, tree).

:- chr_type list(T) ---> [] ; [T | list(T)].

:- chr_type pair(T1, T2) ---> (T1 - T2).
```

Each algebraic data type definition introduces a distinct type. Two algebraic data types that have the same bodies are considered to be distinct types (name equivalence).

Constructors may be overloaded among different types: there may be any number of constructors with a given name and arity, so long as they all have different types.

Aliases can be defined using ==. For example, if your program uses lists of lists of integers, you can define an alias as follows:

```
:- chr_type lli == list(list(int)).
```

Type Checking Currently two complementary forms of type checking are performed:

1. Static type checking is always performed by the compiler. It is limited to CHR rule heads and CHR constraint calls in rule bodies.

   Two kinds of type error are detected. The first is where a variable has to belong to two types. For example, in the program:

   ```
   :-chr_type foo ---> foo.
   :-chr_type bar ---> bar.

   :-chr_constraint abc(?foo).
   :-chr_constraint def(?bar).

   foobar @ abc(X) <=> def(X).
   ```

   the variable `X` has to be of both type `foo` and `bar`. This is reported as a type clash error:

   ```
   CHR compiler ERROR:
       `--> Type clash for variable _G5398 in rule foobar:
                   expected type foo in body goal def(_G5398, _G5448)
                   expected type bar in head def(_G5448, _G5398)
   ```

   The second kind of error is where a functor is used that does not belong to the declared type. For example in:

   ```
   :-chr_type foo ---> foo.
   :-chr_type bar ---> bar.

   :-chr_constraint abc(?foo).

   foo @ abc(bar) <=> true.
   ```

   `bar` appears in the head of the rule where something of type `foo` is expected. This is reported as:

   ```
   CHR compiler ERROR:
       `--> Invalid functor in head abc(bar) of rule foo:
                   found `bar',
                   expected type `foo'!
   ```

   No runtime overhead is incurred in static type checking.

2. Dynamic type checking checks at runtime, during program execution, whether the arguments of CHR constraints respect their declared types. The `when/2` co-routining library is used to delay dynamic type checks until variables are instantiated.

   The kind of error detected by dynamic type checking is where a functor is used that does not belong to the declared type. E.g. for the program:

```
:-chr_type foo ---> foo.

:-chr_constraint abc(?foo).
```

we get the following error in an erroneous query:

```
?- abc(bar).
ERROR: Type error: `foo' expected, found `bar' (CHR Runtime Type Error)
```

Dynamic type checking is weaker than static type checking in the sense that it only checks the particular program execution at hand rather than all possible executions. It is stronger in the sense that it tracks types throughout the whole program.

Note that it is enabled only in debug mode, as it incurs some (minor) runtime overhead.

7.3.3 Compilation

The SWI-Prolog CHR compiler exploits `term_expansion/2` rules to translate the constraint handling rules to plain Prolog. These rules are loaded from the library `chr`. They are activated if the compiled file has the `.chr` extension or after finding a declaration in the following format:

```
:- chr_constraint ...
```

It is advised to define CHR rules in a module file, where the module declaration is immediately followed by including the library(chr) library as exemplified below:

```
:- module(zebra, [ zebra/0 ]).
:- use_module(library(chr)).

:- chr_constraint ...
```

Using this style, CHR rules can be defined in ordinary Prolog .pl files and the operator definitions required by CHR do not leak into modules where they might cause conflicts.

7.4 Debugging

The CHR debugging facilities are currently rather limited. Only tracing is currently available. To use the CHR debugging facilities for a CHR file it must be compiled for debugging. Generating debug info is controlled by the CHR option `debug`, whose default is derived from the SWI-Prolog flag `generate_debug_info`. Therefore debug info is provided unless the `-nodebug` is used.

7.4.1 Ports

For CHR constraints the four standard ports are defined:

call
A new constraint is called and becomes active.

exit
An active constraint exits: it has either been inserted in the store after trying all rules or has been removed from the constraint store.

fail
An active constraint fails.

redo
An active constraint starts looking for an alternative solution.

In addition to the above ports, CHR constraints have five additional ports:

wake
A suspended constraint is woken and becomes active.

insert
An active constraint has tried all rules and is suspended in the constraint store.

remove
An active or passive constraint is removed from the constraint store.

try
An active constraint tries a rule with possibly some passive constraints. The try port is entered just before committing to the rule.

apply
An active constraint commits to a rule with possibly some passive constraints. The apply port is entered just after committing to the rule.

7.4.2 Tracing

Tracing is enabled with the `chr_trace/0` predicate and disabled with the `chr_notrace/0` predicate.

When enabled the tracer will step through the `call`, `exit`, `fail`, `wake` and `apply` ports, accepting debug commands, and simply write out the other ports.

The following debug commands are currently supported:

```
    CHR debug options:

            <cr>    creep           c       creep
            s       skip
            g       ancestors
            n       nodebug
            b       break
            a       abort
            f       fail
            ?       help            h       help
```

Their meaning is:

creep
Step to the next port.

skip
Skip to exit port of this call or wake port.

ancestors
Print list of ancestor call and wake ports.

nodebug
Disable the tracer.

break
Enter a recursive Prolog top-level. See `break/0`.

abort
Exit to the top-level. See `abort/0`.

fail
Insert failure in execution.

help
Print the above available debug options.

7.4.3 CHR Debugging Predicates

The `chr` module contains several predicates that allow inspecting and printing the content of the constraint store.

chr_trace
Activate the CHR tracer. By default the CHR tracer is activated and deactivated automatically by the Prolog predicates `trace/0` and `notrace/0`.

chr_notrace
Deactivate the CHR tracer. By default the CHR tracer is activated and deactivated automatically by the Prolog predicates `trace/0` and `notrace/0`.

chr_leash(+*Spec***)**
Define the set of CHR ports on which the CHR tracer asks for user intervention (i.e. stops). *Spec* is either a list of ports as defined in section 7.4.1 or a predefined 'alias'. Defined aliases are: `full` to stop at all ports, `none` or `off` to never stop, and `default` to stop at the `call`, `exit`, `fail`, `wake` and `apply` ports. See also `leash/1`.

chr_show_store(+*Mod***)**
Prints all suspended constraints of module *Mod* to the standard output. This predicate is automatically called by the SWI-Prolog top-level at the end of each query for every CHR module currently loaded. The Prolog flag `chr_toplevel_show_store` controls whether the top-level shows the constraint stores. The value `true` enables it. Any other value disables it.

find_chr_constraint(-*Constraint***)**
Returns a constraint in the constraint store. Via backtracking, all constraints in the store can be enumerated.

7.5 Examples

Here are two example constraint solvers written in CHR.

- The program below defines a solver with one constraint, `leq/2/`, which is a less-than-or-equal constraint, also known as a partial order constraint.

```
:- module(leq,[leq/2]).
:- use_module(library(chr)).

:- chr_constraint leq/2.
reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

  When the above program is saved in a file and loaded in SWI-Prolog, you can call the `leq/2` constraints in a query, e.g.:

```
?- leq(X,Y), leq(Y,Z).
leq(_G23837, _G23841)
leq(_G23838, _G23841)
leq(_G23837, _G23838)

X = _G23837{leq = ...}
Y = _G23838{leq = ...}
Z = _G23841{leq = ...}

Yes
```

  When the query succeeds, the SWI-Prolog top-level prints the content of the CHR constraint store and displays the bindings generated during the query. Some of the query variables may have been bound to attributed variables, as you see in the above example.

- The program below implements a simple finite domain constraint solver.

```
:- module(dom,[dom/2]).
:- use_module(library(chr)).

:- chr_constraint dom(?int,+list(int)).
:- chr_type list(T) ---> [] ; [T|list(T)].

dom(X,[]) <=> fail.
dom(X,[Y]) <=> X = Y.
dom(X,L) <=> nonvar(X) | memberchk(X,L).
dom(X,L1), dom(X,L2) <=> intersection(L1,L2,L3), dom(X,L3).
```

When the above program is saved in a file and loaded in SWI-Prolog, you can call the `dom/2` constraints in a query, e.g.:

```
?- dom(A,[1,2,3]), dom(A,[3,4,5]).

A = 3

Yes
```

7.6 Backwards Compatibility

7.6.1 The Old SICStus CHR implemenation

There are small differences between the current K.U.Leuven CHR system in SWI-Prolog, older versions of the same system, and SICStus' CHR system.

The current system maps old syntactic elements onto new ones and ignores a number of no longer required elements. However, for each a *deprecated* warning is issued. You are strongly urged to replace or remove deprecated features.

Besides differences in available options and pragmas, the following differences should be noted:

- *The* `constraints/1` *declaration*
  This declaration is deprecated. It has been replaced with the `chr_constraint/1` declaration.

- *The* `option/2` *declaration*
  This declaration is deprecated. It has been replaced with the `chr_option/2` declaration.

- *The* `handler/1` *declaration*
  In SICStus every CHR module requires a `handler/1` declaration declaring a unique handler name. This declaration is valid syntax in SWI-Prolog, but will have no effect. A warning will be given during compilation.

- *The* `rules/1` *declaration*
  In SICStus, for every CHR module it is possible to only enable a subset of the available rules through the `rules/1` declaration. The declaration is valid syntax in SWI-Prolog, but has no effect. A warning is given during compilation.

- *Guard bindings*
  The `check_guard_bindings` option only turns invalid calls to unification into failure. In SICStus this option does more: it intercepts instantiation errors from Prolog built-ins such as `is/2` and turns them into failure. In SWI-Prolog, we do not go this far, as we like to separate concerns more. The CHR compiler is aware of the CHR code, the Prolog system, and the programmer should be aware of the appropriate meaning of the Prolog goals used in guards and bodies of CHR rules.

7.6.2 The Old ECLiPSe CHR implemenation

The old ECLiPSe CHR implementation features a `label_with/1` construct for labeling variables in CHR constraints. This feature has long since been abandoned. However, a simple transformation is all that is required to port the functionality.

```
label_with Constraint1 if Condition1.
...
label_with ConstraintN if ConditionN.
Constraint1 :- Body1.
...
ConstraintN :- BodyN.
```

is transformed into

```
:- chr_constraint my_labeling/0.

my_labeling \ Constraint1 <=> Condition1 | Body1.
...
my_labeling \ ConstraintN <=> ConditionN | BodyN.
my_labeling <=> true.
```

Be sure to put this code after all other rules in your program! With `my_labeling/0` (or another predicate name of your choosing) the labeling is initiated, rather than ECLiPSe's `chr_labeling/0`.

7.7 Programming Tips and Tricks

In this section we cover several guidelines on how to use CHR to write constraint solvers and how to do so efficiently.

- *Check guard bindings yourself*
  It is considered bad practice to write guards that bind variables of the head and to rely on the system to detect this at runtime. It is inefficient and obscures the working of the program.

- *Set semantics*
  The CHR system allows the presence of identical constraints, i.e. multiple constraints with the same functor, arity and arguments. For most constraint solvers, this is not desirable: it affects efficiency and possibly termination. Hence appropriate simpagation rules should be added of the form:
  $$constraint \backslash constraint <=> true$$

- *Multi-headed rules*
  Multi-headed rules are executed more efficiently when the constraints share one or more variables.

- *Mode and type declarations*
  Provide mode and type declarations to get more efficient program execution. Make sure to disable debug (-nodebug) and enable optimization (-O).

- *Compile once, run many times*
  Does consulting your CHR program take a long time in SWI-Prolog? Probably it takes the CHR compiler a long time to compile the CHR rules into Prolog code. When you disable optimizations the CHR compiler will be a lot quicker, but you may lose performance. Alternatively, you can just use SWI-Prolog's `qcompile/1` to generate a `.qlf` file once from your `.pl` file. This `.qlf` contains the generated code of the CHR compiler (be it in a binary format). When you consult the `.qlf` file, the CHR compiler is not invoked and consultation is much faster.

- *Finding Constraints*
  The `find_chr_constraint/1` predicate is fairly expensive. Avoid it, if possible. If you must use it, try to use it with an instantiated top-level constraint symbol.

7.8 Compiler Errors and Warnings

In this section we summarize the most important error and warning messages of the CHR compiler.

7.8.1 CHR Compiler Errors

Type clash for variable ... in rule ...

This error indicates an inconsistency between declared types; a variable can not belong to two types. See static type checking.

Invalid functor in head ... of rule ...

This error indicates an inconsistency between a declared type and the use of a functor in a rule. See static type checking.

Cyclic alias definition: ... == ...

You have defined a type alias in terms of itself, either directly or indirectly.

Ambiguous type aliases You have defined two overlapping type aliases.

Multiple definitions for type

You have defined the same type multiple times.

Non-ground type in constraint definition: ...

You have declared a non-ground type for a constraint argument.

Could not find type definition for ...

You have used an undefined type in a type declaration.

Illegal mode/type declaration You have used invalid syntax in a constraint declaration.

Constraint multiply defined There is more than one declaration for the same constraint.

Undeclared constraint ... in head of ...

You have used an undeclared constraint in the head of a rule. This often indicates a misspelled constraint name or wrong number of arguments.

Invalid pragma ... in ... Pragma should not be a variable.

You have used a variable as a pragma in a rule. This is not allowed.

Invalid identifier ... in pragma passive in ...

You have used an identifier in a passive pragma that does not correspond to an identifier in the head of the rule. Likely the identifier name is misspelled.

Unknown pragma ... in ...

You have used an unknown pragma in a rule. Likely the pragma is misspelled or not supported.

Something unexpected happened in the CHR compiler

You have most likely bumped into a bug in the CHR compiler. Please contact Tom Schrijvers to notify him of this error.

Multi-threaded applications 8

SWI-Prolog multithreading is based on standard C-language multithreading support. It is not like *ParLog* or other parallel implementations of the Prolog language. Prolog threads have their own stacks and only share the Prolog *heap*: predicates, records, flags and other global non-backtrackable data. SWI-Prolog thread support is designed with the following goals in mind.

- *Multi-threaded server applications*
  Today's computing services often focus on (internet) server applications. Such applications often have need for communication between services and/or fast non-blocking service to multiple concurrent clients. The shared heap provides fast communication, and thread creation is relatively cheap.[1]

- *Interactive applications*
  Interactive applications often need to perform extensive computation. If such computations are executed in a new thread, the main thread can process events and allow the user to cancel the ongoing computation. User interfaces can also use multiple threads, each thread dealing with input from a distinct group of windows. See also section 8.8.

- *Natural integration with foreign code*
  Each Prolog thread runs in a native thread of the operating system, automatically making them cooperate with *MT-safe* foreign-code. In addition, any foreign thread can create its own Prolog engine for dealing with calling Prolog from C-code.

SWI-Prolog multithreading is based on the POSIX thread standard [Butenhof, 1997] used on most popular systems except for MS-Windows. On Windows it uses the pthread-win32 emulation of POSIX threads mixed with the Windows native API for smoother and faster operation.

8.1 Creating and destroying Prolog threads

thread_create(*:Goal, -Id, +Options***)**

Create a new Prolog thread (and underlying C-thread) and start it by executing *Goal*. If the thread is created successfully, the thread-identifier of the created thread is unified to *Id*. *Options* is a list of options. The currently defined options are below. Stack size options can also take the value `inf` or `infinite`, which is mapped to the maximum stack size supported by the platform.

alias(*AliasName***)**

Associate an 'alias name' with the thread. This name may be used to refer to the thread and remains valid until the thread is joined (see `thread_join/2`).

---

[1] On a dual AMD-Athlon 1600, SWI-Prolog 5.1.0 creates and joins 4,957 threads per second elapsed time.

at_exit(*:AtExit***)**
Register *AtExit* as using `thread_at_exit/1` before entering the thread goal. Unlike calling `thread_at_exit/1` as part of the normal *Goal*, this *ensures* the *Goal* is called. Using `thread_at_exit/1`, the thread may be signalled or run out of resources before `thread_at_exit/1` is reached.

detached(*Bool***)**
If `false` (default), the thread can be waited for using `thread_join/2`. `thread_join/2` must be called on this thread to reclaim all resources associated with the thread. If `true`, the system will reclaim all associated resources automatically after the thread finishes. Please note that thread identifiers are freed for reuse after a detached thread finishes or a normal thread has been joined. See also `thread_join/2` and `thread_detach/1`.

If a detached thread dies due to failure or exception of the initial goal, the thread prints a message using `print_message/2`. If such termination is considered normal, the code must be wrapped using `ignore/1` and/or `catch/3` to ensure successful completion.

global(*K-Bytes***)**
Set the limit to which the global stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the `-G` command-line option.

local(*K-Bytes***)**
Set the limit to which the local stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the `-L` command-line option.

c_stack(*K-Bytes***)**
Set the limit to which the system stack of this thread may grow. The default, minimum and maximum values are system-dependent.[2].

trail(*K-Bytes***)**
Set the limit to which the trail stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the `-T` command-line option.

The *Goal* argument is *copied* to the new Prolog engine. This implies that further instantiation of this term in either thread does not have consequences for the other thread: Prolog threads do not share data from their stacks.

thread_self(*-Id***)**
Get the Prolog thread identifier of the running thread. If the thread has an alias, the alias name is returned.

thread_join(*+Id, -Status***)**
Wait for the termination of the thread with the given *Id*. Then unify the result status of the thread with *Status*. After this call, *Id* becomes invalid and all resources associated with the thread are reclaimed. Note that threads with the attribute `detached`(*true*) cannot be joined. See also `thread_property/2`.

A thread that has been completed without `thread_join/2` being called on it is partly reclaimed: the Prolog stacks are released and the C-thread is destroyed. A small data structure representing the exit status of the thread is retained until `thread_join/2` is called on the thread. Defined values for *Status* are:

---

[2] Older versions used `stack`. This is still accepted as a synonym.

true

The goal has been proven successfully.

false

The goal has failed.

exception(*Term*)

The thread is terminated on an exception. See `print_message/2` to turn system exceptions into readable messages.

exited(*Term*)

The thread is terminated on `thread_exit/1` using the argument *Term*.

thread_detach(+*Id*)

Switch thread into detached state (see `detached`(*Bool*) option at `thread_create/3`) at runtime. *Id* is the identifier of the thread placed in detached state. This may be the result of `thread_self/1`.

One of the possible applications is to simplify debugging. Threads that are created as *detached* leave no traces if they crash. For non-detached threads the status can be inspected using `thread_property/2`. Threads nobody is waiting for may be created normally and detach themselves just before completion. This way they leave no traces on normal completion and their reason for failure can be inspected.

thread_exit(+*Term*) *[deprecated]*

Terminates the thread immediately, leaving `exited`(*Term*) as result state for `thread_join/2`. If the thread has the attribute `detached`(*true*) it terminates, but its exit status cannot be retrieved using `thread_join/2`, making the value of *Term* irrelevant. The Prolog stacks and C-thread are reclaimed.

The current implementation does not guarantee proper releasing of all mutexes and proper cleanup in `setup_call_cleanup/3`, etc. Please use the exception mechanism (`throw/1`) to abort execution using non-standard control.

thread_initialization(*:Goal*)

Run *Goal* when thread is started. This predicate is similar to `initialization/1`, but is intended for initialization operations of the runtime stacks, such as setting global variables as described in section 6.3. *Goal* is run on four occasions: at the call to this predicate, after loading a saved state, on starting a new thread and on creating a Prolog engine through the C interface. On loading a saved state, *Goal* is executed *after* running the `initialization/1` hooks.

thread_at_exit(*:Goal*)

Run *Goal* just before releasing the thread resources. This is to be compared to `at_halt/1`, but only for the current thread. These hooks are run regardless of why the execution of the thread has been completed. When these hooks are run, the return code is already available through `thread_property/2` using the result of `thread_self/1` as thread identifier. Note that there are two scenarios for using exit hooks. Using `thread_at_exit/1` is typically used if the thread creates a side-effect that must be reverted if the thread dies. Another scenario is where the creator of the thread wants to be informed when the thread ends. That cannot be guaranteed by means of `thread_at_exit/1` because it is possible that the thread cannot be created or dies almost instantly due to a signal or resource error. The `at_exit`(*Goal*) option of `thread_create/3` is designed to deal with this scenario.

thread_setconcurrency(*-Old, +New***)**
Determine the concurrency of the process, which is defined as the maximum number of concurrently active threads. 'Active' here means they are using CPU time. This option is provided if the thread implementation provides pthread_setconcurrency(). Solaris is a typical example of this family. On other systems this predicate unifies *Old* to 0 (zero) and succeeds silently.

8.2 Monitoring threads

Normal multithreaded applications should not need the predicates from this section because almost any usage of these predicates is unsafe. For example checking the existence of a thread before signalling it is of no use as it may vanish between the two calls. Catching exceptions using `catch/3` is the only safe way to deal with thread-existence errors.

These predicates are provided for diagnosis and monitoring tasks. See also section 8.5, describing more high-level primitives.

thread_property(*?Id, ?Property***)**
True if thread *Id* has *Property*. Either or both arguments may be unbound, enumerating all relations on backtracking. Calling `thread_property/2` does not influence any thread. See also `thread_join/2`. For threads that have an alias name, this name is returned in *Id* instead of the opaque thread identifier. Defined properties are:

alias(*Alias***)**
Alias is the alias name of thread *Id*.

detached(*Boolean***)**
Current detached status of the thread.

status(*Status***)**
Current status of the thread. *Status* is one of:

running
The thread is running. This is the initial status of a thread. Please note that threads waiting for something are considered running too.

false
The *Goal* of the thread has been completed and failed.

true
The *Goal* of the thread has been completed and succeeded.

exited(*Term***)**
The *Goal* of the thread has been terminated using `thread_exit/1` with *Term* as argument. If the underlying native thread has exited (using pthread_exit()) *Term* is unbound.

exception(*Term***)**
The *Goal* of the thread has been terminated due to an uncaught exception (see `throw/1` and `catch/3`).

See also `thread_statistics/3` to obtain resource usage information and `message_queue_property/2` to get the number of queued messages for a thread.

thread_statistics(+*Id*, +*Key*, -*Value***)**

Obtains statistical information on thread *Id* as `statistics/2` does in single-threaded applications. This call supports all keys of `statistics/2`, although only stack sizes and CPU time yield different values for each thread.[3]

mutex_statistics

Print usage statistics on internal mutexes and mutexes associated with dynamic predicates. For each mutex two numbers are printed: the number of times the mutex was acquired and the number of *collisions*: the number of times the calling thread has to wait for the mutex. Generally collision count is close to zero on single-CPU hardware.

8.3 Thread communication

8.3.1 Message queues

Prolog threads can exchange data using dynamic predicates, database records, and other globally shared data. These provide no suitable means to wait for data or a condition as they can only be checked in an expensive polling loop. *Message queues* provide a means for threads to wait for data or conditions without using the CPU.

Each thread has a message queue attached to it that is identified by the thread. Additional queues are created using `message_queue_create/1`.

thread_send_message(+*QueueOrThreadId*, +*Term***)**

Place *Term* in the given queue or default queue of the indicated thread (which can even be the message queue of itself, see `thread_self/1`). Any term can be placed in a message queue, but note that the term is copied to the receiving thread and variable bindings are thus lost. This call returns immediately.

If more than one thread is waiting for messages on the given queue and at least one of these is waiting with a partially instantiated *Term*, the waiting threads are *all* sent a wake-up signal, starting a rush for the available messages in the queue. This behaviour can seriously harm performance with many threads waiting on the same queue as all-but-the-winner perform a useless scan of the queue. If there is only one waiting thread or all waiting threads wait with an unbound variable, an arbitrary thread is restarted to scan the queue.[4]

thread_get_message(?*Term***)**

Examines the thread message queue and if necessary blocks execution until a term that unifies to *Term* arrives in the queue. After a term from the queue has been unified to *Term*, the term is deleted from the queue.

Please note that non-unifying messages remain in the queue. After the following has been executed, thread 1 has the term `b`(*gnu*) in its queue and continues execution using *A* = `gnat`.

---

[3] There is no portable interface to obtain thread-specific CPU time and some operating systems provide no access to this information at all. On such systems the total process CPU is returned. Thread CPU time is supported on MS-Windows, Linux and MacOSX.

[4] See the documentation for the POSIX thread functions pthread_cond_signal() v.s. pthread_cond_broadcast() for background information.

```
<thread 1>
thread_get_message(a(A)),

<thread 2>
thread_send_message(Thread_1, b(gnu)),
thread_send_message(Thread_1, a(gnat)),
```

See also `thread_peek_message/1`.

thread_peek_message(*?Term***)**

Examines the thread message queue and compares the queued terms with *Term* until one unifies or the end of the queue has been reached. In the first case the call succeeds, possibly instantiating *Term*. If no term from the queue unifies, this call fails. I.e., `thread_peek_message/1` never waits and does not remove any term from the queue. See also `thread_get_message/3`.

message_queue_create(*?Queue***)**

If *Queue* is an atom, create a named queue. To avoid ambiguity of `thread_send_message/2`, the name of a queue may not be in use as a thread name. If *Queue* is unbound an anonymous queue is created and *Queue* is unified to its identifier.

message_queue_create(*-Queue, +Options***)**

Create a message queue from *Options*. Defined options are:

alias(*+Alias***)**

Same as `message_queue_create`(*Alias*), but according to the ISO draft on Prolog threads.

max_size(*+Size***)**

Maximum number of terms in the queue. If this number is reached, `thread_send_message/2` will suspend until the queue is drained. The option can be used if the source, sending messages to the queue, is faster than the drain, consuming the messages.

message_queue_destroy(*+Queue***)** *[det]*

Destroy a message queue created with `message_queue_create/1`. A permission error is raised if *Queue* refers to (the default queue of) a thread. Other threads that are waiting for *Queue* using `thread_get_message/2` receive an existence error.

thread_get_message(*+Queue, ?Term***)** *[det]*

As `thread_get_message/1`, operating on a given queue. It is allowed (but not advised) to get messages from the queue of other threads. This predicate raises an existence error exception if *Queue* doesn't exist or is destroyed using `message_queue_destroy/1` while this predicate is waiting.

thread_get_message(*+Queue, ?Term, +Options***)** *[semidet]*

As `thread_get_message/2`, but providing additional *Options*:

deadline(+*AbsTime***)**
The call fails (silently) if no message has arrived before *AbsTime*. See `get_time/1` for the representation of absolute time. If *AbsTime* is earlier then the current time, `thread_get_message/3` fails immediately. Both resolution and maximum wait time is platform-dependent.[5]

timeout(+*Time***)**
Time is a float or integer and specifies the maximum time to wait in seconds. This is a relative-time version of the `deadline` option. If both options are provided, the earliest time is effective.

It *Time* is 0 or 0.0, `thread_get_message/3` examines the queue but does not suspend if no matching term is available. Note that unlike `thread_peek_message/2`, a matching term is removed from the queue.

It *Time* < 0, `thread_get_message/3` fails immediately.

thread_peek_message(+*Queue, ?Term***)** *[semidet]*
As `thread_peek_message/1`, operating on a given queue. It is allowed to peek into another thread's message queue, an operation that can be used to check whether a thread has swallowed a message sent to it.

message_queue_property(*?Queue, ?Property***)**
True if *Property* is a property of *Queue*. Defined properties are:

alias(*Alias***)**
Queue has the given alias name.

max_size(*Size***)**
Maximum number of terms that can be in the queue. See `message_queue_create/2`. This property is not present if there is no limit (default).

size(*Size***)**
Queue currently contains *Size* terms. Note that due to concurrent access the returned value may be outdated before it is returned. It can be used for debugging purposes as well as work distribution purposes.

The `size`(*Size*) property is always present and may be used to enumerate the created message queues. Note that this predicate does *not enumerate* threads, but can be used to query the properties of the default queue of a thread.

Explicit message queues are designed with the *worker-pool* model in mind, where multiple threads wait on a single queue and pick up the first goal to execute. Below is a simple implementation where the workers execute arbitrary Prolog goals. Note that this example provides no means to tell when all work is done. This must be realised using additional synchronisation.

```
%%      create_workers(?Id, +N)
%
%       Create a pool with Id and number of workers.  After the pool is
```

---

[5] The implementation uses MsgWaitForMultipleObjects() on MS-Windows and pthread_cond_timedwait() on other systems.

```
%       created, post_job/1 can be used to send jobs to the pool.

create_workers(Id, N) :-
        message_queue_create(Id),
        forall(between(1, N, _),
               thread_create(do_work(Id), _, [])).

do_work(Id) :-
        repeat,
          thread_get_message(Id, Goal),
          (   catch(Goal, E, print_message(error, E))
          ->  true
          ;   print_message(error, goal_failed(Goal, worker(Id)))
          ),
        fail.

%%      post_job(+Id, +Goal)
%
%       Post a job to be executed by one of the pool's workers.

post_job(Id, Goal) :-
        thread_send_message(Id, Goal).
```

8.3.2 Signalling threads

These predicates provide a mechanism to make another thread execute some goal as an *interrupt*. Signalling threads is safe as these interrupts are only checked at safe points in the virtual machine. Nevertheless, signalling in multithreaded environments should be handled with care as the receiving thread may hold a *mutex* (see `with_mutex/2`). Signalling probably only makes sense to start debugging threads and to cancel no-longer-needed threads with `throw/1`, where the receiving thread should be designed carefully to handle exceptions at any point.

thread_signal(+*ThreadId, :Goal***)**

Make thread *ThreadId* execute *Goal* at the first opportunity. In the current implementation, this implies at the first pass through the *Call-port*. The predicate `thread_signal/2` itself places *Goal* into the signalled thread's signal queue and returns immediately.

Signals (interrupts) do not cooperate well with the world of multithreading, mainly because the status of mutexes cannot be guaranteed easily. At the call-port, the Prolog virtual machine holds no locks and therefore the asynchronous execution is safe.

Goal can be any valid Prolog goal, including `throw/1` to make the receiving thread generate an exception, and `trace/0` to start tracing the receiving thread.

In the Windows version, the receiving thread immediately executes the signal if it reaches a Windows GetMessage() call, which generally happens if the thread is waiting for (user-)input.

8.3.3 Threads and dynamic predicates

Besides queues (section 8.3.1) threads can share and exchange data using dynamic predicates. The multithreaded version knows about two types of dynamic predicates. By default, a predicate declared *dynamic* (see `dynamic/1`) is shared by all threads. Each thread may assert, retract and run the dynamic predicate. Synchronisation inside Prolog guarantees the consistency of the predicate. Updates are *logical*: visible clauses are not affected by assert/retract after a query started on the predicate. In many cases primitives from section 8.4 should be used to ensure that application invariants on the predicate are maintained.

Besides shared predicates, dynamic predicates can be declared with the `thread_local/1` directive. Such predicates share their attributes, but the clause list is different in each thread.

thread_local *+Functor/+Arity, ...*

This directive is related to the `dynamic/1` directive. It tells the system that the predicate may be modified using `assert/1`, `retract/1`, etc., during execution of the program. Unlike normal shared dynamic data, however, each thread has its own clause list for the predicate. As a thread starts, this clause list is empty. If there are still clauses when the thread terminates, these are automatically reclaimed by the system (see also `volatile/1`). The thread_local property implies the properties *dynamic* and *volatile*.

Thread-local dynamic predicates are intended for maintaining thread-specific state or intermediate results of a computation.

It is not recommended to put clauses for a thread-local predicate into a file, as in the example below, because the clause is only visible from the thread that loaded the source file. All other threads start with an empty clause list.

```
:- thread_local
        foo/1.

foo(gnat).
```

DISCLAIMER Whether or not this declaration is appropriate in the sense of the proper mechanism to reach the goal is still debated. If you have strong feelings in favour or against, please share them in the SWI-Prolog mailing list.

8.4 Thread synchronisation

All internal Prolog operations are thread-safe. This implies that two Prolog threads can operate on the same dynamic predicate without corrupting the consistency of the predicate. This section deals with user-level *mutexes* (called *monitors* in ADA or *critical-sections* by Microsoft). A mutex is a **MUT**ual **EX**clusive device, which implies that at most one thread can *hold* a mutex.

Mutexes are used to realise related updates to the Prolog database. With ‘related’, we refer to the situation where a ‘transaction’ implies two or more changes to the Prolog database. For example, we have a predicate `address/2`, representing the address of a person and we want to change the address by retracting the old and asserting the new address. Between these two operations the database is invalid: this person has either no address or two addresses, depending on the assert/retract order.

Here is how to realise a correct update:

```
:- initialization
        mutex_create(addressbook).

change_address(Id, Address) :-
        mutex_lock(addressbook),
        retractall(address(Id, _)),
        asserta(address(Id, Address)),
        mutex_unlock(addressbook).
```

mutex_create(*?MutexId***)**

Create a mutex. If *MutexId* is an atom, a *named* mutex is created. If it is a variable, an anonymous mutex reference is returned. There is no limit to the number of mutexes that can be created.

mutex_create(*-MutexId, +Options***)**

Create a mutex using options. Defined options are:

alias(*Alias***)**

Set the alias name. Using `mutex_create`(*X, [alias(name)]*) is preferred over the equivalent `mutex_create`(*name*).

mutex_destroy(*+MutexId***)**

Destroy a mutex. After this call, *MutexId* becomes invalid and further references yield an `existence_error` exception.

with_mutex(*+MutexId, :Goal***)**

Execute *Goal* while holding *MutexId*. If *Goal* leaves choice-points, these are destroyed (as in `once/1`). The mutex is unlocked regardless of whether *Goal* succeeds, fails or raises an exception. An exception thrown by *Goal* is re-thrown after the mutex has been successfully unlocked. See also `mutex_create/1` and `setup_call_cleanup/3`.

Although described in the thread section, this predicate is also available in the single-threaded version, where it behaves simply as `once/1`.

mutex_lock(*+MutexId***)**

Lock the mutex. Prolog mutexes are *recursive* mutexes: they can be locked multiple times by the same thread. Only after unlocking it as many times as it is locked does the mutex become available for locking by other threads. If another thread has locked the mutex the calling thread is suspended until the mutex is unlocked.

If *MutexId* is an atom, and there is no current mutex with that name, the mutex is created automatically using `mutex_create/1`. This implies named mutexes need not be declared explicitly.

Please note that locking and unlocking mutexes should be paired carefully. Especially make sure to unlock mutexes even if the protected code fails or raises an exception. For most common cases, use `with_mutex/2`, which provides a safer way for handling Prolog-level mutexes. The predicate `setup_call_cleanup/3` is another way to guarantee that the mutex is unlocked while retaining non-determinism.

mutex_trylock(+*MutexId***)**

As `mutex_lock/1`, but if the mutex is held by another thread, this predicates fails immediately.

mutex_unlock(+*MutexId***)**

Unlock the mutex. This can only be called if the mutex is held by the calling thread. If this is not the case, a `permission_error` exception is raised.

mutex_unlock_all

Unlock all mutexes held by the current thread. This call is especially useful to handle thread termination using `abort/0` or exceptions. See also `thread_signal/2`.

mutex_property(*?MutexId, ?Property***)**

True if *Property* is a property of *MutexId*. Defined properties are:

alias(*Alias***)**

Mutex has the defined alias name. See `mutex_create/2` using the 'alias' option.

status(*Status***)**

Current status of the mutex. One of `unlocked` if the mutex is currently not locked, or `locked`(*Owner, Count*) if mutex is locked *Count* times by thread *Owner*. Note that unless *Owner* is the calling thread, the locked status can change at any time. There is no useful application of this property, except for diagnostic purposes.[6]

8.5 Thread support library(threadutil)

This library defines a couple of useful predicates for demonstrating and debugging multithreaded applications. This library is certainly not complete.

threads

Lists all current threads and their status.

join_threads

Join all terminated threads. For normal applications, dealing with terminated threads must be part of the application logic, either detaching the thread before termination or making sure it will be joined. The predicate `join_threads/0` is intended for interactive sessions to reclaim resources from threads that died unexpectedly during development.

interactor

Create a new console and run the Prolog top-level in this new console. See also `attach_console/0`. In the Windows version a new interactor can also be created from the Run/New thread menu.

8.5.1 Debugging threads

Support for debugging threads is still very limited. Debug and trace mode are flags that are local to each thread. Individual threads can be debugged either using the graphical debugger described in section 3.5 (see `tspy/1` and friends) or by attaching a console to the thread and running the

---

[6] BUG: As *Owner* and *Count* are fetched separately from the mutex, the values may be inconsistent.

traditional command-line debugger (see `attach_console/0`). When using the graphical debugger, the debugger must be *loaded* from the main thread (for example using guitracer) before `gtrace/0` can be called from a thread.

attach_console

If the current thread has no console attached yet, attach one and redirect the user streams (input, output, and error) to the new console window. On Unix systems the console is an `xterm` application. On Windows systems this requires the GUI version `swipl-win.exe` rather than the console-based `swipl.exe`.

This predicate has a couple of useful applications. One is to separate (debugging) I/O of different threads. Another is to start debugging a thread that is running in the background. If thread 10 is running, the following sequence starts the tracer on this thread:

```
?- thread_signal(10, (attach_console, trace)).
```

tdebug(+*ThreadId***)**

Prepare *ThreadId* for debugging using the graphical tracer. This implies installing the tracer hooks in the thread and switching the thread to debug mode using `debug/0`. The call is injected into the thread using `thread_signal/2`. We refer to the documentation of this predicate for asynchronous interaction with threads. New threads created inherit their debug mode from the thread that created them.

tdebug

Call `tdebug/1` in all running threads.

tnodebug(+*ThreadId***)**

Disable debugging thread *ThreadId*.

tnodebug

Disable debugging in all threads.

tspy(:*Spec*, +*ThreadId***)**

Set a spy-point as `spy/1` and enable the thread for debugging using `tdebug/1`. Note that a spy-point is a global flag on a predicate that is visible from all threads. Spy points are honoured in all threads that are in debug mode and ignored in threads that are in nodebug mode.

tspy(:*Spec***)**

Set a spy-point as `spy/1` and enable debugging in all threads using `tdebug/0`. Note that removing spy-points can be done using `nospy/1`. Disabling spy-points in a specific thread is achieved by `tnodebug/1`.

8.5.2 Profiling threads

In the current implementation, at most one thread can be profiled at any moment. Any thread can call `profile/1` to profile the execution of some part of its code. The predicate `tprofile/1` allows for profiling the execution of another thread until the user stops collecting profile data.

tprofile(+*ThreadId***)**

Start collecting profile data in *ThreadId* and ask the user to hit ⟨*return*⟩ to stop the profiler. See section 4.39 for details on the execution profiler.

8.6 Unbounded thread creation

(SWI-)Prolog threads are rather heavyweight objects, notably on 32-bit systems, because every thread uses a considerable amount of *virtual* address space. SWI-Prolog threads claim the stack *limit* in virtual address space for each of the runtime stacks, while on 32-bit systems this resource is generally limited somewhere between 1 GB and 3.5 GB, depending on the operating system and operating configuration.

If SWI-Prolog starts a thread it copies the initial goal and starts a POSIX thread which allocates a new Prolog engine that starts proving the given goal. If allocation of the engine fails, typically due to lack of virtual memory space, the thread is still created with minimal (8 Kbyte) stacks and immediately calls its exit handlers. See the option `at_exit`(*Goal*). Although this mechanism allows for handling this type of error gracefully it is not safe to rely on it. Allocating an engine that nearly exhausts virtual address space may cause failures in normal memory allocation that can appear anywhere in Prolog or the foreign libraries used by it. Such errors typically kill the process with a fatal error.

Especially on 32-bit hardware, the design of the application must consider this issue and avoid ungraceful termination, being conservative with the dynamic creation of new threads.

8.7 Multi-threaded mixed C and Prolog applications

All foreign code linked to the multithreading version of SWI-Prolog should be thread-safe (*reentrant*) or guarded in Prolog using `with_mutex/2` from simultaneous access from multiple Prolog threads. If you want to write mixed multithreaded C and Prolog applications you should first familiarise yourself with writing multithreaded applications in C (C++).

If you are using SWI-Prolog as an embedded engine in a multithreaded application you can access the Prolog engine from multiple threads by creating an *engine* in each thread from which you call Prolog. Without creating an engine, a thread can only use functions that do *not* use the `term_t` type (for example `PL_new_atom()`).

The system supports two models. Section 8.7.1 describes the original one-to-one mapping. In this schema a native thread attaches a Prolog thread if it needs to call Prolog and detaches it when finished, as opposed to the model from section 8.7.2, where threads temporarily use a Prolog engine.

8.7.1 A Prolog thread for each native thread (one-to-one)

In the one-to-one model, the thread that called `PL_initialise()` has a Prolog engine attached. If another C-thread in the system wishes to call Prolog it must first attach an engine using `PL_thread_attach_engine()` and call `PL_thread_destroy_engine()` after all Prolog work is finished. This model is especially suitable with long running threads that need to do Prolog work regularly. See section 8.7.2 for the alternative many-to-many model.

`int` **PL_thread_self**()

Returns the integer Prolog identifier of the engine or -1 if the calling thread has no Prolog engine. This function is also provided in the single-threaded version of SWI-Prolog, where it returns -2.

`int` **PL_unify_thread_id**(*term_t t, int i*)

Unify *t* with the Prolog thread identifier for thread *i*. Thread identifiers are normally returned from `PL_thread_self()`. Returns -1 if the thread does not exist or the unification fails.

`int` **PL_thread_attach_engine**(*const PL_thread_attr_t *attr*)

Creates a new Prolog engine in the calling thread. If the calling thread already has an engine the reference count of the engine is incremented. The *attr* argument can be `NULL` to create a thread with default attributes. Otherwise it is a pointer to a structure with the definition below. For any field with value '0', the default is used. The `cancel` field may be filled with a pointer to a function that is called when `PL_cleanup()` terminates the running Prolog engines. If this function is not present or returns `FALSE` pthread_cancel() is used.

```
typedef struct
{ unsigned long     local_size;     /* Stack sizes (K-bytes) */
  unsigned long     global_size;
  unsigned long     trail_size;
  unsigned long     argument_size;
  char *            alias;          /* alias name */
  int              (*cancel)(int thread);
} PL_thread_attr_t;
```

The structure may be destroyed after `PL_thread_attach_engine()` has returned. On success it returns the Prolog identifier for the thread (as returned by `PL_thread_self()`). If an error occurs, -1 is returned. If this Prolog is not compiled for multithreading, -2 is returned.

`int` **PL_thread_destroy_engine**()

Destroy the Prolog engine in the calling thread. Only takes effect if `PL_thread_destroy_engine()` is called as many times as `PL_thread_attach_engine()` in this thread. Returns `TRUE` on success and `FALSE` if the calling thread has no engine or this Prolog does not support threads.

Please note that construction and destruction of engines are relatively expensive operations. Only destroy an engine if performance is not critical and memory is a critical resource.

`int` **PL_thread_at_exit**(*void (*function)(void *), void *closure, int global*)

Register a handle to be called as the Prolog engine is destroyed. The handler function is called with one `void *` argument holding *closure*. If *global* is `TRUE`, the handler is installed *for all threads*. Globally installed handlers are executed after the thread-local handlers. If the handler is installed local for the current thread only (*global* == `FALSE`) it is stored in the same FIFO queue as used by `thread_at_exit/1`.

8.7.2 Pooling Prolog engines (many-to-many)

In this model Prolog engines live as entities that are independent from threads. If a thread needs to call Prolog it takes one of the engines from the pool and returns the engine when done. This model is suitable in the following identified cases:

- *Compatibility with the single-threaded version*
  In the single-threaded version, foreign threads must serialise access to the one and only thread engine. Functions from this section allow sharing one engine among multiple threads.

- *Many native threads with infrequent Prolog work*
  Prolog threads are expensive in terms of memory and time to create and destroy them. For

systems that use a large number of threads that only infrequently need to call Prolog, it is better to take an engine from a pool and return it there.

- *Prolog status must be handed to another thread*
  This situation has been identified by Uwe Lesta when creating a .NET interface for SWI-Prolog. .NET distributes work for an active internet connection over a pool of threads. If a Prolog engine contains the state for a connection, it must be possible to detach the engine from a thread and re-attach it to another thread handling the same connection.

`PL_engine_t` **PL_create_engine**(*PL_thread_attr_t *attributes*)

Create a new Prolog engine. *attributes* is described with `PL_thread_attach_engine()`. Any thread can make this call after `PL_initialise()` returns success. The returned engine is not attached to any thread and lives until `PL_destroy_engine()` is used on the returned handle.

In the single-threaded version this call always returns `NULL`, indicating failure.

`int` **PL_destroy_engine**(*PL_engine_t e*)

Destroy the given engine. Destroying an engine is only allowed if the engine is not attached to any thread or attached to the calling thread. On success this function returns `TRUE`, on failure the return value is `FALSE`.

`int` **PL_set_engine**(*PL_engine_t engine, PL_engine_t *old*)

Make the calling thread ready to use *engine*. If *old* is non-`NULL` the current engine associated with the calling thread is stored at the given location. If *engine* equals `PL_ENGINE_MAIN` the initial engine is attached to the calling thread. If *engine* is `PL_ENGINE_CURRENT` the engine is not changed. This can be used to query the current engine. This call returns `PL_ENGINE_SET` if the engine was switched successfully, `PL_ENGINE_INVAL` if *engine* is not a valid engine handle and `PL_ENGINE_INUSE` if the engine is currently in use by another thread.

Engines can be changed at any time. For example, it is allowed to select an engine to initiate a Prolog goal, detach it and at a later moment execute the goal from another thread. Note however that the `term_t`, `qid_t` and `fid_t` types are interpreted relative to the engine for which they are created. Behaviour when passing one of these types from one engine to another is undefined.

In the single-threaded version this call only succeeds if *engine* refers to the main engine.

Engines in single-threaded SWI-Prolog

In theory it is possible to port the API of section 8.7.2 to the single-threaded version of SWI-Prolog. This allows C-programs to control multiple Prolog engines concurrently. This has not yet been realised.

8.8 Multithreading and the XPCE graphics system

GUI applications written in XPCE can benefit from Prolog threads if they need to do expensive computations that would otherwise block the UI. The XPCE message passing system is guarded with a single *mutex*, which synchronises both access from Prolog and activation through the GUI. In MS-Windows, GUI events are processed by the thread that created the window in which the event occurred, whereas in Unix/X11 they are processed by the thread that dispatches messages. In practice, the most feasible approach to graphical Prolog implementations is to control XPCE from a single thread and deploy other threads for (long) computations.

Traditionally, XPCE runs in the foreground (`main`) thread. We are working towards a situation where XPCE can run comfortably in a separate thread. A separate XPCE thread can be created using `pce_dispatch/1`. It is also possible to create this thread as the (pce) is loaded by setting the `xpce_threaded` to `true`.

Threads other than the thread in which XPCE runs are provided with two predicates to communicate with XPCE.

in_pce_thread(:*Goal***)** *[det]*

Assuming XPCE is running in the foreground thread, this call gives background threads the opportunity to make calls to the XPCE thread. A call to `in_pce_thread/1` succeeds immediately, copying *Goal* to the XPCE thread. *Goal* is added to the XPCE event queue and executed synchronous to normal user events like typing and clicking.

in_pce_thread_sync(:*Goal***)** *[semidet]*

Same as `in_pce_thread/1`, but wait for *Goal* to be completed. Success depends on the success of executing *Goal*. Variable bindings inside *Goal* are visible to the caller, but it should be noted that the values are being *copied*. If *Goal* throws an exception, this exception is re-thrown by `in_pce_thread/1`. If the calling thread is the 'pce thread', `in_pce_thread_sync/1` executes a direct meta-call. See also `pce_thread/1`.

Note that `in_pce_thread_sync/1` is expensive because it requires copying and thread communication. For example, `in_pce_thread_synctrue` runs at approximately 50,000 calls per second (AMD Phenom 9600B, Ubuntu 11.04).

pce_dispatch(+*Options***)**

Create a Prolog thread with the alias name `pce` for XPCE event-handling. In the X11 version this call creates a thread that executes the X11 event-dispatch loop. In MS-Windows it creates a thread that executes a windows event-dispatch loop. The XPCE event-handling thread has the alias `pce`. *Options* specifies the thread attributes as `thread_create/3`.

Foreign Language Interface

9

SWI-Prolog offers a powerful interface to C [Kernighan & Ritchie, 1978]. The main design objectives of the foreign language interface are flexibility and performance. A foreign predicate is a C function that has the same number of arguments as the predicate represented. C functions are provided to analyse the passed terms, convert them to basic C types as well as to instantiate arguments using unification. Non-deterministic foreign predicates are supported, providing the foreign function with a handle to control backtracking.

C can call Prolog predicates, providing both a query interface and an interface to extract multiple solutions from a non-deterministic Prolog predicate. There is no limit to the nesting of Prolog calling C, calling Prolog, etc. It is also possible to write the 'main' in C and use Prolog as an embedded logical engine.

9.1 Overview of the Interface

A special include file called `SWI-Prolog.h` should be included with each C-source file that is to be loaded via the foreign interface. The installation process installs this file in the directory `include` in the SWI-Prolog home directory (`?- current_prolog_flag(home, Home).`). This C-header file defines various data types, macros and functions that can be used to communicate with SWI-Prolog. Functions and macros can be divided into the following categories:

- Analysing Prolog terms
- Constructing new terms
- Unifying terms
- Returning control information to Prolog
- Registering foreign predicates with Prolog
- Calling Prolog from C
- Recorded database interactions
- Global actions on Prolog (halt, break, abort, etc.)

9.2 Linking Foreign Modules

Foreign modules may be linked to Prolog in two ways. Using *static linking*, the extensions, a (short) file defining main() which attaches the extensions calls to Prolog, and the SWI-Prolog kernel distributed as a C library are linked together to form a new executable. Using *dynamic linking*, the extensions are linked to a shared library (`.so` file on most Unix systems) or dynamic link library (`.DLL` file on Microsoft platforms) and loaded into the running Prolog process.[1]

---

[1] The system also contains code to load `.o` files directly for some operating systems, notably Unix systems using the BSD `a.out` executable format. As the number of Unix platforms supporting this quickly gets smaller and this interface is

9.2.1 What linking is provided?

The *static linking* schema can be used on all versions of SWI-Prolog. Whether or not dynamic linking is supported can be deduced from the Prolog flag `open_shared_object` (see `current_prolog_flag/2`). If this Prolog flag yields `true`, `open_shared_object/2` and related predicates are defined. See section 9.2.3 for a suitable high-level interface to these predicates.

9.2.2 What kind of loading should I be using?

All described approaches have their advantages and disadvantages. Static linking is portable and allows for debugging on all platforms. It is relatively cumbersome and the libraries you need to pass to the linker may vary from system to system, though the utility program `swipl-ld` described in section 9.5 often hides these problems from the user.

Loading shared objects (DLL files on Windows) provides sharing and protection and is generally the best choice. If a saved state is created using `qsave_program/[1,2]`, an `initialization/1` directive may be used to load the appropriate library at start-up.

Note that the definition of the foreign predicates is the same, regardless of the linking type used.

9.2.3 library(shlib): Utility library for loading foreign objects (DLLs, shared objects)

This section discusses the functionality of the (autoload) library(shlib), providing an interface to manage shared libraries. We describe the procedure for using a foreign resource (DLL in Windows and shared object in Unix) called `mylib`.

First, one must assemble the resource and make it compatible to SWI-Prolog. The details for this vary between platforms. The plld(1) utility can be used to deal with this in a portable manner. The typical commandline is:

```
plld -o mylib file.{c,o,cc,C} ...
```

Make sure that one of the files provides a global function `install_mylib()` that initialises the module using calls to PL_register_foreign(). Here is a simple example file mylib.c, which creates a Windows MessageBox:

```
#include <windows.h>
#include <SWI-Prolog.h>

static foreign_t
pl_say_hello(term_t to)
{ char *a;

  if ( PL_get_atom_chars(to, &a) )
  { MessageBox(NULL, a, "DLL test", MB_OK|MB_TASKMODAL);

    PL_succeed;
  }
```

---

difficult to port and slow, it is no longer described in this manual. The best alternative would be to use the dld package on machines that do not have shared libraries.

```
  PL_fail;
}

install_t
install_mylib()
{ PL_register_foreign("say_hello", 1, pl_say_hello, 0);
}
```

Now write a file `mylib.pl`:

```
:- module(mylib, [ say_hello/1 ]).
:- use_foreign_library(foreign(mylib)).
```

The file `mylib.pl` can be loaded as a normal Prolog file and provides the predicate defined in C.

load_foreign_library(:*FileSpec***)** *[det]*
load_foreign_library(:*FileSpec, +Entry:atom***)** *[det]*

Load a *shared object* or *DLL*. After loading the *Entry* function is called without arguments. The default entry function is composed from =install_=, followed by the file base-name. E.g., the load-call below calls the function `install_mylib()`. If the platform prefixes extern functions with =_=, this prefix is added before calling.

```
        ...
        load_foreign_library(foreign(mylib)),
        ...
```

Parameters

FileSpec	is a specification for `absolute_file_name/3`. If searching the file fails, the plain name is passed to the OS to try the default method of the OS for locating foreign objects. The default definition of `file_search_path/2` searches <prolog home>/lib/<arch> on Unix and <prolog home>/bin on Windows.

See also `use_foreign_library/1`,2 are intended for use in directives.

use_foreign_library(+*FileSpec***)** *[det]*
use_foreign_library(+*FileSpec, +Entry:atom***)** *[det]*

Load and install a foreign library as `load_foreign_library/1`,2 and register the installation using `initialization/2` with the option `now`. This is similar to using:

```
:- initialization(load_foreign_library(foreign(mylib))).
```

but using the `initialization/1` wrapper causes the library to be loaded *after* loading of the file in which it appears is completed, while `use_foreign_library/1` loads the library *immediately*. I.e. the difference is only relevant if the remainder of the file uses functionality of the C-library.

unload_foreign_library(+*FileSpec***)** *[det]*
unload_foreign_library(+*FileSpec*, +*Exit:atom***)** *[det]*

Unload a *shared object* or *DLL*. After calling the *Exit* function, the shared object is removed from the process. The default exit function is composed from =uninstall_=, followed by the file base-name.

current_foreign_library(*?File, ?Public***)**

Query currently loaded shared libraries.

reload_foreign_libraries

Reload all foreign libraries loaded (after restore of a state created using `qsave_program/2`.

9.2.4 Low-level operations on shared libraries

The interface defined in this section allows the user to load shared libraries (`.so` files on most Unix systems, `.dll` files on Windows). This interface is portable to Windows as well as to Unix machines providing `dlopen(2)` (Solaris, Linux, FreeBSD, Irix and many more) or `shl_open(2)` (HP/UX). It is advised to use the predicates from section 9.2.3 in your application.

open_shared_object(+*File*, -*Handle***)**

File is the name of a shared object file (DLL in MS-Windows). This file is attached to the current process and *Handle* is unified with a handle to the library. Equivalent to `open_shared_object(File, Handle, [])`. See also `open_shared_object/3` and `load_foreign_library/1`.

On errors, an exception `shared_object`(*Action, Message*) is raised. *Message* is the return value from dlerror().

open_shared_object(+*File*, -*Handle*, +*Options***)**

As `open_shared_object/2`, but allows for additional flags to be passed. *Options* is a list of atoms. `now` implies the symbols are resolved immediately rather than lazy (default). `global` implies symbols of the loaded object are visible while loading other shared objects (by default they are local). Note that these flags may not be supported by your operating system. Check the documentation of dlopen() or equivalent on your operating system. Unsupported flags are silently ignored.

close_shared_object(+*Handle***)**

Detach the shared object identified by *Handle*.

call_shared_object_function(+*Handle*, +*Function***)**

Call the named function in the loaded shared library. The function is called without arguments and the return value is ignored. Normally this function installs foreign language predicates using calls to `PL_register_foreign()`.

9.2.5 Static Linking

Below is an outline of the files structure required for statically linking SWI-Prolog with foreign extensions. `.../swipl` refers to the SWI-Prolog home directory (see the Prolog flag `home`). ⟨*arch*⟩ refers to the architecture identifier that may be obtained using the Prolog flag `arch`.

`.../swipl/runtime/`⟨*arch*⟩`/libswipl.a`	SWI-Library
`.../swipl/include/SWI-Prolog.h`	Include file
`.../swipl/include/SWI-Stream.h`	Stream I/O include file
`.../swipl/include/SWI-Exports`	Export declarations (AIX only)
`.../swipl/include/stub.c`	Extension stub

The definition of the foreign predicates is the same as for dynamic linking. Unlike with dynamic linking however, there is no initialisation function. Instead, the file `.../swipl/include/stub.c` may be copied to your project and modified to define the foreign extensions. Below is `stub.c`, modified to link the lowercase example described later in this chapter:

```
#include <stdio.h>
#include <SWI-Prolog.h>

extern foreign_t pl_lowercase(term, term);

PL_extension predicates[] =
{
/*{ "name",      arity,  function,      PL_FA_<flags> },*/

  { "lowercase", 2       pl_lowercase,  0 },
  { NULL,        0,      NULL,          0 }     /* terminating line */
};


int
main(int argc, char **argv)
{ PL_register_extensions(predicates);

  if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  PL_install_readline();                /* delete if not required */

  PL_halt(PL_toplevel() ? 0 : 1);
}
```

Now, a new executable may be created by compiling this file and linking it to `libpl.a` from the runtime directory and the libraries required by both the extensions and the SWI-Prolog kernel. This may be done by hand, or by using the `swipl-ld` utility described in section 9.5. If the linking is performed by hand, the command line option `-dump-runtime-variables` (see section 2.4) can be used to obtain the required paths, libraries and linking options to link the new executable.

9.3 Interface Data types

9.3.1 Type `term_t`: a reference to a Prolog term

The principal data type is `term_t`. Type `term_t` is what Quintus calls `QP_term_ref`. This name indicates better what the type represents: it is a *handle* for a term rather than the term itself. Terms can only be represented and manipulated using this type, as this is the only safe way to ensure the Prolog kernel is aware of all terms referenced by foreign code and thus allows the kernel to perform garbage collection and/or stack-shifts while foreign code is active, for example during a callback from C.

A term reference is a C unsigned long, representing the offset of a variable on the Prolog environment stack. A foreign function is passed term references for the predicate arguments, one for each argument. If references for intermediate results are needed, such references may be created using `PL_new_term_ref()` or `PL_new_term_refs()`. These references normally live till the foreign function returns control back to Prolog. Their scope can be explicitly limited using `PL_open_foreign_frame()` and `PL_close_foreign_frame()`/`PL_discard_foreign_frame()`.

A `term_t` always refers to a valid Prolog term (variable, atom, integer, float or compound term). A term lives either until backtracking takes us back to a point before the term was created, the garbage collector has collected the term, or the term was created after a `PL_open_foreign_frame()` and `PL_discard_foreign_frame()` has been called.

The foreign interface functions can either *read*, *unify* or *write* to term references. In this document we use the following notation for arguments of type `term_t`:

`term_t +t`	Accessed in read-mode. The '+' indicates the argument is 'input'.
`term_t -t`	Accessed in write-mode.
`term_t ?t`	Accessed in unify-mode.

Term references are obtained in any of the following ways:

- *Passed as argument*
  The C functions implementing foreign predicates are passed their arguments as term references. These references may be read or unified. Writing to these variables causes undefined behaviour.

- *Created by* `PL_new_term_ref()`
  A term created by `PL_new_term_ref()` is normally used to build temporary terms or to be written by one of the interface functions. For example, `PL_get_arg()` writes a reference to the term argument in its last argument.

- *Created by* `PL_new_term_refs(`*int n*`)`
  This function returns a set of term references with the same characteristics as `PL_new_term_ref()`. See `PL_open_query()`.

- *Created by* `PL_copy_term_ref(`*term_t t*`)`
  Creates a new term reference to the same term as the argument. The term may be written to. See figure 9.2.

Term references can safely be copied to other C variables of type `term_t`, but all copies will always refer to the same term.

`term_t` **PL_new_term_ref**()

Return a fresh reference to a term. The reference is allocated on the *local* stack. Allocating a term reference may trigger a stack-shift on machines that cannot use sparse memory management for allocation of the Prolog stacks. The returned reference describes a variable.

`term_t` **PL_new_term_refs**(*int n*)

Return *n* new term references. The first term reference is returned. The others are $t + 1$, $t + 2$, etc. There are two reasons for using this function. `PL_open_query()` expects the arguments as a set of consecutive term references, and *very* time-critical code requiring a number of term references can be written as:

```
pl_mypredicate(term_t a0, term_t a1)
{ term_t t0 = PL_new_term_refs(2);
  term_t t1 = t0+1;

  ...
}
```

`term_t` **PL_copy_term_ref**(*term_t from*)

Create a new term reference and make it point initially to the same term as *from*. This function is commonly used to copy a predicate argument to a term reference that may be written.

`void` **PL_reset_term_refs**(*term_t after*)

Destroy all term references that have been created after *after*, including *after* itself. Any reference to the invalidated term references after this call results in undefined behaviour.

Note that returning from the foreign context to Prolog will reclaim all references used in the foreign context. This call is only necessary if references are created inside a loop that never exits back to Prolog. See also `PL_open_foreign_frame()`, `PL_close_foreign_frame()` and `PL_discard_foreign_frame()`.

Interaction with the garbage collector and stack-shifter

Prolog implements two mechanisms for avoiding stack overflow: garbage collection and stack expansion. On machines that allow for it, Prolog will use virtual memory management to detect stack overflow and expand the runtime stacks. On other machines Prolog will reallocate the stacks and update all pointers to them. To do so, Prolog needs to know which data is referenced by C code. As all Prolog data known by C is referenced through term references (`term_t`), Prolog has all the information necessary to perform its memory management without special precautions from the C programmer.

9.3.2 Other foreign interface types

atom_t An atom in Prolog's internal representation. Atoms are pointers to an opaque structure. They are a unique representation for represented text, which implies that atom *A* represents the same text as atom *B* if and only if *A* and *B* are the same pointer.

Atoms are the central representation for textual constants in Prolog. The transformation of a character string *C* to an atom implies a hash-table lookup. If the same atom is needed often, it is advised to store its reference in a global variable to avoid repeated lookup.

functor_t A functor is the internal representation of a name/arity pair. They are used to find the name and arity of a compound term as well as to construct new compound terms. Like atoms they live for the whole Prolog session and are unique.

predicate_t Handle to a Prolog predicate. Predicate handles live forever (although they can lose their definition).

qid_t Query identifier. Used by `PL_open_query()`, `PL_next_solution()` and `PL_close_query()` to handle backtracking from C.

fid_t Frame identifier. Used by `PL_open_foreign_frame()` and `PL_close_foreign_frame()`.

module_t A module is a unique handle to a Prolog module. Modules are used only to call predicates in a specific module.

foreign_t Return type for a C function implementing a Prolog predicate.

control_t Passed as additional argument to non-deterministic foreign functions. See PL_retry*() and PL_foreign_context*().

install_t Type for the install() and uninstall() functions of shared or dynamic link libraries. See section 9.2.3.

int64_t Actually part of the C99 standard rather than Prolog. As of version 5.5.6, Prolog integers are 64-bit on all hardware. The C99 type `int64_t` is defined in the `stdint.h` standard header and provides platform-independent 64-bit integers. Portable code accessing Prolog should use this type to exchange integer values. Please note that `PL_get_long()` can return `FALSE` on Prolog integers that cannot be represented as a C long. Robust code should not assume any of the integer fetching functions to succeed, *even* if the Prolog term is known to be an integer.

9.4 The Foreign Include File

9.4.1 Argument Passing and Control

If Prolog encounters a foreign predicate at run time it will call a function specified in the predicate definition of the foreign predicate. The arguments $1, \ldots, \langle arity \rangle$ pass the Prolog arguments to the goal as Prolog terms. Foreign functions should be declared of type `foreign_t`. Deterministic foreign functions have two alternatives to return control back to Prolog:

(return) foreign_t **PL_succeed()**

Succeed deterministically. PL_succeed is defined as `return TRUE`.

(return) foreign_t **PL_fail()**

Fail and start Prolog backtracking. PL_fail is defined as `return FALSE`.

Non-deterministic Foreign Predicates

By default foreign predicates are deterministic. Using the `PL_FA_NONDETERMINISTIC` attribute (see `PL_register_foreign()`) it is possible to register a predicate as a non-deterministic predicate. Writing non-deterministic foreign predicates is slightly more complicated as the foreign function needs context information for generating the next solution. Note that the same foreign function should be prepared to be simultaneously active in more than one goal. Suppose the natural_number_below_n/2 is a non-deterministic foreign predicate, backtracking over all natural numbers lower than the first argument. Now consider the following predicate:

```
quotient_below_n(Q, N) :-
        natural_number_below_n(N, N1),
        natural_number_below_n(N, N2),
        Q =:= N1 / N2, !.
```

In this predicate the function natural_number_below_n/2 simultaneously generates solutions for both its invocations.

Non-deterministic foreign functions should be prepared to handle three different calls from Prolog:

- *Initial call (*`PL_FIRST_CALL`*)*
  Prolog has just created a frame for the foreign function and asks it to produce the first answer.
- *Redo call (*`PL_REDO`*)*
  The previous invocation of the foreign function associated with the current goal indicated it was possible to backtrack. The foreign function should produce the next solution.
- *Terminate call (*`PL_PRUNED`*)*
  The choice point left by the foreign function has been destroyed by a cut. The foreign function is given the opportunity to clean the environment.

Both the context information and the type of call is provided by an argument of type `control_t` appended to the argument list for deterministic foreign functions. The macro `PL_foreign_control()` extracts the type of call from the control argument. The foreign function can pass a context handle using the `PL_retry*()` macros and extract the handle from the extra argument using the `PL_foreign_context*()` macro.

(return) foreign_t **PL_retry(***intptr_t value***)**

The foreign function succeeds while leaving a choice point. On backtracking over this goal the foreign function will be called again, but the control argument now indicates it is a ‘Redo’ call and the macro `PL_foreign_context()` returns the handle passed via `PL_retry()`. This handle is a signed value two bits smaller than a pointer, i.e., 30 or 62 bits (two bits are used for status indication). Defined as `return _PL_retry(`*n*`)`. See also `PL_succeed()`.

(return) foreign_t **PL_retry_address(***void ****)**

As `PL_retry()`, but ensures an address as returned by malloc() is correctly recovered by `PL_foreign_context_address()`. Defined as `return _PL_retry_address(`*n*`)`. See also `PL_succeed()`.

```
typedef struct                    /* define a context structure */
{ ...
} context;

foreign_t
my_function(term_t a0, term_t a1, control_t handle)
{ struct context * ctxt;

  switch( PL_foreign_control(handle) )
  { case PL_FIRST_CALL:
        ctxt = malloc(sizeof(struct context));
        ...
        PL_retry_address(ctxt);
    case PL_REDO:
        ctxt = PL_foreign_context_address(handle);
        ...
        PL_retry_address(ctxt);
    case PL_PRUNED:
        ctxt = PL_foreign_context_address(handle);
        ...
        free(ctxt);
        PL_succeed;
  }
}
```

Figure 9.1: Skeleton for non-deterministic foreign functions

int **PL_foreign_control(***control_t***)**

Extracts the type of call from the control argument. The return values are described above. Note that the function should be prepared to handle the `PL_PRUNED` case and should be aware that the other arguments are not valid in this case.

intptr_t **PL_foreign_context(***control_t***)**

Extracts the context from the context argument. If the call type is `PL_FIRST_CALL` the context value is 0L. Otherwise it is the value returned by the last `PL_retry()` associated with this goal (both if the call type is `PL_REDO` or `PL_PRUNED`).

void * **PL_foreign_context_address(***control_t***)**

Extracts an address as passed in by `PL_retry_address()`.

Note: If a non-deterministic foreign function returns using `PL_succeed()` or `PL_fail()`, Prolog assumes the foreign function has cleaned its environment. **No** call with control argument `PL_PRUNED` will follow.

The code of figure 9.1 shows a skeleton for a non-deterministic foreign predicate definition.

9.4.2 Atoms and functors

The following functions provide for communication using atoms and functors.

`atom_t` **PL_new_atom**(*const char* *)

Return an atom handle for the given C-string. This function always succeeds. The returned handle is valid as long as the atom is referenced (see section 9.4.2).

`const char*` **PL_atom_chars**(*atom_t atom*)

Return a C-string for the text represented by the given atom. The returned text will not be changed by Prolog. It is not allowed to modify the contents, not even 'temporary' as the string may reside in read-only memory. The returned string becomes invalid if the atom is garbage-collected (see section 9.4.2). Foreign functions that require the text from an atom passed in a `term_t` normally use `PL_get_atom_chars()` or `PL_get_atom_nchars()`.

`functor_t` **PL_new_functor**(*atom_t name, int arity*)

Returns a *functor identifier*, a handle for the name/arity pair. The returned handle is valid for the entire Prolog session.

`atom_t` **PL_functor_name**(*functor_t f*)

Return an atom representing the name of the given functor.

`int` **PL_functor_arity**(*functor_t f*)

Return the arity of the given functor.

Atoms and atom garbage collection

With the introduction of atom garbage collection in version 3.3.0, atoms no longer live as long as the process. Instead, their lifetime is guaranteed only as long as they are referenced. In the single-threaded version, atom garbage collections are only invoked at the *call-port*. In the multi-threaded version (see chapter 8), they appear asynchronously, except for the invoking thread.

For dealing with atom garbage collection, two additional functions are provided:

`void` **PL_register_atom**(*atom_t atom*)

Increment the reference count of the atom by one. `PL_new_atom()` performs this automatically, returning an atom with a reference count of at least one.[2]

`void` **PL_unregister_atom**(*atom_t atom*)

Decrement the reference count of the atom. If the reference count drops below zero, an assertion error is raised.

Please note that the following two calls are different with respect to atom garbage collection:

```
PL_unify_atom_chars(t, "text");
PL_unify_atom(t, PL_new_atom("text"));
```

The latter increments the reference count of the atom `text`, which effectively ensures the atom will never be collected. It is advised to use the *_chars() or *_nchars() functions whenever applicable.

---

[2] Otherwise asynchronous atom garbage collection might destroy the atom before it is used.

9.4.3 Analysing Terms via the Foreign Interface

Each argument of a foreign function (except for the control argument) is of type `term_t`, an opaque handle to a Prolog term. Three groups of functions are available for the analysis of terms. The first just validates the type, like the Prolog predicates `var/1`, `atom/1`, etc., and are called `PL_is_*()`. The second group attempts to translate the argument into a C primitive type. These predicates take a `term_t` and a pointer to the appropriate C type and return `TRUE` or `FALSE` depending on successful or unsuccessful translation. If the translation fails, the pointed-to data is never modified.

Testing the type of a term

`int` **PL_term_type**(*term_t*)

Obtain the type of a term, which should be a term returned by one of the other interface predicates or passed as an argument. The function returns the type of the Prolog term. The type identifiers are listed below. Note that the extraction functions `PL_get_*()` also validate the type and thus the two sections below are equivalent.

```
        if ( PL_is_atom(t) )
        { char *s;

          PL_get_atom_chars(t, &s);
          ...;
        }

or

        char *s;
        if ( PL_get_atom_chars(t, &s) )
        { ...;
        }
```

`PL_VARIABLE`	An unbound variable. The value of term as such is a unique identifier for the variable.
`PL_ATOM`	A Prolog atom.
`PL_STRING`	A Prolog string.
`PL_INTEGER`	A Prolog integer.
`PL_FLOAT`	A Prolog floating point number.
`PL_TERM`	A compound term. Note that a list is a compound term `./2`.

The functions PL_is_⟨*type*⟩ are an alternative to `PL_term_type()`. The test `PL_is_variable(`*term*`)` is equivalent to `PL_term_type(`*term*`) == PL_VARIABLE`, but the first is considerably faster. On the other hand, using a switch over `PL_term_type()` is faster and more readable then using an if-then-else using the functions below. All these functions return either `TRUE` or `FALSE`.

`int` **PL_is_variable**(*term_t*)

Returns non-zero if *term* is a variable.

`int` **PL_is_ground**(*term_t*)
Returns non-zero if *term* is a ground term. See also `ground/1`. This function is cycle-safe.

`int` **PL_is_atom**(*term_t*)
Returns non-zero if *term* is an atom.

`int` **PL_is_string**(*term_t*)
Returns non-zero if *term* is a string.

`int` **PL_is_integer**(*term_t*)
Returns non-zero if *term* is an integer.

`int` **PL_is_float**(*term_t*)
Returns non-zero if *term* is a float.

`int` **PL_is_compound**(*term_t*)
Returns non-zero if *term* is a compound term.

`int` **PL_is_functor**(*term_t, functor_t*)
Returns non-zero if *term* is compound and its functor is *functor*. This test is equivalent to `PL_get_functor()`, followed by testing the functor, but easier to write and faster.

`int` **PL_is_list**(*term_t*)
Returns non-zero if *term* is a compound term with functor `./2` or the atom `[]`. See also `PL_is_pair()` and `PL_skip_list()`.

`int` **PL_is_pair**(*term_t*)
Returns non-zero if *term* is a compound term with functor `./2`. See also `PL_is_list()` and `PL_skip_list()`.

`int` **PL_is_atomic**(*term_t*)
Returns non-zero if *term* is atomic (not variable or compound).

`int` **PL_is_number**(*term_t*)
Returns non-zero if *term* is an integer or float.

`int` **PL_is_acyclic**(*term_t*)
Returns non-zero if *term* is acyclic (i.e. a finite tree).

Reading data from a term

The functions `PL_get_*()` read information from a Prolog term. Most of them take two arguments. The first is the input term and the second is a pointer to the output value or a term reference.

`int` **PL_get_atom**(*term_t +t, atom_t *a*)
If *t* is an atom, store the unique atom identifier over *a*. See also `PL_atom_chars()` and `PL_new_atom()`. If there is no need to access the data (characters) of an atom, it is advised to manipulate atoms using their handle. As the atom is referenced by *t*, it will live at least as long as *t* does. If longer live-time is required, the atom should be locked using `PL_register_atom()`.

`int` **PL_get_atom_chars**(*term_t +t, char **s*)

If *t* is an atom, store a pointer to a 0-terminated C-string in *s*. It is explicitly **not** allowed to modify the contents of this string. Some built-in atoms may have the string allocated in read-only memory, so 'temporary manipulation' can cause an error.

`int` **PL_get_string_chars**(*term_t +t, char **s, int *len*)

If *t* is a string object, store a pointer to a 0-terminated C-string in *s* and the length of the string in *len*. Note that this pointer is invalidated by backtracking, garbage collection and stack-shifts, so generally the only save operations are to pass it immediately to a C function that doesn't involve Prolog.

`int` **PL_get_chars**(*term_t +t, char **s, unsigned flags*)

Convert the argument term *t* to a 0-terminated C-string. *flags* is a bitwise disjunction from two groups of constants. The first specifies which term types should be converted and the second how the argument is stored. Below is a specification of these constants. `BUF_RING` implies, if the data is not static (as from an atom), that the data is copied to the next buffer from a ring of 16 buffers. This is a convenient way of converting multiple arguments passed to a foreign predicate to C-strings. If BUF_MALLOC is used, the data must be freed using `PL_free()` when no longer needed.

With the introduction of wide characters (see section 2.18.1), not all atoms can be converted into a `char*`. This function fails if *t* is of the wrong type, but also if the text cannot be represented. See the `REP_*` flags below for details.

`CVT_ATOM`	Convert if term is an atom
`CVT_STRING`	Convert if term is a string
`CVT_LIST`	Convert if term is a list of integers between 1 and 255
`CVT_INTEGER`	Convert if term is an integer (using `%d`)
`CVT_FLOAT`	Convert if term is a float (using `%f`)
`CVT_NUMBER`	Convert if term is an integer or float
`CVT_ATOMIC`	Convert if term is atomic
`CVT_VARIABLE`	Convert variable to print-name
`CVT_WRITE`	Convert any term that is not converted by any of the other flags using `write/1`. If no `BUF_*` is provided, `BUF_RING` is implied.
`CVT_WRITE_CANINICAL`	As `CVT_WRITE`, but use `write_canonical/2`.
`CVT_ALL`	Convert if term is any of the above, except for `CVT_VARIABLE` and `CVT_WRITE`
`CVT_EXCEPTION`	If conversion fails due to a type error, raise a Prolog type error exception in addition to failure
`BUF_DISCARDABLE`	Data must copied immediately
`BUF_RING`	Data is stored in a ring of buffers
`BUF_MALLOC`	Data is copied to a new buffer returned by `PL_malloc(3)`. When no longer needed the user must call `PL_free()` on the data.
`REP_ISO_LATIN_1`	(0, default). Text is in ISO Latin-1 encoding and the call fails if text cannot be represented.
`REP_UTF8`	Convert the text to a UTF-8 string. This works for all text.
`REP_MB`	Convert to default locale-defined 8-bit string. Success depends on the locale. Conversion is done using the wcrtomb() C library function.

`int` **PL_get_list_chars**(*+term_t l, char **s, unsigned flags*)

Same as `PL_get_chars(`*l, s, CVT_LIST—flags*`)`, provided *flags* contains none of the `CVT_*` flags.

`int` **PL_get_integer**(*+term_t t, int *i*)

If *t* is a Prolog integer, assign its value over *i*. On 32-bit machines, this is the same as `PL_get_long()`, but avoids a warning from the compiler. See also `PL_get_long()`.

`int` **PL_get_long**(*term_t +t, long *i*)

If *t* is a Prolog integer that can be represented as a long, assign its value over *i*. If *t* is an integer that cannot be represented by a C long, this function returns `FALSE`. If *t* is a floating point number that can be represented as a long, this function succeeds as well. See also `PL_get_int64()`.

`int` **PL_get_int64**(*term_t +t, int64_t *i*)

If *t* is a Prolog integer or float that can be represented as a `int64_t`, assign its value over *i*. Currently all Prolog integers can be represented using this type, but this might change if SWI-Prolog introduces unbounded integers.

`int` **PL_get_intptr**(*term_t +t, intptr_t *i*)

Get an integer that is at least as wide as a pointer. On most platforms this is the same as `PL_get_long()`, but on Win64 pointers are 8 bytes and longs only 4. Unlike `PL_get_pointer()`, the value is not modified.

`int` **PL_get_bool**(*term_t +t, int *val*)

If *t* has the value `true` or `false`, set *val* to the C constant `TRUE` or `FALSE` and return success, otherwise return failure.

`int` **PL_get_pointer**(*term_t +t, void **ptr*)

In the current system, pointers are represented by Prolog integers, but need some manipulation to make sure they do not get truncated due to the limited Prolog integer range. `PL_put_pointer()` and `PL_get_pointer()` guarantee pointers in the range of malloc() are handled without truncating.

`int` **PL_get_float**(*term_t +t, double *f*)

If *t* is a float or integer, its value is assigned over *f*.

`int` **PL_get_functor**(*term_t +t, functor_t *f*)

If *t* is compound or an atom, the Prolog representation of the name-arity pair will be assigned over *f*. See also `PL_get_name_arity()` and `PL_is_functor()`.

`int` **PL_get_name_arity**(*term_t +t, atom_t *name, int *arity*)

If *t* is compound or an atom, the functor name will be assigned over *name* and the arity over *arity*. See also `PL_get_functor()` and `PL_is_functor()`.

`int` **PL_get_module**(*term_t +t, module_t *module*)

If *t* is an atom, the system will look up or create the corresponding module and assign an opaque pointer to it over *module*.

`int` **PL_get_arg**(*int index, term_t +t, term_t -a*)

If *t* is compound and index is between 1 and arity (inclusive), assign *a* with a term reference to the argument.

`int` **_PL_get_arg**(*int index, term_t +t, term_t -a*)

Same as `PL_get_arg()`, but no checking is performed, neither whether *t* is actually a term nor whether *index* is a valid argument index.

Exchanging text using length and string

All internal text representation in SWI-Prolog is represented using `char *` plus length and allow for *0-bytes* in them. The foreign library supports this by implementing a *_nchars() function for each applicable *_chars() function. Below we briefly present the signatures of these functions. For full documentation consult the *_chars() function.

`int` **PL_get_atom_nchars**(*term_t t, size_t *len, char **s*)

See `PL_get_atom_chars()`.

`int` **PL_get_list_nchars**(*term_t t, size_t *len, char **s*)

See `PL_get_list_chars()`.

`int` **PL_get_nchars**(*term_t t, size_t *len, char **s, unsigned int flags*)

See `PL_get_chars()`.

`int` **PL_put_atom_nchars**(*term_t t, size_t len, const char *s*)
See `PL_put_atom_chars()`.

`int` **PL_put_string_nchars**(*term_t t, size_t len, const char *s*)
See `PL_put_string_chars()`.

`int` **PL_put_list_ncodes**(*term_t t, size_t len, const char *s*)
See `PL_put_list_codes()`.

`int` **PL_put_list_nchars**(*term_t t, size_t len, const char *s*)
See `PL_put_list_chars()`.

`int` **PL_unify_atom_nchars**(*term_t t, size_t len, const char *s*)
See `PL_unify_atom_chars()`.

`int` **PL_unify_string_nchars**(*term_t t, size_t len, const char *s*)
See `PL_unify_string_chars()`.

`int` **PL_unify_list_ncodes**(*term_t t, size_t len, const char *s*)
See `PL_unify_codes()`.

`int` **PL_unify_list_nchars**(*term_t t, size_t len, const char *s*)
See `PL_unify_list_chars()`.

In addition, the following functions are available for creating and inspecting atoms:

`atom_t` **PL_new_atom_nchars**(*size_t len, const char *s*)
Create a new atom as `PL_new_atom()`, but using the given length and characters.

`const char *` **PL_atom_nchars**(*atom_t a, size_t *len*)
Extract the text and length of an atom.

Wide-character versions

Support for exchange of wide-character strings is still under consideration. The functions dealing with 8-bit character strings return failure when operating on a wide-character atom or Prolog string object. The functions below can extract and unify both 8-bit and wide atoms and string objects. Wide character strings are represented as C arrays of objects of the type `pl_wchar_t`, which is guaranteed to be the same as `wchar_t` on platforms supporting this type. For example, on MS-Windows, this represents 16-bit UCS2 characters, while using the GNU C library (glibc) this represents 32-bit UCS4 characters.

`atom_t` **PL_new_atom_wchars**(*size_t len, const pl_wchar_t *s*)
Create atom from wide-character string as `PL_new_atom_nchars()` does for ISO-Latin-1 strings. If *s* only contains ISO-Latin-1 characters a normal byte-array atom is created.

`pl_wchar_t*` **PL_atom_wchars**(*atom_t atom, int *len*)
Extract characters from a wide-character atom. Fails (returns `NULL`) if *atom* is not a wide-character atom. This is the wide-character version of `PL_atom_nchars()`. Note that only one of these functions succeeds on a particular atom. Especially, after creating an atom with `PL_new_atom_wchars()`, extracting the text using `PL_atom_wchars()` will fail if the atom only contains ISO-Latin-1 characters.

`int` **PL_get_wchars**(*term_t t, size_t *len, pl_wchar_t **s, unsigned flags*)

Wide-character version of `PL_get_chars()`. The *flags* argument is the same as for `PL_get_chars()`.

`int` **PL_unify_wchars**(*term_t t, int type, size_t len, const pl_wchar_t *s*)

Unify *t* with a textual representation of the C wide-character array *s*. The *type* argument defines the Prolog representation and is one of `PL_ATOM`, `PL_STRING`, `PL_CODE_LIST` or `PL_CHAR_LIST`.

`int` **PL_unify_wchars_diff**(*term_t +t, term_t -tail, int type, size_t len, const pl_wchar_t *s*)

Difference list version of `PL_unify_wchars()`, only supporting the types `PL_CODE_LIST` and `PL_CHAR_LIST`. It serves two purposes. It allows for returning very long lists from data read from a stream without the need for a resizing buffer in C. Also, the use of difference lists is often practical for further processing in Prolog. Examples can be found in `packages/clib/readutil.c` from the source distribution.

Reading a list

The functions from this section are intended to read a Prolog list from C. Suppose we expect a list of atoms; the following code will print the atoms, each on a line:

```
foreign_t
pl_write_atoms(term_t l)
{ term_t head = PL_new_term_ref();      /* variable for the elements */
  term_t list = PL_copy_term_ref(l);    /* copy as we need to write */

  while( PL_get_list(list, head, list) )
  { char *s;

    if ( PL_get_atom_chars(head, &s) )
      Sprintf("%s\n", s);
    else
      PL_fail;
  }

  return PL_get_nil(list);              /* test end for [] */
}
```

`int` **PL_get_list**(*term_t +l, term_t -h, term_t -t*)

If *l* is a list and not `[]` assign a term reference to the head to *h* and to the tail to *t*.

`int` **PL_get_head**(*term_t +l, term_t -h*)

If *l* is a list and not `[]` assign a term reference to the head to *h*.

`int` **PL_get_tail**(*term_t +l, term_t -t*)

If *l* is a list and not `[]` assign a term reference to the tail to *t*.

`int` **PL_get_nil**(*term_t +l*)

Succeeds if *l* represents the atom `[]`.

`int` **PL_skip_list**(*term_t +list, term_t -tail, size_t *len*)

This is a multi-purpose function to deal with lists. It allows for finding the length of a list, checking whether something is a list, etc. The reference *tail* is set to point to the end of the list, *len* is filled with the number of list-cells skipped, and the return value indicates the status of the list:

PL_LIST

The list is a 'proper' list: one that ends in `[]` and *tail* is filled with `[]`

PL_PARTIAL_LIST

The list is a 'partial' list: one that ends in a variable and *tail* is a reference to this variable.

PL_CYCLIC_TERM

The list is cyclic (e.g. X = [a—X]). *tail* points to an arbitrary cell of the list and *len* is at most twice the cycle length of the list.

PL_NOT_A_LIST

The term *list* is not a list at all. *tail* is bound to the non-list term and *len* is set to the number of list-cells skipped.

It is allowed to pass 0 for *tail* and `NULL` for *len*.

An example: defining `write/1` in C

Figure 9.2 shows a simplified definition of `write/1` to illustrate the described functions. This simplified version does not deal with operators. It is called `display/1`, because it mimics closely the behaviour of this Edinburgh predicate.

9.4.4 Constructing Terms

Terms can be constructed using functions from the `PL_put_*()` and `PL_cons_*()` families. This approach builds the term 'inside-out', starting at the leaves and subsequently creating compound terms. Alternatively, terms may be created 'top-down', first creating a compound holding only variables and subsequently unifying the arguments. This section discusses functions for the first approach. This approach is generally used for creating arguments for `PL_call()` and `PL_open_query()`.

`void` **PL_put_variable**(*term_t -t*)

Put a fresh variable in the term, resetting the term reference to its initial state.[3]

`void` **PL_put_atom**(*term_t -t, atom_t a*)

Put an atom in the term reference from a handle. See also `PL_new_atom()` and `PL_atom_chars()`.

`void` **PL_put_bool**(*term_t -t, int val*)

Put one of the atoms `true` or `false` in the term reference See also `PL_put_atom()`, `PL_unify_bool()` and `PL_get_bool()`.

`int` **PL_put_atom_chars**(*term_t -t, const char *chars*)

Put an atom in the term reference constructed from the zero-terminated string. The string itself will never be referenced by Prolog after this function.

---

[3] Older versions created a variable on the global stack.

```
foreign_t
pl_display(term_t t)
{ functor_t functor;
  int arity, len, n;
  char *s;

  switch( PL_term_type(t) )
  { case PL_VARIABLE:
    case PL_ATOM:
    case PL_INTEGER:
    case PL_FLOAT:
      PL_get_chars(t, &s, CVT_ALL);
      Sprintf("%s", s);
      break;
    case PL_STRING:
      PL_get_string_chars(t, &s, &len);
      Sprintf("\"%s\"", s);
      break;
    case PL_TERM:
    { term_t a = PL_new_term_ref();

      PL_get_name_arity(t, &name, &arity);
      Sprintf("%s(", PL_atom_chars(name));
      for(n=1; n<=arity; n++)
      { PL_get_arg(n, t, a);
        if ( n > 1 )
          Sprintf(", ");
        pl_display(a);
      }
      Sprintf(")");
      break;
    default:
      PL_fail;                          /* should not happen */
    }
  }

  PL_succeed;
}
```

Figure 9.2: A Foreign definition of `display/1`

`int` **PL_put_string_chars**(*term_t -t, const char *chars*)

Put a zero-terminated string in the term reference. The data will be copied. See also `PL_put_string_nchars()`.

`int` **PL_put_string_nchars**(*term_t -t, size_t len, const char *chars*)

Put a string, represented by a length/start pointer pair in the term reference. The data will be copied. This interface can deal with 0-bytes in the string. See also section 9.4.20.

`int` **PL_put_list_chars**(*term_t -t, const char *chars*)

Put a list of ASCII values in the term reference.

`int` **PL_put_integer**(*term_t -t, long i*)

Put a Prolog integer in the term reference.

`int` **PL_put_int64**(*term_t -t, int64_t i*)

Put a Prolog integer in the term reference.

`int` **PL_put_pointer**(*term_t -t, void *ptr*)

Put a Prolog integer in the term reference. Provided *ptr* is in the ‘malloc()-area’, `PL_get_pointer()` will get the pointer back.

`int` **PL_put_float**(*term_t -t, double f*)

Put a floating-point value in the term reference.

`int` **PL_put_functor**(*term_t -t, functor_t functor*)

Create a new compound term from *functor* and bind *t* to this term. All arguments of the term will be variables. To create a term with instantiated arguments, either instantiate the arguments using the `PL_unify_*()` functions or use `PL_cons_functor()`.

`int` **PL_put_list**(*term_t -l*)

Same as `PL_put_functor(`*l, PL_new_functor(PL_new_atom(”.”)*`, 2))`.

`void` **PL_put_nil**(*term_t -l*)

Same as `PL_put_atom_chars(`*”[]”*`)`.

`void` **PL_put_term**(*term_t -t1, term_t +t2*)

Make *t1* point to the same term as *t2*.

`int` **PL_cons_functor**(*term_t -h, functor_t f, ...*)

Create a term whose arguments are filled from a variable argument list holding the same number of `term_t` objects as the arity of the functor. To create the term `animal(gnu, 50)`, use:

```
{ term_t a1 = PL_new_term_ref();
  term_t a2 = PL_new_term_ref();
  term_t t  = PL_new_term_ref();
  functor_t animal2;

  /* animal2 is a constant that may be bound to a global
     variable and re-used
  */
  animal2 = PL_new_functor(PL_new_atom("animal"), 2);

  PL_put_atom_chars(a1, "gnu");
  PL_put_integer(a2, 50);
  PL_cons_functor(t, animal2, a1, a2);
}
```

After this sequence, the term references *a1* and *a2* may be used for other purposes.

`int` **PL_cons_functor_v**(*term_t -h, functor_t f, term_t a0*)

Create a compound term like `PL_cons_functor()`, but *a0* is an array of term references as returned by `PL_new_term_refs()`. The length of this array should match the number of arguments required by the functor.

`int` **PL_cons_list**(*term_t -l, term_t +h, term_t +t*)

Create a list (cons-) cell in *l* from the head *h* and tail *t*. The code below creates a list of atoms from a `char **`. The list is built tail-to-head. The `PL_unify_*()` functions can be used to build a list head-to-tail.

```
void
put_list(term_t l, int n, char **words)
{ term_t a = PL_new_term_ref();

  PL_put_nil(l);
  while( --n >= 0 )
  { PL_put_atom_chars(a, words[n]);
    PL_cons_list(l, a, l);
  }
}
```

Note that *l* can be redefined within a `PL_cons_list` call as shown here because operationally its old value is consumed before its new value is set.

9.4.5 Unifying data

The functions of this section *unify* terms with other terms or translated C data structures. Except for `PL_unify()`, these functions are specific to SWI-Prolog. They have been introduced because they shorten the code for returning data to Prolog and at the same time make this more efficient by avoiding the need to allocate temporary term references and reduce the number of calls to the Prolog API. Consider the case where we want a foreign function to return the host name of the machine Prolog is running on. Using the `PL_get_*()` and `PL_put_*()` functions, the code becomes:

```
foreign_t
pl_hostname(term_t name)
{ char buf[100];

  if ( gethostname(buf, sizeof(buf)) )
  { term_t tmp = PL_new_term_ref();
```

```
    PL_put_atom_chars(tmp, buf);
    return PL_unify(name, tmp);
  }

  PL_fail;
}
```

Using `PL_unify_atom_chars()`, this becomes:

```
foreign_t
pl_hostname(term_t name)
{ char buf[100];

  if ( gethostname(buf, sizeof(buf)) )
    return PL_unify_atom_chars(name, buf);

  PL_fail;
}
```

Note that unification functions that perform multiple bindings may leave part of the bindings in case of failure. See `PL_unify()` for details.

`int` **PL_unify**(*term_t ?t1, term_t ?t2*)

Unify two Prolog terms and return `TRUE` on success.

Care is needed if `PL_unify()` returns `FAIL` and the foreign function does not *immediately* return to Prolog with `FAIL`. Unification may perform multiple changes to either *t1* or *t2*. A failing unification may have created bindings before failure is detected. *Already created bindings are not undone*. For example, calling `PL_unify()` on `a`(*X, a*) and `a`(*c,b*) binds *X* to `c` and fails when trying to unify `a` to `b`. If control remains in C or even if we want to return success to Prolog, we *must* undo such bindings. This is achieved using `PL_open_foreign_frame()` and `PL_rewind_foreign_frame()`, as shown in the snippet below.

```
    { fid_t fid = PL_open_foreign_frame();

      ...
      if ( !PL_unify(t1, t2) )
        PL_rewind_foreign_frame(fid);
      ...

      PL_close_foreign_frame(fid);
    }
```

In addition, `PL_unify()` may have failed on an **exception**, typically a resource (stack) overflow. This can be tested using `PL_exception()`, passing 0 (zero) for the query-id argument. Foreign functions that encounter an exception must return `FAIL` to Prolog as soon as possible or call `PL_clear_exception()` if they wish to ignore the exception.

`int` **PL_unify_atom**(*term_t ?t, atom_t a*)

Unify *t* with the atom *a* and return non-zero on success.

`int` **PL_unify_bool**(*term_t ?t, int a*)

Unify *t* with either `true` or `false`.

`int` **PL_unify_chars**(*term_t ?t, int flags, size_t len, const char *chars*)

New function to deal with unification of `char*` with various encodings to a Prolog representation. The *flags* argument is a bitwise *or* specifying the Prolog target type and the encoding of *chars*. A Prolog type is one of `PL_ATOM`, `PL_STRING`, `PL_CODE_LIST` or `PL_CHAR_LIST`. A representation is one of `REP_ISO_LATIN_1`, `REP_UTF8` or `REP_MB`. See `PL_get_chars()` for a definition of the representation types. If *len* is `-1` *chars* must be zero-terminated and the length is computed from *chars* using strlen().

If *flags* includes `PL_DIFF_LIST` and type is one of `PL_CODE_LIST` or `PL_CHAR_LIST`, the text is converted to a *difference list*. The tail of the difference list is $t+1$.

`int` **PL_unify_atom_chars**(*term_t ?t, const char *chars*)

Unify *t* with an atom created from *chars* and return non-zero on success.

`int` **PL_unify_list_chars**(*term_t ?t, const char *chars*)

Unify *t* with a list of ASCII characters constructed from *chars*.

`void` **PL_unify_string_chars**(*term_t ?t, const char *chars*)

Unify *t* with a Prolog string object created from the zero-terminated string *chars*. The data will be copied. See also `PL_unify_string_nchars()`.

`void` **PL_unify_string_nchars**(*term_t ?t, size_t len, const char *chars*)

Unify *t* with a Prolog string object created from the string created from the *len*/*chars* pair. The data will be copied. This interface can deal with 0-bytes in the string. See also section 9.4.20.

`int` **PL_unify_integer**(*term_t ?t, intptr_t n*)

Unify *t* with a Prolog integer from *n*.

`int` **PL_unify_int64**(*term_t ?t, int64_t n*)

Unify *t* with a Prolog integer from *n*.

`int` **PL_unify_float**(*term_t ?t, double f*)

Unify *t* with a Prolog float from *f*.

`int` **PL_unify_pointer**(*term_t ?t, void *ptr*)

Unify *t* with a Prolog integer describing the pointer. See also `PL_put_pointer()` and `PL_get_pointer()`.

`int` **PL_unify_functor**(*term_t ?t, functor_t f*)

If *t* is a compound term with the given functor, just succeed. If it is unbound, create a term and bind the variable, else fail. Note that this function does not create a term if the argument is already instantiated.

`int` **PL_unify_list**(*term_t ?l, term_t -h, term_t -t*)

Unify *l* with a list-cell (`./2`). If successful, write a reference to the head of the list into *h* and a reference to the tail of the list into *t*. This reference may be used for subsequent calls to this function. Suppose we want to return a list of atoms from a `char **`. We could use the example described by `PL_put_list()`, followed by a call to `PL_unify()`, or we can use the code below. If the predicate argument is unbound, the difference is minimal (the code based on `PL_put_list()` is probably slightly faster). If the argument is bound, the code below may fail before reaching the end of the word list, but even if the unification succeeds, this code avoids a duplicate (garbage) list and a deep unification.

```
foreign_t
pl_get_environ(term_t env)
{ term_t l = PL_copy_term_ref(env);
  term_t a = PL_new_term_ref();
  extern char **environ;
  char **e;

  for(e = environ; *e; e++)
  { if ( !PL_unify_list(l, a, l) ||
         !PL_unify_atom_chars(a, *e) )
      PL_fail;
  }

  return PL_unify_nil(l);
}
```

`int` **PL_unify_nil**(*term_t ?l*)

Unify *l* with the atom `[]`.

`int` **PL_unify_arg**(*int index, term_t ?t, term_t ?a*)

Unifies the *index-th* argument (1-based) of *t* with *a*.

`int` **PL_unify_term**(*term_t ?t, . . .*)

Unify *t* with a (normally) compound term. The remaining arguments are a sequence of a type identifier followed by the required arguments. This predicate is an extension to the Quintus and SICStus foreign interface from which the SWI-Prolog foreign interface has been derived, but has proved to be a powerful and comfortable way to create compound terms from C. Due to the vararg packing/unpacking and the required type-switching this interface is slightly slower than using the primitives. Please note that some bad C compilers have fairly low limits on the number of arguments that may be passed to a function.

Special attention is required when passing numbers. C 'promotes' any integral smaller than `int` to `int`. That is, the types `char`, `short` and `int` are all passed as `int`. In addition, on most 32-bit platforms `int` and `long` are the same. Up to version 4.0.5, only `PL_INTEGER` could be specified, which was taken from the stack as `long`. Such code fails when passing small integral types on machines where `int` is smaller than `long`. It is advised to use `PL_SHORT`, `PL_INT` or `PL_LONG` as appropriate. Similarly, C compilers promote `float` to `double` and therefore `PL_FLOAT` and `PL_DOUBLE` are synonyms.

The type identifiers are:

`PL_VARIABLE` *none*
: No op. Used in arguments of `PL_FUNCTOR`.

`PL_BOOL` *int*
: Unify the argument with `true` or `false`.

`PL_ATOM` *atom_t*
: Unify the argument with an atom, as in `PL_unify_atom()`.

`PL_CHARS` *const char **
: Unify the argument with an atom constructed from the C `char *`, as in `PL_unify_atom_chars()`.

`PL_NCHARS` *size_t, const char **
: Unify the argument with an atom constructed from length and `char*` as in `PL_unify_atom_nchars()`.

`PL_UTF8_CHARS` *const char **
: Create an atom from a UTF-8 string.

`PL_UTF8_STRING` *const char **
: Create a packed string object from a UTF-8 string.

`PL_MBCHARS` *const char **
: Create an atom from a multi-byte string in the current locale.

`PL_MBCODES` *const char **
: Create a list of character codes from a multi-byte string in the current locale.

`PL_MBSTRING` *const char **
: Create a packed string object from a multi-byte string in the current locale.

`PL_NWCHARS` *size_t, const wchar_t **
: Create an atom from a length and a wide character pointer.

`PL_NWCODES` *size_t, const wchar_t **
: Create a list of character codes from a length and a wide character pointer.

`PL_NWSTRING` *size_t, const wchar_t **
: Create a packed string object from a length and a wide character pointer.

`PL_SHORT` *short*
: Unify the argument with an integer, as in `PL_unify_integer()`. As `short` is promoted to `int`, `PL_SHORT` is a synonym for `PL_INT`.

`PL_INTEGER` *long*
: Unify the argument with an integer, as in `PL_unify_integer()`.

`PL_INT` *int*
: Unify the argument with an integer, as in `PL_unify_integer()`.

`PL_LONG` *long*
: Unify the argument with an integer, as in `PL_unify_integer()`.

`PL_INT64` *int64_t*
: Unify the argument with a 64-bit integer, as in `PL_unify_int64()`.

`PL_INTPTR` *intptr_t*
: Unify the argument with an integer with the same width as a pointer. On most machines this is the same as `PL_LONG`. but on 64-bit MS-Windows pointers are 64 bits while longs are only 32 bits.

`PL_DOUBLE` *double*
: Unify the argument with a float, as in `PL_unify_float()`. Note that, as the argument is passed using the C vararg conventions, a float must be casted to a double explicitly.

`PL_FLOAT` *double*
: Unify the argument with a float, as in `PL_unify_float()`.

`PL_POINTER` *void **
: Unify the argument with a pointer, as in `PL_unify_pointer()`.

`PL_STRING` *const char **
: Unify the argument with a string object, as in `PL_unify_string_chars()`.

`PL_TERM` *term_t*
: Unify a subterm. Note this may be the return value of a `PL_new_term_ref()` call to get access to a variable.

`PL_FUNCTOR` *functor_t, ...*
: Unify the argument with a compound term. This specification should be followed by exactly as many specifications as the number of arguments of the compound term.

`PL_FUNCTOR_CHARS` *const char *name, int arity, ...*
: Create a functor from the given name and arity and then behave as `PL_FUNCTOR`.

`PL_LIST` *int length, ...*
: Create a list of the indicated length. The remaining arguments contain the elements of the list.

For example, to unify an argument with the term `language(dutch)`, the following skeleton may be used:

```
static functor_t FUNCTOR_language1;

static void
init_constants()
{ FUNCTOR_language1 = PL_new_functor(PL_new_atom("language"), 1);
}

foreign_t
pl_get_lang(term_t r)
{ return PL_unify_term(r,
                       PL_FUNCTOR, FUNCTOR_language1,
                           PL_CHARS, "dutch");
}

install_t
install()
{ PL_register_foreign("get_lang", 1, pl_get_lang, 0);
  init_constants();
}
```

`int` **PL_chars_to_term**(*const char *chars, term_t -t*)

Parse the string *chars* and put the resulting Prolog term into *t*. *chars* may or may not be closed using a Prolog full-stop (i.e., a dot followed by a blank). Returns `FALSE` if a syntax error was encountered and `TRUE` after successful completion. In addition to returning `FALSE`, the exception-term is returned in *t* on a syntax error. See also `term_to_atom/2`.

The following example builds a goal term from a string and calls it.

```
int
call_chars(const char *goal)
{ fid_t fid = PL_open_foreign_frame();
  term_t g = PL_new_term_ref();
  BOOL rval;

  if ( PL_chars_to_term(goal, g) )
    rval = PL_call(goal, NULL);
  else
    rval = FALSE;

  PL_discard_foreign_frame(fid);
  return rval;
}
  ...
  call_chars("consult(load)");
  ...
```

`char *` **PL_quote**(*int chr, const char *string*)

Return a quoted version of *string*. If *chr* is `'\''`, the result is a quoted atom. If *chr* is `'"'`, the result is a string. The result string is stored in the same ring of buffers as described with the `BUF_RING` argument of `PL_get_chars();`

In the current implementation, the string is surrounded by *chr* and any occurrence of *chr* is doubled. In the future the behaviour will depend on the `character_escapes` Prolog flag.

9.4.6 Convenient functions to generate Prolog exceptions

The typical implementation of a foreign predicate first uses the PL_get_*() functions to extract C data types from the Prolog terms. Failure of any of these functions is normally because the Prolog term is of the wrong type. The *_ex() family of functions are wrappers around (mostly) the PL_get_*() functions, such that we can write code in the style below and get proper exceptions if an argument is uninstantiated or of the wrong type.

```
/** set_size(+Name:atom, +Width:int, +Height:int) is det.

static foreign_t
set_size(term_t name, term_t width, term_t height)
{ char *n;
  int w, h;
  if ( !PL_get_chars(name, &n, CVT_ATOM|CVT_EXCEPTION) ||
       !PL_get_integer_ex(with, &w) ||
       !PL_get_integer_ex(height, &h) )
    return FALSE;

  ...

}
```

`int` **PL_get_atom_ex**(*term_t t, atom_t *a*)

As `PL_get_atom()`, but raises a type or instantiation error if *t* is not an atom.

`int` **PL_get_integer_ex**(*term_t t, int *i*)

As `PL_get_integer()`, but raises a type or instantiation error if *t* is not an integer, or a representation error if the Prolog integer does not fit in a C `int`.

`int` **PL_get_long_ex**(*term_t t, long *i*)

As `PL_get_long()`, but raises a type or instantiation error if *t* is not an atom, or a representation error if the Prolog integer does not fit in a C `long`.

`int` **PL_get_int64_ex**(*term_t t, int64_t *i*)

As `PL_get_int64()`, but raises a type or instantiation error if *t* is not an atom, or a representation error if the Prolog integer does not fit in a C `int64_t`.

`int` **PL_get_intptr_ex**(*term_t t, intptr_t *i*)

As `PL_get_intptr()`, but raises a type or instantiation error if *t* is not an atom, or a representation error if the Prolog integer does not fit in a C `intptr_t`.

`int` **PL_get_size_ex**(*term_t t, size_t *i*)

As `PL_get_size()`, but raises a type or instantiation error if *t* is not an atom, or a representation error if the Prolog integer does not fit in a C `size_t`.

`int` **PL_get_bool_ex**(*term_t t, int *i*)

As `PL_get_bool()`, but raises a type or instantiation error if *t* is not an boolean.

`int` **PL_get_float_ex**(*term_t t, double *f*)

As `PL_get_float()`, but raises a type or instantiation error if *t* is not a float.

`int` **PL_get_char_ex**(*term_t t, int *p, int eof*)

Get a character code from *t*, where *t* is either an integer or an atom with length one. If *eof* is `TRUE` and *t* is -1, *p* is filled with -1. Raises an appropriate error if the conversion is not possible.

`int` **PL_get_pointer_ex**(*term_t t, void **addrp*)

As `PL_get_pointer()`, but raises a type or instantiation error if *t* is not a pointer.

`int` **PL_get_list_ex**(*term_t l, term_t h, term_t t*)

As `PL_get_list()`, but raises a type or instantiation error if *t* is not a list.

`int` **PL_get_nil_ex**(*term_t l*)
As `PL_get_nil()`, but raises a type or instantiation error if *t* is not the empty list.

`int` **PL_unify_list_ex**(*term_t l, term_t h, term_t t*)
As `PL_unify_list()`, but raises a type error if *t* is not a variable, list-cell or the empty list.

`int` **PL_unify_nil_ex**(*term_t l*)
As `PL_unify_nil()`, but raises a type error if *t* is not a variable, list-cell or the empty list.

`int` **PL_unify_bool_ex**(*term_t t, int val*)
As `PL_unify_bool()`, but raises a type error if *t* is not a variable or a boolean.

The second family of functions in this section simplifies the generation of ISO compatible error terms. Any foreign function that calls this function must return to Prolog with the return code of the error function or the constant `FALSE`. If available, these error functions add the name of the calling predicate to the error context. See also `PL_raise_exception()`.

`int` **PL_instantiation_error**(*term_t culprit*)
Raise `instantiation_error`. *Culprit* is ignored, but should be bound to the term that is not a variable. See `instantiation_error/1`.

`int` **PL_representation_error**(*const char *resource*)
Raise `representation_error(resource)`. See `representation_error/1`.

`int` **PL_type_error**(*const char *expected, term_t culprit*)
Raise `type_error(expected, culprit)`. See `type_error/2`.

`int` **PL_domain_error**(*const char *expected, term_t culprit*)
Raise `domain_error(expected, culprit)`. See `domain_error/2`.

`int` **PL_existence_error**(*const char *type, term_t culprit*)
Raise `existence_error(type, culprit)`. See `type_error/2`.

`int` **PL_permission_error**(*const char *operation, const char *type, term_t culprit*)
Raise `permission_error(operation, type, culprit)`. See `permission_error/3`.

`int` **PL_resource_error**(*const char *resource*)
Raise `resource_error(resource)`. See `resource_error/1`.

9.4.7 BLOBS: Using atoms to store arbitrary binary data

SWI-Prolog atoms as well as strings can represent arbitrary binary data of arbitrary length. This facility is attractive for storing foreign data such as images in an atom. An atom is a unique handle to this data and the atom garbage collector is able to destroy atoms that are no longer referenced by the Prolog engine. This property of atoms makes them attractive as a handle to foreign resources, such as Java atoms, Microsoft's COM objects, etc., providing safe combined garbage collection.

To exploit these features safely and in an organised manner, the SWI-Prolog foreign interface allows for creating 'atoms' with additional type information. The type is represented by a structure holding C function pointers that tell Prolog how to handle releasing the atom, writing it, sorting it, etc. Two atoms created with different types can represent the same sequence of bytes. Atoms are first ordered on the rank number of the type and then on the result of the `compare()` function. Rank numbers are assigned when the type is registered.

Defining a BLOB type

The type `PL_blob_t` represents a structure with the layout displayed below. The structure contains additional fields at the … for internal bookkeeping as well as future extensions.

```
typedef struct PL_blob_t
{ unsigned long         magic;          /* PL_BLOB_MAGIC */
  unsigned long         flags;          /* Bitwise or of PL_BLOB_* */
  char *                name;           /* name of the type */
  int                   (*release)(atom_t a);
  int                   (*compare)(atom_t a, atom_t b);
  int                   (*write)(IOSTREAM *s, atom_t a, int flags);
  void                  (*acquire)(atom_t a);
  ...
} PL_blob_t;
```

For each type, exactly one such structure should be allocated. Its first field must be initialised to `PL_BLOB_MAGIC`. The *flags* is a bitwise *or* of the following constants:

PL_BLOB_TEXT
If specified the blob is assumed to contain text and is considered a normal Prolog atom.

PL_BLOB_UNIQUE
If specified the system ensures that the blob-handle is a unique reference for a blob with the given type, length and content. If this flag is not specified, each lookup creates a new blob.

PL_BLOB_NOCOPY
By default the content of the blob is copied. Using this flag the blob references the external data directly. The user must ensure the provided pointer is valid as long as the atom lives. If `PL_BLOB_UNIQUE` is also specified, uniqueness is determined by comparing the pointer rather than the data pointed at.

The *name* field represents the type name as available to Prolog. See also `current_blob/2`. The other fields are function pointers that must be initialised to proper functions or `NULL` to get the default behaviour of built-in atoms. Below are the defined member functions:

`void` **acquire**(*atom_t a*)
Called if a new blob of this type is created through `PL_put_blob()` or `PL_unify_blob()`. This callback may be used together with the release hook to deal with reference-counted external objects.

`int` **release**(*atom_t a*)
The blob (atom) *a* is about to be released. This function can retrieve the data of the blob using `PL_blob_data()`. If it returns `FALSE` the atom garbage collector will *not* reclaim the atom.

`int` **compare**(*atom_t a, atom_t b*)

Compare the blobs *a* and *b*, both of which are of the type associated to this blob type. Return values are, as memcmp(), < 0 if *a* is less than *b*, $= 0$ if both are equal, and > 0 otherwise.

`int` **write**(*IOSTREAM *s, atom_t a, int flags*)

Write the content of the blob *a* to the stream *s* respecting the *flags*. The *flags* are a bitwise *or of zero or more of the* `PL_WRT_*` *flags defined in* `SWI-Prolog.h`*. This prototype is available if the undocumented* `SWI-Stream.h` *is included* before `SWI-Prolog.h`.

If this function is not provided, `write/1` *emits the content of the blob for blobs of type* `PL_BLOB_TEXT` *or a string of the format <#hex data> for binary blobs.*

If a blob type is registered from a loadable object (shared object or DLL) the blob type must be deregistered before the object may be released.

`int` **PL_unregister_blob_type**(*PL_blob_t *type*)

Unlink the blob type from the registered type and transform the type of possible living blobs to `unregistered`, avoiding further reference to the type structure, functions referred by it, as well as the data. This function returns `TRUE` if no blobs of this type existed and `FALSE` otherwise. `PL_unregister_blob_type()` is intended for the uninstall() hook of foreign modules, avoiding further references to the module.

Accessing blobs

The blob access functions are similar to the atom accessing functions. Blobs being atoms, the atom functions operate on blobs and vice versa. For clarity and possible future compatibility issues, however, it is not advised to rely on this.

`int` **PL_is_blob**(*term_t t, PL_blob_t **type*)

Succeeds if *t* refers to a blob, in which case *type* is filled with the type of the blob.

`int` **PL_unify_blob**(*term_t t, void *blob, size_t len, PL_blob_t *type*)

Unify *t* to a new blob constructed from the given data and associated to the given type. See also `PL_unify_atom_nchars()`.

`int` **PL_put_blob**(*term_t t, void *blob, size_t len, PL_blob_t *type*)

Store the described blob in *t*. The return value indicates whether a new blob was allocated (`FALSE`) or the blob is a reference to an existing blob (`TRUE`). Reporting new/existing can be used to deal with external objects having their own reference counts. If the return is `TRUE` this reference count must be incremented, and it must be decremented on blob destruction callback. See also `PL_put_atom_nchars()`.

`int` **PL_get_blob**(*term_t t, void **blob, size_t *len, PL_blob_t **type*)

If *t* holds a blob or atom, get the data and type and return `TRUE`. Otherwise return `FALSE`. Each result pointer may be `NULL`, in which case the requested information is ignored.

`void *` **PL_blob_data**(*atom_t a, size_t *len, PL_blob_t **type*)

Get the data and type associated to a blob. This function is mainly used from the callback functions described in section 9.4.7.

9.4.8 Exchanging GMP numbers

If SWI-Prolog is linked with the GNU Multiple Precision Arithmetic Library (GMP, used by default), the foreign interface provides functions for exchanging numeric values to GMP types. To access these functions the header `<gmp.h>` must be included *before* `<SWI-Prolog.h>`. Foreign code using GMP linked to SWI-Prolog asks for some considerations.

- SWI-Prolog normally rebinds the GMP allocation functions using mp_set_memory_functions(). This means SWI-Prolog must be initialised before the foreign code touches any GMP function. You can call `\cfuncref{PL_action}{PL_GMP_SET_ALLOC_FUNCTIONS, TRUE}` to force Prolog's GMP initialization without doing the rest of the Prolog initialization. If you do not want Prolog rebinding the GMP allocation, call `\cfuncref{PL_action}{PL_GMP_SET_ALLOC_FUNCTIONS, FALSE}` *before* initializing Prolog.
- On Windows, each DLL has its own memory pool. To make exchange of GMP numbers between Prolog and foreign code possible you must either let Prolog rebind the allocation functions (default) or you must recompile SWI-Prolog to link to a DLL version of the GMP library.

Here is an example exploiting the function mpz_nextprime():

```
#include <gmp.h>
#include <SWI-Prolog.h>

static foreign_t
next_prime(term_t n, term_t prime)
{ mpz_t mpz;
  int rc;

  mpz_init(mpz);
  if ( PL_get_mpz(n, mpz) )
  { mpz_nextprime(mpz, mpz);

    rc = PL_unify_mpz(prime, mpz);
  } else
    rc = FALSE;

  mpz_clear(mpz);
  return rc;
}

install_t
install()
{ PL_register_foreign("next_prime", 2, next_prime, 0);
}
```

`int` **PL_get_mpz**(*term_t t, mpz_t mpz*)

If *t* represents an integer, *mpz* is filled with the value and the function returns `TRUE`. Otherwise *mpz* is untouched and the function returns `FALSE`. Note that *mpz* must have been initialised before calling this function and must be cleared using mpz_clear() to reclaim any storage associated with it.

`int` **PL_get_mpq**(*term_t t, mpq_t mpq*)
If *t* is an integer or rational number (term `rdiv/2`), *mpq* is filled with the *normalised* rational number and the function returns `TRUE`. Otherwise *mpq* is untouched and the function returns `FALSE`. Note that *mpq* must have been initialised before calling this function and must be cleared using mpq_clear() to reclaim any storage associated with it.

`int` **PL_unify_mpz**(*term_t t, mpz_t mpz*)
Unify *t* with the integer value represented by *mpz* and return `TRUE` on success. The *mpz* argument is not changed.

`int` **PL_unify_mpq**(*term_t t, mpq_t mpq*)
Unify *t* with a rational number represented by *mpq* and return `TRUE` on success. Note that *t* is unified with an integer if the denominator is 1. The *mpq* argument is not changed.

9.4.9 Calling Prolog from C

The Prolog engine can be called from C. There are two interfaces for this. For the first, a term is created that could be used as an argument to `call/1`, and then `PL_call()` is used to call Prolog. This system is simple, but does not allow to inspect the different answers to a non-deterministic goal and is relatively slow as the runtime system needs to find the predicate. The other interface is based on `PL_open_query()`, `PL_next_solution()` and `PL_cut_query()` or `PL_close_query()`. This mechanism is more powerful, but also more complicated to use.

Predicate references

This section discusses the functions used to communicate about predicates. Though a Prolog predicate may be defined or not, redefined, etc., a Prolog predicate has a handle that is neither destroyed nor moved. This handle is known by the type `predicate_t`.

`predicate_t` **PL_pred**(*functor_t f, module_t m*)
Return a handle to a predicate for the specified name/arity in the given module. This function always succeeds, creating a handle for an undefined predicate if no handle was available. If the module argument *m* is `NULL`, the current context module is used.

`predicate_t` **PL_predicate**(*const char *name, int arity, const char* module*)
Same as `PL_pred()`, but provides a more convenient interface to the C programmer.

`void` **PL_predicate_info**(*predicate_t p, atom_t *n, int *a, module_t *m*)
Return information on the predicate *p*. The name is stored over *n*, the arity over *a*, while *m* receives the definition module. Note that the latter need not be the same as specified with `PL_predicate()`. If the predicate is imported into the module given to `PL_predicate()`, this function will return the module where the predicate is defined. Any of the arguments *n*, *a* and *m* can be `NULL`.

Initiating a query from C

This section discusses the functions for creating and manipulating queries from C. Note that a foreign context can have at most one active query. This implies that it is allowed to make strictly nested calls between C and Prolog (Prolog calls C, calls Prolog, calls C, etc.), but it is **not** allowed to open multiple queries and start generating solutions for each of them by calling `PL_next_solution()`. Be sure to call `PL_cut_query()` or `PL_close_query()` on any query you opened before opening the next or returning control back to Prolog.

`qid_t` **PL_open_query**(*module_t ctx, int flags, predicate_t p, term_t +t0*)

Opens a query and returns an identifier for it. *ctx* is the *context module* of the goal. When `NULL`, the context module of the calling context will be used, or `user` if there is no calling context (as may happen in embedded systems). Note that the context module only matters for *meta-predicates*. See `meta_predicate/1`, `context_module/1` and `module_transparent/1`. The *p* argument specifies the predicate, and should be the result of a call to `PL_pred()` or `PL_predicate()`. Note that it is allowed to store this handle as global data and reuse it for future queries. The term reference *t0* is the first of a vector of term references as returned by `PL_new_term_refs(`*n*`)`.

The *flags* arguments provides some additional options concerning debugging and exception handling. It is a bitwise *or* of the following values:

`PL_Q_NORMAL`

Normal operation. The debugger inherits its settings from the environment. If an exception occurs that is not handled in Prolog, a message is printed and the tracer is started to debug the error.[4]

`PL_Q_NODEBUG`

Switch off the debugger while executing the goal. This option is used by many calls to hook-predicates to avoid tracing the hooks. An example is `print/1` calling `portray/1` from foreign code.

`PL_Q_CATCH_EXCEPTION`

If an exception is raised while executing the goal, do not report it, but make it available for `PL_exception()`.

`PL_Q_PASS_EXCEPTION`

As `PL_Q_CATCH_EXCEPTION`, but do not invalidate the exception-term while calling `PL_close_query()`. This option is experimental.

`PL_open_query()` can return the query identifier '0' if there is not enough space on the environment stack. This function succeeds, even if the referenced predicate is not defined. In this case, running the query using `PL_next_solution()` will return an existence_error. See `PL_exception()`.

The example below opens a query to the predicate `is_a/2` to find the ancestor of 'me'. The reference to the predicate is valid for the duration of the process and may be cached by the client.

---

[4] Do not pass the integer 0 for normal operation, as this is interpreted as `PL_Q_NODEBUG` for backward compatibility reasons.

```
char *
ancestor(const char *me)
{ term_t a0 = PL_new_term_refs(2);
  static predicate_t p;

  if ( !p )
    p = PL_predicate("is_a", 2, "database");

  PL_put_atom_chars(a0, me);
  PL_open_query(NULL, PL_Q_NORMAL, p, a0);
  ...
}
```

`int` **PL_next_solution**(*qid_t qid*)

Generate the first (next) solution for the given query. The return value is `TRUE` if a solution was found, or `FALSE` to indicate the query could not be proven. This function may be called repeatedly until it fails to generate all solutions to the query.

`void` **PL_cut_query**(*qid_t qid*)

Discards the query, but does not delete any of the data created by the query. It just invalidates *qid*, allowing for a new call to `PL_open_query()` in this context.

`void` **PL_close_query**(*qid_t qid*)

As `PL_cut_query()`, but all data and bindings created by the query are destroyed.

`int` **PL_call_predicate**(*module_t m, int flags, predicate_t pred, term_t +t0*)

Shorthand for `PL_open_query()`, `PL_next_solution()`, `PL_cut_query()`, generating a single solution. The arguments are the same as for `PL_open_query()`, the return value is the same as `PL_next_solution()`.

`int` **PL_call**(*term_t t, module_t m*)

Call term *t* just like the Prolog predicate `once/1`. *t* is called in the module *m*, or in the context module if *m* == NULL. Returns `TRUE` if the call succeeds, `FALSE` otherwise. Figure 9.3 shows an example to obtain the number of defined atoms. All checks are omitted to improve readability.

9.4.10 Discarding Data

The Prolog data created and term references needed to set up the call and/or analyse the result can in most cases be discarded right after the call. `PL_close_query()` allows for destroying the data, while leaving the term references. The calls below may be used to destroy term references and data. See figure 9.3 for an example.

`fid_t` **PL_open_foreign_frame**()

Create a foreign frame, holding a mark that allows the system to undo bindings and destroy data created after it, as well as providing the environment for creating term references. This function is called by the kernel before calling a foreign predicate.

```
int
count_atoms()
{ fid_t fid = PL_open_foreign_frame();
  term_t goal  = PL_new_term_ref();
  term_t a1    = PL_new_term_ref();
  term_t a2    = PL_new_term_ref();
  functor_t s2 = PL_new_functor(PL_new_atom("statistics"), 2);
  int atoms;

  PL_put_atom_chars(a1, "atoms");
  PL_cons_functor(goal, s2, a1, a2);
  PL_call(goal, NULL);         /* call it in current module */

  PL_get_integer(a2, &atoms);
  PL_discard_foreign_frame(fid);

  return atoms;
}
```

Figure 9.3: Calling Prolog

`void` **PL_close_foreign_frame**(*fid_t id*)

Discard all term references created after the frame was opened. All other Prolog data is retained. This function is called by the kernel whenever a foreign function returns control back to Prolog.

`void` **PL_discard_foreign_frame**(*fid_t id*)

Same as `PL_close_foreign_frame()`, but also undo all bindings made since the open and destroy all Prolog data.

`void` **PL_rewind_foreign_frame**(*fid_t id*)

Undo all bindings and discard all term references created since the frame was created, but do not pop the frame. That is, the same frame can be rewound multiple times, and must eventually be closed or discarded.

It is obligatory to call either of the two closing functions to discard a foreign frame. Foreign frames may be nested.

9.4.11 Foreign Code and Modules

Modules are identified via a unique handle. The following functions are available to query and manipulate modules.

`module_t` **PL_context**()

Return the module identifier of the context module of the currently active foreign predicate.

`int` **PL_strip_module**(*term_t +raw, module_t *m, term_t -plain*)

Utility function. If *raw* is a term, possibly holding the module construct ⟨*module*⟩**:**⟨*rest*⟩, this function will make *plain* a reference to ⟨*rest*⟩ and fill *module ** with ⟨*module*⟩. For further nested module constructs the innermost module is returned via *module **. If *raw* is not a module construct, *raw* will simply be put in *plain*. The value pointed to by *m* must be initialized before calling `PL_strip_module()`, either to the default module or to `NULL`. A `NULL` value is replaced by the current context module if *raw* carries no module. The following example shows how to obtain the plain term and module if the default module is the user module:

```
{ module m = PL_new_module(PL_new_atom("user"));
  term_t plain = PL_new_term_ref();

  PL_strip_module(term, &m, plain);
  ...
}
```

`atom_t` **PL_module_name**(*module_t module*)

Return the name of *module* as an atom.

`module_t` **PL_new_module**(*atom_t name*)

Find an existing module or create a new module with the name *name*.

9.4.12 Prolog exceptions in foreign code

This section discusses `PL_exception()`, `PL_throw()` and `PL_raise_exception()`, the interface functions to detect and generate Prolog exceptions from C code. `PL_throw()` and `PL_raise_exception()` from the C interface raise an exception from foreign code. `PL_throw()` exploits the C function longjmp() to return immediately to the innermost `PL_next_solution()`. `PL_raise_exception()` registers the exception term and returns `FALSE`. If a foreign predicate returns `FALSE`, while an exception term is registered, a Prolog exception will be raised by the virtual machine.

Calling these functions outside the context of a function implementing a foreign predicate results in undefined behaviour.

`PL_exception()` may be used after a call to `PL_next_solution()` fails, and returns a term reference to an exception term if an exception was raised, and 0 otherwise.

If a C function implementing a predicate calls Prolog and detects an exception using `PL_exception()`, it can handle this exception or return with the exception. Some caution is required though. It is **not** allowed to call `PL_close_query()` or `PL_discard_foreign_frame()` afterwards, as this will invalidate the exception term. Below is the code that calls a Prolog-defined arithmetic function (see `arithmetic_function/1`).

If `PL_next_solution()` succeeds, the result is analysed and translated to a number, after which the query is closed and all Prolog data created after `PL_open_foreign_frame()` is destroyed. On the other hand, if `PL_next_solution()` fails and if an exception was raised, just pass it. Otherwise generate an exception (`PL_error()` is an internal call for building the standard error terms and calling `PL_raise_exception()`). After this, the Prolog environment should be discarded using `PL_cut_query()` and `PL_close_foreign_frame()` to avoid invalidating the exception term.

```
static int
prologFunction(ArithFunction f, term_t av, Number r)
{ int arity = f->proc->definition->functor->arity;
  fid_t fid = PL_open_foreign_frame();
  qid_t qid;
  int rval;

  qid = PL_open_query(NULL, PL_Q_NORMAL, f->proc, av);

  if ( PL_next_solution(qid) )
  { rval = valueExpression(av+arity-1, r);
    PL_close_query(qid);
    PL_discard_foreign_frame(fid);
  } else
  { term_t except;

    if ( (except = PL_exception(qid)) )
    { rval = PL_throw(except);          /* pass exception */
    } else
    { char *name = stringAtom(f->proc->definition->functor->name);

                                        /* generate exception */
      rval = PL_error(name, arity-1, NULL, ERR_FAILED, f->proc);
    }

    PL_cut_query(qid);                  /* donot destroy data */
    PL_close_foreign_frame(fid);        /* same */
  }

  return rval;
}
```

`int` **PL_raise_exception**(*term_t exception*)

Generate an exception (as `throw/1`) and return `FALSE`. Below is an example returning an exception from a foreign predicate:

```
foreign_t
pl_hello(term_t to)
{ char *s;

  if ( PL_get_atom_chars(to, &s) )
  { Sprintf("Hello \"%s\"\n", s);

    PL_succeed;
  } else
  { term_t except = PL_new_term_ref();

    PL_unify_term(except,
                  PL_FUNCTOR_CHARS, "type_error", 2,
                    PL_CHARS, "atom",
                    PL_TERM, to);

    return PL_raise_exception(except);
  }
}
```

`int` **PL_throw**(*term_t exception*)

Similar to `PL_raise_exception()`, but returns using the C longjmp() function to the innermost `PL_next_solution()`.

`term_t` **PL_exception**(*qid_t qid*)

If `PL_next_solution()` fails, this can be due to normal failure of the Prolog call, or because an exception was raised using `throw/1`. This function returns a handle to the exception term if an exception was raised, or 0 if the Prolog goal simply failed. If there is an exception, `PL_exception()` allocates a term-handle using `PL_new_term_ref()` that is used to return the exception term.

Additionally, `\cfuncref{PL_exception}{0}` returns the pending exception in the current query or 0 if no exception is pending. This can be used to check the error status after a failing call to, e.g., one of the unification functions.

`void` **PL_clear_exception**(*void*)

Tells Prolog that the encountered exception must be ignored. This function must be called if control remains in C after a previous API call fails with an exception.[5]

9.4.13 Catching Signals (Software Interrupts)

SWI-Prolog offers both a C and Prolog interface to deal with software interrupts (signals). The Prolog mapping is defined in section 4.11. This subsection deals with handling signals from C.

If a signal is not used by Prolog and the handler does not call Prolog in any way, the native signal interface routines may be used.

Some versions of SWI-Prolog, notably running on popular Unix platforms, handle `SIG_SEGV` for guarding the Prolog stacks. If the application wishes to handle this signal too, it should use `PL_signal()` to install its handler after initialising Prolog. SWI-Prolog will pass `SIG_SEGV` to the user code if it detected the signal is not related to a Prolog stack overflow.

Any handler that wishes to call one of the Prolog interface functions should call `PL_signal()` for its installation.

`void (*)()` **PL_signal**(*sig, func*)

This function is equivalent to the BSD-Unix signal() function, regardless of the platform used.

---

[5] This feature is non-portable. Other Prolog systems (e.g., YAP) have no facilities to ignore raised exceptions, and the design of YAP's exception handling does not support such a facility.

The signal handler is blocked while the signal routine is active, and automatically reactivated after the handler returns.

After a signal handler is registered using this function, the native signal interface redirects the signal to a generic signal handler inside SWI-Prolog. This generic handler validates the environment, creates a suitable environment for calling the interface functions described in this chapter and finally calls the registered user-handler.

By default, signals are handled asynchronously (i.e., at the time they arrive). It is inherently dangerous to call extensive code fragments, and especially exception related code from asynchronous handlers. The interface allows for *synchronous* handling of signals. In this case the native OS handler just schedules the signal using `PL_raise()`, which is checked by `PL_handle_signals()` at the call- and redo-port. This behaviour is realised by *or*-ing *sig* with the constant `PL_SIGSYNC`.[6]

Signal handling routines may raise exceptions using `PL_raise_exception()`. The use of `PL_throw()` is not safe. If a synchronous handler raises an exception, the exception is delayed to the next call to `PL_handle_signals()`;

`int` **PL_raise**(*int sig*)

Register *sig* for *synchronous* handling by Prolog. Synchronous signals are handled at the call-port or if foreign code calls `PL_handle_signals()`. See also `thread_signal/2`.

`int` **PL_handle_signals**(*void*)

Handle any signals pending from `PL_raise()`. `PL_handle_signals()` is called at each pass through the call- and redo-port at a safe point. Exceptions raised by the handler using `PL_raise_exception()` are properly passed to the environment.

The user may call this function inside long-running foreign functions to handle scheduled interrupts. This routine returns the number of signals handled. If a handler raises an exception, the return value is -1 and the calling routine should return with `FALSE` as soon as possible.

`int` **PL_get_signum_ex**(*term_t t, int *sig*)

Extract a signal specification from a Prolog term and store as an integer signal number in *sig*. The specification is an integer, a lowercase signal name without `SIG` or the full signal name. These refer to the same: `9`, `kill` and `SIGKILL`. Leaves a typed, domain or instantiation error if the conversion fails.

9.4.14 Miscellaneous

Term Comparison

`int` **PL_compare**(*term_t t1, term_t t2*)

Compares two terms using the standard order of terms and returns -1, 0 or 1. See also `compare/3`.

`int` **PL_same_compound**(*term_t t1, term_t t2*)

Yields `TRUE` if *t1* and *t2* refer to physically the same compound term and `FALSE` otherwise.

---

[6] A better default would be to use synchronous handling, but this interface preserves backward compatibility.

Recorded database

In some applications it is useful to store and retrieve Prolog terms from C code. For example, the XPCE graphical environment does this for storing arbitrary Prolog data as slot-data of XPCE objects.

Please note that the returned handles have no meaning at the Prolog level and the recorded terms are not visible from Prolog. The functions `PL_recorded()` and `PL_erase()` are the only functions that can operate on the stored term.

Two groups of functions are provided. The first group (`PL_record()` and friends) store Prolog terms on the Prolog heap for retrieval during the same session. These functions are also used by `recorda/3` and friends. The recorded database may be used to communicate Prolog terms between threads.

`record_t` **PL_record**(*term_t +t*)

Record the term *t* into the Prolog database as `recorda/3` and return an opaque handle to the term. The returned handle remains valid until `PL_erase()` is called on it. `PL_recorded()` is used to copy recorded terms back to the Prolog stack.

`int` **PL_recorded**(*record_t record, term_t -t*)

Copy a recorded term back to the Prolog stack. The same record may be used to copy multiple instances at any time to the Prolog stack. Returns `TRUE` on success, and `FALSE` if there is not enough space on the stack to accommodate the term. See also `PL_record()` and `PL_erase()`.

`void` **PL_erase**(*record_t record*)

Remove the recorded term from the Prolog database, reclaiming all associated memory resources.

The second group (headed by `PL_record_external()`) provides the same functionality, but the returned data has properties that enable storing the data on an external device. It has been designed to make it possible to store Prolog terms fast and compact in an external database. Here are the main features:

- *Independent of session*
  Records can be communicated to another Prolog session and made visible using `PL_recorded_external()`.
- *Binary*
  The representation is binary for maximum performance. The returned data may contain zero bytes.
- *Byte-order independent*
  The representation can be transferred between machines with different byte order.
- *No alignment restrictions*
  There are no memory alignment restrictions and copies of the record can thus be moved freely. For example, it is possible to use this representation to exchange terms using shared memory between different Prolog processes.
- *Compact*
  It is assumed that a smaller memory footprint will eventually outperform slightly faster representations.

- *Stable*
  The format is designed for future enhancements without breaking compatibility with older records.

`char *` **PL_record_external**(*term_t +t, size_t *len*)
Record the term *t* into the Prolog database as `recorda/3` and return an opaque handle to the term. The returned handle remains valid until `PL_erase_external()` is called on it.

It is allowed to copy the data and use `PL_recorded_external()` on the copy. The user is responsible for the memory management of the copy. After copying, the original may be discarded using `PL_erase_external()`.

`PL_recorded_external()` is used to copy such recorded terms back to the Prolog stack.

`int` **PL_recorded_external**(*const char *record, term_t -t*)
Copy a recorded term back to the Prolog stack. The same record may be used to copy multiple instances at any time to the Prolog stack. See also `PL_record_external()` and `PL_erase_external()`.

`int` **PL_erase_external**(*char *record*)
Remove the recorded term from the Prolog database, reclaiming all associated memory resources.

Getting file names

The function `PL_get_file_name()` provides access to Prolog filenames and its file-search mechanism described with `absolute_file_name/3`. Its existence is motivated to realise a uniform interface to deal with file properties, search, naming conventions, etc., from foreign code.

`int` **PL_get_file_name**(*term_t spec, char **name, int flags*)
Translate a Prolog term into a file name. The name is stored in the static buffer ring described with th `PL_get_chars()` option `BUF_RING`. Conversion from the internal UNICODE encoding is done using standard C library functions. *flags* is a bit-mask controlling the conversion process. Options are:

`PL_FILE_ABSOLUTE`
Return an absolute path to the requested file.

`PL_FILE_OSPATH`
Return the name using the hosting OS conventions. On MS-Windows, `\` is used to separate directories rather than the canonical `/`.

`PL_FILE_SEARCH`
Invoke `absolute_file_name/3`. This implies rules from `file_search_path/2` are used.

`PL_FILE_EXIST`
Demand the path to refer to an existing entity.

`PL_FILE_READ`
Demand read-access on the result.

`PL_FILE_WRITE`
Demand write-access on the result.

`PL_FILE_EXECUTE`
Demand execute-access on the result.

`PL_FILE_NOERRORS`
Do not raise any exceptions.

`int` **PL_get_file_nameW**(*term_t spec, wchar_t **name, int flags*)
Same as `PL_get_file_name()`, but returns the filename as a wide-character string. This is intended for Windows to access the Unicode version of the Win32 API. Note that the flag `PL_FILE_OSPATH` must be provided to fetch a filename in OS native (e.g., `C:\x\y`) notation.

9.4.15 Errors and warnings

`PL_warning()` prints a standard Prolog warning message to the standard error (`user_error`) stream. Please note that new code should consider using `PL_raise_exception()` to raise a Prolog exception. See also section 4.10.

`int` **PL_warning**(*format, a1, . . .*)
Print an error message starting with '`[WARNING: `', followed by the output from *format*, followed by a '`]`' and a newline. Then start the tracer. *format* and the arguments are the same as for `printf(2)`. Always returns `FALSE`.

9.4.16 Environment Control from Foreign Code

`int` **PL_action**(*int, ...*)
Perform some action on the Prolog system. *int* describes the action. Remaining arguments depend on the requested action. The actions are listed in table 9.1.

9.4.17 Querying Prolog

`long` **PL_query**(*int*)
Obtain status information on the Prolog system. The actual argument type depends on the information required. *int* describes what information is wanted.[7] The options are given in table 9.2.

9.4.18 Registering Foreign Predicates

`int` **PL_register_foreign_in_module**(*const char *module, const char *name, int arity, foreign_t (*function)(), int flags, ...*
Register a C function to implement a Prolog predicate. After this call returns successfully a predicate with name *name* (a `char *`) and arity *arity* (a C `int`) is created in module *module*. If *module* is `NULL`, the predicate is created in the module of the calling context, or if no context is present in the module `user`.

When called in Prolog, Prolog will call *function*. *flags* form a bitwise *or*'ed list of options for the installation. These are:

---

[7] Returning pointers and integers as a long is bad style. The signature of this function should be changed.

`PL_ACTION_TRACE`	Start Prolog tracer (`trace/0`). Requires no arguments.
`PL_ACTION_DEBUG`	Switch on Prolog debug mode (`debug/0`). Requires no arguments.
`PL_ACTION_BACKTRACE`	

`PL_QUERY_ARGC`	Return an integer holding the number of arguments given to Prolog from Unix.
`PL_QUERY_ARGV`	Return a `char **` holding the argument vector given to Prolog from Unix.
`PL_QUERY_SYMBOLFILE`	Return a `char *` holding the current symbol file of the running process.
`PL_MAX_INTEGER`	Return a long, representing the maximal integer value represented by a Prolog integer.
`PL_MIN_INTEGER`	Return a long, representing the minimal integer value.
`PL_QUERY_VERSION`	Return a long, representing the version as $10,000 \times M + 100 \times m + p$, where M is the major, m the minor version number and p the patch level. For example, `20717` means `2.7.17`.
`PL_QUERY_ENCODING`	Return the default stream encoding of Prolog (of type `IOENC`).
`PL_QUERY_USER_CPU`	Get amount of user CPU time of the process in milliseconds.

Table 9.2: `PL_query()` options

`PL_FA_META`	Provide meta-predicate info (see below)
`PL_FA_TRANSPARENT`	Predicate is module transparent (deprecated)
`PL_FA_NONDETERMINISTIC`	Predicate is non-deterministic. See also `PL_retry()`.
`PL_FA_NOTRACE`	Predicate cannot be seen in the tracer
`PL_FA_VARARGS`	Use alternative calling convention.

If `PL_FA_META` is provided, `PL_register_foreign_in_module()` takes one extra argument. This argument is of type `const char*`. This string must be exactly as long as the number of arguments of the predicate and filled with characters from the set `0-9:^-+?`. See `meta_predicate/1` for details. `PL_FA_TRANSPARENT` is implied if at least one meta-argument is provided (`0-9:^`). Note that meta-arguments are *not always* passed as ⟨*module*⟩:⟨*term*⟩. Always use `PL_strip_module()` to extract the module and plain term from a meta-argument.[8]

Predicates may be registered either before or after `PL_initialise()`. When registered before initialisation the registration is recorded and executed after installing the system predicates and before loading the saved state.

Default calling (i.e. without `PL_FA_VARARGS`) *function* is passed the same number of `term_t` arguments as the arity of the predicate and, if the predicate is non-deterministic, an extra argument of type `control_t` (see section 9.4.1). If `PL_FA_VARARGS` is provided, *function* is called with three arguments. The first argument is a `term_t` handle to the first argument. Further arguments can be reached by adding the offset (see also `PL_new_term_refs()`). The second argument is the arity, which defines the number of valid term references in the argument vector. The last argument is used for non-deterministic calls. It is currently undocumented and should be defined of type `void*`. Here is an example:

---

[8] It is encouraged to pass an additional `NULL` pointer for non-meta-predicates.

```
static foreign_t
atom_checksum(term_t a0, int arity, void* context)
{ char *s;

  if ( PL_get_atom_chars(a0, &s) )
  { int sum;

    for(sum=0; *s; s++)
      sum += *s&0xff;

    return PL_unify_integer(a0+1, sum&0xff);
  }

  return FALSE;
}

install_t
install()
{ PL_register_foreign("atom_checksum", 2, atom_checksum, PL_FA_VARARGS);
}
```

`int` **PL_register_foreign**(*const char *name, int arity, foreign_t (*function)(), int flags, ...*)

Same as `PL_register_foreign_in_module()`, passing `NULL` for the *module*.

`void` **PL_register_extensions_in_module**(*const char *module, PL_extension *e*)

Register a series of predicates from an array of definitions of the type `PL_extension` in the given *module*. If *module* is `NULL`, the predicate is created in the module of the calling context, or if no context is present in the module `user`. The `PL_extension` type is defined as

```
typedef struct PL_extension
{ char          *predicate_name;        /* Name of the predicate */
  short         arity;                  /* Arity of the predicate */
  pl_function_t function;               /* Implementing functions */
  short         flags;                  /* Or of PL_FA_... */
} PL_extension;
```

For details, see `PL_register_foreign_in_module()`. Here is an example of its usage:

```
static PL_extension predicates[] = {
{ "foo",        1,      pl_foo, 0 },
{ "bar",        2,      pl_bar, PL_FA_NONDETERMINISTIC },
{ NULL,         0,      NULL,   0 }
};

main(int argc, char **argv)
```

```
{ PL_register_extensions_in_module("user", predicates);

  if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  ...
}
```

`void` **PL_register_extensions**(*PL_extension *e*)

Same as `PL_register_extensions_in_module()` using `NULL` for the *module* argument.

9.4.19 Foreign Code Hooks

For various specific applications some hooks are provided.

`PL_dispatch_hook_t` **PL_dispatch_hook**(*PL_dispatch_hook_t*)

If this hook is not NULL, this function is called when reading from the terminal. It is supposed to dispatch events when SWI-Prolog is connected to a window environment. It can return two values: `PL_DISPATCH_INPUT` indicates Prolog input is available on file descriptor 0 or `PL_DISPATCH_TIMEOUT` to indicate a timeout. The old hook is returned. The type `PL_dispatch_hook_t` is defined as:

```
typedef int  (*PL_dispatch_hook_t)(void);
```

`void` **PL_abort_hook**(*PL_abort_hook_t*)

Install a hook when `abort/0` is executed. SWI-Prolog `abort/0` is implemented using C setjmp()/longjmp() construct. The hooks are executed in the reverse order of their registration after the longjmp() took place and before the Prolog top-level is reinvoked. The type `PL_abort_hook_t` is defined as:

```
typedef void (*PL_abort_hook_t)(void);
```

`int` **PL_abort_unhook**(*PL_abort_hook_t*)

Remove a hook installed with `PL_abort_hook()`. Returns `FALSE` if no such hook is found, `TRUE` otherwise.

`void` **PL_on_halt**(*void (*f)(int, void *), void *closure*)

Register the function *f* to be called if SWI-Prolog is halted. The function is called with two arguments: the exit code of the process (0 if this cannot be determined on your operating system) and the *closure* argument passed to the `PL_on_halt()` call. See also `at_halt/1`.

`PL_agc_hook_t` **PL_agc_hook**(*PL_agc_hook_t new*)

Register a hook with the atom-garbage collector (see `garbage_collect_atoms/0`) that is called on any atom that is reclaimed. The old hook is returned. If no hook is currently defined, `NULL` is returned. The argument of the called hook is the atom that is to be garbage collected.

The return value is an `int`. If the return value is zero, the atom is **not** reclaimed. The hook may invoke any Prolog predicate.

The example below defines a foreign library for printing the garbage collected atoms for debugging purposes.

```
#include <SWI-Stream.h>
#include <SWI-Prolog.h>

static int
atom_hook(atom_t a)
{ Sdprintf("AGC: deleting %s\n", PL_atom_chars(a));

  return TRUE;
}

static PL_agc_hook_t old;

install_t
install()
{ old = PL_agc_hook(atom_hook);
}

install_t
uninstall()
{ PL_agc_hook(old);
}
```

9.4.20 Storing foreign data

When combining foreign code with Prolog, it can be necessary to make data represented in the foreign language available to Prolog. For example, to pass it to another foreign function. At the end of this section, there is a partial implementation of using foreign functions to manage bit-vectors. Another example is the SGML/XML library that manages a 'parser' object, an object that represents the current state of the parser and that can be directed to perform actions such as parsing a document or make queries about the document content.

This section provides some hints for handling foreign data in Prolog. There are four options for storing such data:

- *Natural Prolog data*
  Uses the representation one would choose if no foreign interface was required. E.g., a bitvector representing a list of small integers can be represented as a Prolog list of integers.

- *Opaque packed data on the stacks*
  It is possible to represent the raw binary representation of the foreign object as a Prolog string (see section 4.23). Strings may be created from foreign data using `PL_put_string_nchars()` and retrieved using `PL_get_string_chars()`. It is good

practice to wrap the string in a compound term with arity 1, so Prolog can identify the type. The hook `portray/1` rules may be used to streamline printing such terms during development.

- *Opaque packed data in a blob*
  Similar to the above solution, binary data can be stored in an atom. The blob interface (section 9.4.7) provides additional facilities to assign a type and hook-functions that act on creation and destruction of the underlying atom.
- *Natural foreign data, passed as a pointer*
  An alternative is to pass a pointer to the foreign data. Again, the pointer is often wrapped in a compound term.

The choice may be guided using the following distinctions

- *Is the data opaque to Prolog*
  With 'opaque' data, we refer to data handled in foreign functions, passed around in Prolog, but where Prolog never examines the contents of the data itself. If the data is opaque to Prolog, the selection will be driven solely by simplicity of the interface and performance.
- *What is the lifetime of the data*
  With 'lifetime' we refer to how it is decided that the object is (or can be) destroyed. We can distinguish three cases:
  1. The object must be destroyed on backtracking and normal Prolog garbage collection (i.e., it acts as a normal Prolog term). In this case, representing the object as a Prolog string (second option above) is the only feasible solution.
  2. The data must survive Prolog backtracking. This leaves two options. One is to represent the object using a pointer and use explicit creation and destruction, making the programmer responsible. The alternative is to use the blob-interface, leaving destruction to the (atom) garbage collector.
  3. The data lives as during the lifetime of a foreign function that implements a predicate. If the predicate is deterministic, foreign automatic variables are suitable. If the predicate is non-deterministic, the data may be allocated using malloc() and a pointer may be passed. See section 9.4.1.

Examples for storing foreign data

In this section, we outline some examples, covering typical cases. In the first example, we will deal with extending Prolog's data representation with integer sets, represented as bit-vectors. Then, we discuss the outline of the DDE interface.

Integer sets with not-too-far-apart upper- and lower-bounds can be represented using bit-vectors. Common set operations, such as union, intersection, etc., are reduced to simple *and*'ing and *or*'ing the bit-vectors. This can be done using Prolog's unbounded integers.

For really demanding applications, foreign representation will perform better, especially time-wise. Bit-vectors are naturally expressed using string objects. If the string is wrapped in `bitvector/1`, the lower-bound of the vector is 0 and the upper-bound is not defined; an implementation for getting and putting the sets as well as the union predicate for it is below.

```
#include <SWI-Prolog.h>

#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) < (b) ? (a) : (b))

static functor_t FUNCTOR_bitvector1;

static int
get_bitvector(term_t in, int *len, unsigned char **data)
{ if ( PL_is_functor(in, FUNCTOR_bitvector1) )
  { term_t a = PL_new_term_ref();

    PL_get_arg(1, in, a);
    return PL_get_string(a, (char **)data, len);
  }

  PL_fail;
}

static int
unify_bitvector(term_t out, int len, const unsigned char *data)
{ if ( PL_unify_functor(out, FUNCTOR_bitvector1) )
  { term_t a = PL_new_term_ref();

    PL_get_arg(1, out, a);

    return PL_unify_string_nchars(a, len, (const char *)data);
  }

  PL_fail;
}

static foreign_t
pl_bitvector_union(term_t t1, term_t t2, term_t u)
{ unsigned char *s1, *s2;
  int l1, l2;

  if ( get_bitvector(t1, &l1, &s1) &&
       get_bitvector(t2, &l2, &s2) )
  { int l = max(l1, l2);
    unsigned char *s3 = alloca(l);

    if ( s3 )
    { int n;
      int ml = min(l1, l2);
      for(n=0; n<ml; n++)
        s3[n] = s1[n] | s2[n];
      for( ; n < l1; n++)
        s3[n] = s1[n];
      for( ; n < l2; n++)
        s3[n] = s2[n];

      return unify_bitvector(u, l, s3);
    }

    return PL_warning("Not enough memory");
  }

  PL_fail;
}


install_t
install()
{ PL_register_foreign("bitvector_union", 3, pl_bitvector_union, 0);

  FUNCTOR_bitvector1 = PL_new_functor(PL_new_atom("bitvector"), 1);
}
```

The DDE interface (see section 4.41) represents another common usage of the foreign interface: providing communication to new operating system features. The DDE interface requires knowledge about active DDE server and client channels. These channels contains various foreign data types. Such an interface is normally achieved using an open/close protocol that creates and destroys a *handle*. The handle is a reference to a foreign data structure containing the relevant information.

There are a couple of possibilities for representing the handle. The choice depends on responsibilities and debugging facilities. The simplest approach is to use `PL_unify_pointer()` and `PL_get_pointer()`. This approach is fast and easy, but has the drawbacks of (untyped) pointers: there is no reliable way to detect the validity of the pointer, nor to verify that it is pointing to a structure of the desired type. The pointer may be wrapped into a compound term with arity 1 (i.e., `dde_channel(`⟨*Pointer*⟩`)`), making the type-problem less serious.

Alternatively (used in the DDE interface), the interface code can maintain a (preferably variable length) array of pointers and return the index in this array. This provides better protection. Especially for debugging purposes, wrapping the handle in a compound is a good suggestion.

9.4.21 Embedding SWI-Prolog in other applications

With embedded Prolog we refer to the situation where the 'main' program is not the Prolog application. Prolog is sometimes embedded in C, C++, Java or other languages to provide logic based services in a larger application. Embedding loads the Prolog engine as a library to the external language. Prolog itself only provides for embedding in the C language (compatible with C++). Embedding in Java is achieved using JPL using a C-glue between the Java and Prolog C interfaces.

The most simple embedded program is below. The interface function `PL_initialise()` **must** be called before any of the other SWI-Prolog foreign language functions described in this chapter, except for `PL_initialise_hook()`, `PL_new_atom()`, `PL_new_functor()` and `PL_register_foreign()`. `PL_initialise()` interprets all the command line arguments, except for the `-t` *`toplevel`* flag that is interpreted by `PL_toplevel()`.

```
int
main(int argc, char **argv)
{
#ifdef READLINE /* Remove if you don't want readline */
  PL_initialise_hook(install_readline);
#endif

  if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  PL_halt(PL_toplevel() ? 0 : 1);
}
```

`int` **PL_initialise**(*int argc, char **argv*)

Initialises the SWI-Prolog heap and stacks, restores the Prolog state, loads the system and personal initialisation files, runs the `at_initialization/1` hooks and finally runs the `-g` *`goal`* hook.

Special consideration is required for `argv[0]`. On **Unix**, this argument passes the part of the command line that is used to locate the executable. Prolog uses this to find the file holding the running executable. The **Windows** version uses this to find a *module* of the running executable. If the specified module cannot be found, it tries the module `libpl.dll`, containing the Prolog runtime kernel. In all these cases, the resulting file is used for two purposes:

- See whether a Prolog saved state is appended to the file. If this is the case, this state will be loaded instead of the default `boot.prc` file from the SWI-Prolog home directory. See also `qsave_program/[1,2]` and section 9.5.
- Find the Prolog home directory. This process is described in detail in section 9.6.

`PL_initialise()` returns 1 if all initialisation succeeded and 0 otherwise.[9]

In most cases, *argc* and *argv* will be passed from the main program. It is allowed to create your own argument vector, provided `argv[0]` is constructed according to the rules above. For example:

```
int
main(int argc, char **argv)
{ char *av[10];
  int ac = 0;
```

---

[9] BUG: Various fatal errors may cause `PL_initialise()` to call `PL_halt(1)`, preventing it from returning at all.

```
  av[ac++] = argv[0];
  av[ac++] = "-x";
  av[ac++] = "mystate";
  av[ac]   = NULL;

  if ( !PL_initialise(ac, av) )
    PL_halt(1);
  ...
}
```

Please note that the passed argument vector may be referred from Prolog at any time and should therefore be valid as long as the Prolog engine is used.

A good setup in Windows is to add SWI-Prolog's `bin` directory to your `PATH` and either pass a module holding a saved state, or `"libpl.dll"` as `argv[0]`. If the Prolog state is attached to a DLL (see the `-dll` option of `swipl-ld`), pass the name of this DLL.

`int` **PL_is_initialised**(*int *argc, char ***argv*)

Test whether the Prolog engine is already initialised. Returns `FALSE` if Prolog is not initialised and `TRUE` otherwise. If the engine is initialised and *argc* is not `NULL`, the argument count used with `PL_initialise()` is stored in *argc*. Same for the argument vector *argv*.

`void` **PL_install_readline**()

Installs the GNU readline line editor. Embedded applications that do not use the Prolog toplevel should normally delete this line, shrinking the Prolog kernel significantly. Note that the Windows version does not use GNU readline.

`int` **PL_toplevel**()

Runs the goal of the `-t` *`toplevel`* switch (default `prolog/0`) and returns 1 if successful, 0 otherwise.

`int` **PL_cleanup**(*int status*)

This function performs the reverse of `PL_initialise()`. It runs the `PL_on_halt()` and `at_halt/1` handlers, closes all streams (except for the 'standard I/O' streams which are flushed only), deallocates all memory and restores all signal handlers. The *status* argument is passed to the various termination hooks and indicates the *exit-status*.

The function returns `TRUE` if successful and `FALSE` otherwise. Currently, `FALSE` is returned when an attempt is made to call `PL_cleanup()` recursively or if `PL_cleanup()` is not called from the main thread.

In theory, this function allows deleting and restarting the Prolog system in the same process. In practice, SWI-Prolog's cleanup process is far from complete, and trying to revive the system using `PL_initialise()` will leak memory in the best case. It can also crash the appliction.

In this state, there is little practical use for this function. If you want to use Prolog temporarily, consider running it in a separate process. If you want to be able to reset Prolog, your options are (again) a separate process, modules or threads.

`void` **PL_cleanup_fork**()

Close file descriptors associated to Prolog streams except for 0,1 and 2. Stop intervaltimer that may be running on behalf of `profile/1`. The call is intended to be used in combination with fork():

```
if ( (pid=fork()) == 0 )
{ PL_cleanup_fork();
  <some exec variation>
}
```

The call behaves the same on Windows, though there is probably no meaningful application.

`int` **PL_halt**(*int status*)
Clean up the Prolog environment using `PL_cleanup()` and call exit() with the status argument. As `PL_cleanup()` can only be called from the main thread, this function returns `FALSE` when called from a thread other than the main one.[10]

Threading, Signals and embedded Prolog

This section applies to Unix-based environments that have signals or multi-threading. The Windows version is compiled for multi-threading, and Windows lacks proper signals.

We can distinguish two classes of embedded executables. There are small C/C++-programs that act as an interfacing layer around Prolog. Most of these programs can be replaced using the normal Prolog executable extended with a dynamically loaded foreign extension and in most cases this is the preferred route. In other cases, Prolog is embedded in a complex application that—like Prolog—wants to control the process environment. A good example is Java. Embedding Prolog is generally the only way to get these environments together in one process image. Java applications, however, are by nature multi-threaded and appear to do signal handling (software interrupts).

On Unix systems, SWI-Prolog uses three signals:

SIGUSR1 is used to sychronise atom and clause garbage collection. The handler is installed at the start of garbage collection and reverted to the old setting after completion.

SIGUSR2 has an empty signal handler. This signal is sent to a thread after sending a thread-signal (see `thread_signal/2`). It causes blocking system calls to return with `EINTR`, which gives them the opportunity to react to thread-signals.

SIGINT is used by the top-level to activate the tracer (typically bound to control-C). The first control-C posts a request for starting the tracer in a safe, synchronous fashion. If control-C is hit again before the safe route is executed, it prompts the user whether or not a forced interrupt is desired.

The `--nosignals` option can be used to inhibit processing of `SIGINT`. The other signals are vital for the functioning of SWI-Prolog. If they conflict with other applications, signal handling of either component must be modified. The SWI-Prolog signals are defined in `pl-thread.h` of the source distribution.

---

[10] BUG: Eventually it may become possible to call `PL_halt()` from any thread.

9.5 Linking embedded applications using swipl-ld

The utility program `swipl-ld` (Win32: swipl-ld.exe) may be used to link a combination of C files and Prolog files into a stand-alone executable. `swipl-ld` automates most of what is described in the previous sections.

In normal usage, a copy is made of the default embedding template `.../swipl/include/stub.c`. The main() routine is modified to suit your application. `PL_initialise()` **must** be passed the program name (*argv[0]*) (Win32: the executing program can be obtained using GetModuleFileName()). The other elements of the command line may be modified. Next, `swipl-ld` is typically invoked as:

```
swipl-ld -o output stubfile.c [other-c-or-o-files] [plfiles]
```

`swipl-ld` will first split the options into various groups for both the C compiler and the Prolog compiler. Next, it will add various default options to the C compiler and call it to create an executable holding the user's C code and the Prolog kernel. Then, it will call the SWI-Prolog compiler to create a saved state from the provided Prolog files and finally, it will attach this saved state to the created emulator to create the requested executable.

Below, it is described how the options are split and whichx which6C6 m 441 /F7-290(()]TJ-5(ecutab253 1.9091 T-hel

-dll

Windows only. Embed SWI-Prolog into a DLL rather than an executable.

-c

Compile C or C++ source files into object files. This turns `swipl-ld` into a replacement for the C or C++ compiler, where proper options such as the location of the include directory are passed automatically to the compiler.

-E

Invoke the C preprocessor. Used to make `swipl-ld` a replacement for the C or C++ compiler.

-pl-options *,...*

Additional options passed to Prolog when creating the saved state. The first character immediately following `pl-options` is used as separator and translated to spaces when the argument is built. Example: `-pl-options,-F,xpce` passes `-F xpce` as additional flags to Prolog.

-ld-options *,...*

Passes options to the linker, similar to `-pl-options`.

-cc-options *,...*

Passes options to the C/C++ compiler, similar to `-pl-options`.

-v

Select verbose operation, showing the various programs and their options.

-o *outfile*

Reserved to specify the final output file.

-l*library*

Specifies a library for the C compiler. By default, `-lswipl` (Win32: libpl.lib) and the libraries needed by the Prolog kernel are given.

-L*library-directory*

Specifies a library directory for the C compiler. By default the directory containing the Prolog C library for the current architecture is passed.

`-g` | `-I`*include-directory* | `-D`*definition*

These options are passed to the C compiler. By default, the include directory containing `SWI-Prolog.h` is passed. `swipl-ld` adds two additional `*` -*D*def flags:

> **-D**`__SWI_PROLOG__`
>
> Indicates the code is to be connected to SWI-Prolog.
>
> **-D**`__SWI_EMBEDDED__`
>
> Indicates the creation of an embedded program.

**.o* | **.c* | **.C* | **.cxx* | **.cpp*

Passed as input files to the C compiler.

.pl* |.qlf*

Passed as input files to the Prolog compiler to create the saved state.

`*`

All other options. These are passed as linker options to the C compiler.

9.5.1 A simple example

The following is a very simple example going through all the steps outlined above. It provides an arithmetic expression evaluator. We will call the application `calc` and define it in the files `calc.c` and `calc.pl`. The Prolog file is simple:

```
calc(Atom) :-
        term_to_atom(Expr, Atom),
        A is Expr,
        write(A),
        nl.
```

The C part of the application parses the command line options, initialises the Prolog engine, locates the `calc/1` predicate and calls it. The coder is in figure 9.4.
The application is now created using the following command line:

```
% swipl-ld -o calc calc.c calc.pl
```

The following indicates the usage of the application:

```
% calc pi/2
1.5708
```

9.6 The Prolog ‘home’ directory

Executables embedding SWI-Prolog should be able to find the ‘home’ directory of the development environment unless a self-contained saved state has been added to the executable (see `qsave_program/[1,2]` and section 9.5).

If Prolog starts up, it will try to locate the development environment. To do so, it will try the following steps until one succeeds:

1. If the `--home=DIR` is provided, use this.
2. If the environment variable `SWI_HOME_DIR` is defined and points to an existing directory, use this.
3. If the environment variable `SWIPL` is defined and points to an existing directory, use this.
4. Locate the primary executable or (Windows only) a component (*module*) thereof and check whether the parent directory of the directory holding this file contains the file `swipl`. If so, this file contains the (relative) path to the home directory. If this directory exists, use this. This is the normal mechanism used by the binary distribution.
5. If the precompiled path exists, use it. This is only useful for a source installation.

If all fails and there is no state attached to the executable or provided Windows module (see `PL_initialise()`), SWI-Prolog gives up. If a state is attached, the current working directory is used.

The `file_search_path/2` alias `swi` is set to point to the home directory located.

```
#include <stdio.h>
#include <SWI-Prolog.h>

#define MAXLINE 1024

int
main(int argc, char **argv)
{ char expression[MAXLINE];
  char *e = expression;
  char *program = argv[0];
  char *plav[2];
  int n;

  /* combine all the arguments in a single string */

  for(n=1; n<argc; n++)
  { if ( n != 1 )
      *e++ = ’ ’;
    strcpy(e, argv[n]);
    e += strlen(e);
  }

  /* make the argument vector for Prolog */

  plav[0] = program;
  plav[1] = NULL;

  /* initialise Prolog */

  if ( !PL_initialise(1, plav) )
    PL_halt(1);

  /* Lookup calc/1 and make the arguments and call */

  { predicate_t pred = PL_predicate("calc", 1, "user");
    term_t h0 = PL_new_term_refs(1);
    int rval;

    PL_put_atom_chars(h0, expression);
    rval = PL_call_predicate(NULL, PL_Q_NORMAL, pred, h0);

    PL_halt(rval ? 0 : 1);
  }

  return 0;
}
```

Figure 9.4: C source for the calc application

9.7 Example of Using the Foreign Interface

Below is an example showing all stages of the declaration of a foreign predicate that transforms atoms possibly holding uppercase letters into an atom only holding lowercase letters. Figure 9.5 shows the C source file, figure 9.6 illustrates compiling and loading of foreign code.

```
/*  Include file depends on local installation */
#include <SWI-Prolog.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

foreign_t
pl_lowercase(term_t u, term_t l)
{ char *copy;
  char *s, *q;
  int rval;

  if ( !PL_get_atom_chars(u, &s) )
    return PL_warning("lowercase/2: instantiation fault");
  copy = malloc(strlen(s)+1);

  for( q=copy; *s; q++, s++)
    *q = (isupper(*s) ? tolower(*s) : *s);
  *q = ’\0’;

  rval = PL_unify_atom_chars(l, copy);
  free(copy);

  return rval;
}

install_t
install()
{ PL_register_foreign("lowercase", 2, pl_lowercase, 0);
}
```

Figure 9.5: Lowercase source file

```
% gcc -I/usr/local/lib/swipl-\plversion/include -fpic -c lowercase.c
% gcc -shared -o lowercase.so lowercase.o
% swipl
Welcome to SWI-Prolog (Version \plversion)
...

1 ?- load_foreign_library(lowercase).

Yes
2 ?- lowercase('Hello World!', L).

L = 'hello world!'

Yes
```

Figure 9.6: Compiling the C source and loading the object file

9.8 Notes on Using Foreign Code

9.8.1 Memory Allocation

SWI-Prolog's heap memory allocation is based on the `malloc(3)` library routines. SWI-Prolog provides the functions below as a wrapper around malloc(). Allocation errors in these functions trap SWI-Prolog's fatal-error handler, in which case `PL_malloc()` or `PL_realloc()` do not return.

Portable applications must use `PL_free()` to release strings returned by `PL_get_chars()` using the `BUF_MALLOC` argument. Portable applications may use both `PL_malloc()` and friends or malloc() and friends but should not mix these two sets of functions on the same memory.

`void *` **PL_malloc**(*size_t bytes*)

Allocate *bytes* of memory. On failure SWI-Prolog's fatal-error handler is called and `PL_malloc()` does not return. Memory allocated using these functions must use `PL_realloc()` and `PL_free()` rather than realloc() and free().

`void *` **PL_realloc**(*void *mem, size_t size*)

Change the size of the allocated chunk, possibly moving it. The *mem* argument must be obtained from a previous `PL_malloc()` or `PL_realloc()` call.

`void` **PL_free**(*void *mem*)

Release memory. The *mem* argument must be obtained from a previous `PL_malloc()` or `PL_realloc()` call.

Boehm-GC support

To accommodate future use of the Boehm garbage collector[11] for heap memory allocation, the interface provides the functions described below. Foreign extensions that wish to use the Boehm-GC facilities can use these wrappers. Please note that if SWI-Prolog is not compiled to use Boehm-GC (default), the user is responsible for calling `PL_free()` to reclaim memory.

`void*` **PL_malloc_atomic**(*size_t bytes*)
`void*` **PL_malloc_uncollectable**(*size_t bytes*)
`void*` **PL_malloc_atomic_uncollectable**(*size_t bytes*)

If Boehm-GC is not used, these are all the same as `PL_malloc()`. With Boehm-GC, these map to the corresponding Boehm-GC functions. *Atomic* means that the content should not be scanned for pointers, and *uncollectable* means that the object should never be garbage collected.

`void*` **PL_malloc_stubborn**(*size_t bytes*)
`void` **PL_end_stubborn_change**(*void *memory*)

These functions allow creating objects, promising GC that the content will not change after `PL_end_stubborn_change()`.

9.8.2 Compatibility between Prolog versions

Great care is taken to ensure binary compatibility of foreign extensions between different Prolog versions. Only the much less frequently used stream interface has been responsible for binary incompatibilities.

---

[11] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

Source code that relies on new features of the foreign interface can use the macro `PLVERSION` to find the version of `SWI-Prolog.h` and `PL_query()` using the option `PL_QUERY_VERSION` to find the version of the attached Prolog system. Both follow the same numbering schema explained with `PL_query()`.

9.8.3 Debugging and profiling foreign code (valgrind)

This section is only relevant for Unix users on platforms supported by valgrind. Valgrind is an excellent binary instrumentation platform. Unlike many other instrumentation platforms, valgrind can deal with code loaded through dlopen().

The `callgrind` tool can be used to profile foreign code loaded under SWI-Prolog. Compile the foreign library adding `-g` option to `gcc` or `swipl-ld`. By setting the environment variable `VALGRIND` to `yes`, SWI-Prolog will *not* release loaded shared objects using dlclose(). This trick is required to get source information on the loaded library. Without, valgrind claims that the shared object has no debugging information.[12] Here is the complete sequence using `bash` as login shell:

```
% VALGRIND=yes valgrind --tool=callgrind pl <args>
<prolog interaction>
% kcachegrind callgrind.out.<pid>
```

9.8.4 Name Conflicts in C modules

In the current version of the system all public C functions of SWI-Prolog are in the symbol table. This can lead to name clashes with foreign code. Someday I should write a program to strip all these symbols from the symbol table (why does Unix not have that?). For now I can only suggest you give your function another name. You can do this using the C preprocessor. If—for example—your foreign package uses a function warning(), which happens to exist in SWI-Prolog as well, the following macro should fix the problem:

```
#define warning warning_
```

Note that shared libraries do not have this problem as the shared library loader will only look for symbols in the main executable for symbols that are not defined in the library itself.

9.8.5 Compatibility of the Foreign Interface

The term reference mechanism was first used by Quintus Prolog version 3. SICStus Prolog version 3 is strongly based on the Quintus interface. The described SWI-Prolog interface is similar to using the Quintus or SICStus interfaces, defining all foreign-predicate arguments of type `+term`. SWI-Prolog explicitly uses type `functor_t`, while Quintus and SICStus use ⟨*name*⟩ and ⟨*arity*⟩. As the names of the functions differ from Prolog to Prolog, a simple macro layer dealing with the names can also deal with this detail. For example:

```
#define QP_put_functor(t, n, a) PL_put_functor(t, PL_new_functor(n, a))
```

---

[12] Tested using valgrind version 3.2.3 on x64.

The `PL_unify_*()` functions are lacking from the Quintus and SICStus interface. They can easily be emulated, or the put/unify approach should be used to write compatible code.

The `PL_open_foreign_frame()`/`PL_close_foreign_frame()` combination is lacking from both other Prologs. SICStus has `PL_new_term_refs(`*0*`)`, followed by `PL_reset_term_refs()`, that allows for discarding term references.

The Prolog interface for the graphical user interface package XPCE shares about 90% of the code using a simple macro layer to deal with different naming and calling conventions of the interfaces.

Generating Runtime Applications

10

This chapter describes the features of SWI-Prolog for delivering applications that can run without the development version of the system installed.

A SWI-Prolog runtime executable is a file consisting of two parts. The first part is the *emulator*, which is machine-dependent. The second part is the *resource archive*, which contains the compiled program in a machine-independent format, startup options and possibly user-defined *resources*; see `resource/3` and `open_resource/3`.

These two parts can be connected in various ways. The most common way for distributed runtime applications is to *concatenate* the two parts. This can be achieved using external commands (Unix: `cat`, Windows: `copy`), or using the `stand_alone` option to `qsave_program/2`. The second option is to attach a startup script in front of the resource that starts the emulator with the proper options. This is the default under Unix. Finally, an emulator can be told to use a specified resource file using the `-x` command-line switch.

qsave_program(+*File*, +*Options***)**

Saves the current state of the program to the file *File*. The result is a resource archive containing a saved state that expresses all Prolog data from the running program and all user-defined resources. Depending on the `stand_alone` option, the resource is headed by the emulator, a Unix shell script or nothing. *Options* is a list of additional options:

local(+*KBytes***)**

Limit for the local stack. See section 2.4.1.

global(+*KBytes***)**

Limit for the global stack. See section 2.4.1.

trail(+*KBytes***)**

Limit for the trail stack. See section 2.4.1.

goal(:*Callable***)**

Initialization goal for the new executable (see `-g`).

toplevel(:*Callable***)**

Top-level goal for the new executable (see `-t`).

init_file(+*Atom***)**

Default initialization file for the new executable. See `-f`.

class(+*Class***)**

If `runtime`, only read resources from the state (default). If `kernel`, lock all predicates as system predicates. If `development`, save the predicates in their current state and keep reading resources from their source (if present). See also `resource/3`.

autoload(+*Boolean***)**

If `true` (default), run `autoload/0` first.

map(+*File***)**

Dump a human-readable trace of what has been saved in *File*.

op(+*Action***)**

One of `save` (default) to save the current operator table or `standard` to use the initial table of the emulator.

stand_alone(+*Boolean***)**

If `true`, the emulator is the first part of the state. If the emulator is started it will test whether a boot-file (state) is attached to the emulator itself and load this state. Provided the application has all libraries loaded, the resulting executable is completely independent of the runtime environment or location where it was built. See also section 2.10.2.

emulator(+*File***)**

File to use for the emulator. Default is the running Prolog image.

foreign(+*Action***)**

If `save`, include shared objects (DLLs) into the saved state. See `current_foreign_library/2`. If the program `strip` is available, this is first used to reduce the size of the shared object. If a state is started, `use_foreign_library/1` first tries to locate the foreign resource in the executable. When found it copies the content of the resource to a temporary file and loads it. If possible (Unix), the temporary object is deleted immediately after opening.[1]

qsave_program(+*File***)**

Equivalent to `qsave_program(File, [])`.

autoload

Check the current Prolog program for predicates that are referred to, are undefined and have a definition in the Prolog library. Load the appropriate libraries.

This predicate is used by `qsave_program/[1,2]` to ensure the saved state does not depend on availability of the libraries. The predicate `autoload/0` examines all clauses of the loaded program (obtained with `clause/2`) and analyses the body for referenced goals. Such an analysis cannot be complete in Prolog, which allows for the creation of arbitrary terms at runtime and the use of them as a goal. The current analysis is limited to the following:

- Direct goals appearing in the body
- Arguments of declared meta-predicates that are marked with an integer (0..9). See `meta_predicate/1`.

The analysis of meta-predicate arguments is limited to cases where the argument appears literally in the clause or is assigned using =/2 before the meta-call. That is, the following fragment is processed correctly:

```
        ...,
        Goal = prove(Theory),
        forall(current_theory(Theory),
               Goal)),
```

---

[1] This option is experimental and currently disabled by default. It will become the default if it proves robust.

But, the calls to `prove_simple/1` and `prove_complex/1` in the example below are *not* discovered by the analysis and therefore the modules that define these predicates must be loaded explicitly using `use_module/1`,2.

```
        ...,
        member(Goal, [ prove_simple(Theory),
                       prove_complex(Theory)
                     ]),
        forall(current_theory(Theory),
               Goal)),
```

It is good practice to use `gxref/0` to make sure that the program has sufficient declarations such that the analaysis tools can verify that all required predicates can be resolved and that all code is called. See `meta_predicate/1`, `dynamic/1`, `public/1` and prolog:called_by/2.

volatile *+Name/Arity, ...*

Declare that the clauses of specified predicates should **not** be saved to the program. The volatile declaration is normally used to prevent the clauses of dynamic predicates that represent data for the current session from being saved in the state file.

10.1 Limitations of qsave_program

There are three areas that require special attention when using `qsave_program/[1,2]`.

- If the program is an embedded Prolog application or uses the foreign language interface, care has to be taken to restore the appropriate foreign context. See section 10.2 for details.
- If the program uses directives (`:- goal.` lines) that perform other actions than setting predicate attributes (dynamic, volatile, etc.) or loading files (consult, etc.), the directive may need to be prefixed with `initialization/1`.
- Database references as returned by `clause/3`, `recorded/3`, etc., are not preserved and may thus not be part of the database when saved.

10.2 Runtimes and Foreign Code

Some applications may need to use the foreign language interface. Object code is by definition machine-dependent and thus cannot be part of the saved program file.

To complicate the matter even further there are various ways of loading foreign code:

- *Using the library(shlib) predicates*
  This is the preferred way of dealing with foreign code. It loads quickly and ensures an acceptable level of independence between the versions of the emulator and the foreign code loaded. It works on Unix machines supporting shared libraries and library functions to load them. Most modern Unixes, as well as Win32 (Windows 95/NT), satisfy this constraint.

- *Static linking*
  This mechanism works on all machines, but generally requires the same C compiler and linker to be used for the external code as is used to build SWI-Prolog itself.

To make a runtime executable that can run on multiple platforms one must make runtime checks to find the correct way of linking. Suppose we have a source file `myextension.c` defining the installation function install().

If this file is compiled into a shared library, `load_foreign_library/1` will load this library and call the installation function to initialise the foreign code. If it is loaded as a static extension, define install() as the predicate `install/0`:

```
static foreign_t
pl_install()
{ install();

  PL_succeed;
}

PL_extension PL_extensions [] =
{
/*{ "name",     arity,  function,       PL_FA_<flags> },*/

  { "install",  0,      pl_install,     0 },
  { NULL,       0,      NULL,           0 }     /* terminating line */
};
```

Now, use the following Prolog code to load the foreign library:

```
load_foreign_extensions :-
        current_predicate(install, install), !, % static loaded
        install.
load_foreign_extensions :-                      % shared library
        load_foreign_library(foreign(myextension)).

:- initialization load_foreign_extensions.
```

The path alias `foreign` is defined by `file_search_path/2`. By default it searches the directories ⟨*home*⟩`/lib/`⟨*arch*⟩ and ⟨*home*⟩`/lib`. The application can specify additional rules for `file_search_path/2`.

10.3 Using program resources

A *resource* is very similar to a file. Resources, however, can be represented in two different formats: on files, as well as part of the resource *archive* of a saved state (see `qsave_program/2`).

A resource has a *name* and a *class*. The *source* data of the resource is a file. Resources are declared by declaring the predicate `resource/3`. They are accessed using the predicate `open_resource/3`.

Before going into details, let us start with an example. Short texts can easily be expressed in Prolog source code, but long texts are cumbersome. Assume our application defines a command ‘help’ that prints a helptext to the screen. We put the content of the helptext into a file called `help.txt`. The following code implements our help command such that `help.txt` is incorporated into the runtime executable.

```
resource(help, text, ’help.txt’).

help :-
        open_resource(help, text, In),
        call_cleanup(copy_stream_data(In, user_output),
                     close(In)).
```

The predicate `help/0` opens the resource as a Prolog stream. If we are executing this from the development environment, this will actually return a stream to the file `help.txt` itself. When executed from the saved state, the stream will actually be a stream opened on the program resource file, taking care of the offset and length of the resource.

10.3.1 Predicate Definitions

resource(*+Name, +Class, +FileSpec***)**

This predicate is defined as a dynamic predicate in the module `user`. Clauses for it may be defined in any module, including the user module. *Name* is the name of the resource (an atom). A resource name may contain any character, except for $ and :, which are reserved for internal usage by the resource library. *Class* describes the kind of object stored in the resource. In the current implementation, it is just an atom. *FileSpec* is a file specification that may exploit `file_search_path/2` (see `absolute_file_name/2`).

Normally, resources are defined as unit clauses (facts), but the definition of this predicate also allows for rules. For proper generation of the saved state, it must be possible to enumerate the available resources by calling this predicate with all its arguments unbound.

Dynamic rules are useful to turn all files in a certain directory into resources, without specifying a resource for each file. For example, assume the `file_search_path/2` `icons` refers to the resource directory containing icon-files. The following definition makes all these images available as resources:

```
resource(Name, image, icons(XpmName)) :-
        atom(Name), !,
        file_name_extension(Name, xpm, XpmName).
resource(Name, image, XpmFile) :-
        var(Name),
        absolute_file_name(icons(.), [type(directory)], Dir)
        concat(Dir, ’/*.xpm’, Pattern),
        expand_file_name(Pattern, XpmFiles),
        member(XpmFile, XpmFiles).
```

open_resource(+*Name, ?Class, -Stream***)**

Opens the resource specified by *Name* and *Class*. If the latter is a variable, it will be unified to the class of the first resource found that has the specified *Name*. If successful, *Stream* becomes a handle to a binary input stream, providing access to the content of the resource.

The predicate `open_resource/3` first checks `resource/3`. When successful it will open the returned resource source file. Otherwise it will look in the program's resource database. When creating a saved state, the system normally saves the resource contents into the resource archive, but does not save the resource clauses.

This way, the development environment uses the files (and modifications) to the `resource/3` declarations and/or files containing resource info, thus immediately affecting the running environment, while the runtime system quickly accesses the system resources.

10.3.2 The `swipl-rc` program

The utility program `swipl-rc` can be used to examine and manipulate the contents of a SWI-Prolog resource file. The options are inspired by the Unix `ar` program. The basic command is:

```
% swipl-rc option resource-file member ...
```

The options are described below.

l

List contents of the archive.

x

Extract named (or all) members of the archive into the current directory.

a

Add files to the archive. If the archive already contains a member with the same name, the contents are replaced. Anywhere in the sequence of members, the options `--class=`*class* and `--encoding=`*encoding* may appear. They affect the class and encoding of subsequent files. The initial class is `data` and encoding `none`.

d

Delete named members from the archive.

This command is also described in the `pl(1)` Unix manual page.

10.4 Finding Application files

If your application uses files that are not part of the saved program such as database files, configuration files, etc., the runtime version has to be able to locate these files. The `file_search_path/2` mechanism in combination with the `-p`alias command-line argument is the preferred way to locate runtime files. The first step is to define an alias for the top-level directory of your application. We will call this directory `gnatdir` in our examples.

A good place for storing data associated with SWI-Prolog runtime systems is below the emulator's home directory. `swi` is a predefined alias for this directory. The following is a useful default definition for the search path.

```
user:file_search_path(gnatdir, swi(gnat)).
```

The application should locate all files using absolute_file_name. Suppose `gnatdir` contains a file `config.pl` to define the local configuration. Then use the code below to load this file:

```
configure_gnat :-
        (   absolute_file_name(gnatdir('config.pl'), ConfigFile)
        ->  consult(ConfigFile)
        ;   format(user_error, 'gnat: Cannot locate config.pl~n'),
            halt(1)
        ).
```

10.4.1 Passing a path to the application

Suppose the system administrator has installed the SWI-Prolog runtime environment in `/usr/local/lib/rt-pl-3.2.0`. A user wants to install `gnat`, but `gnat` will look for its configuration in `/usr/local/lib/rt-pl-3.2.0/gnat` where the user cannot write.

The user decides to install the `gnat` runtime files in `/users/bob/lib/gnat`. For one-time usage, the user may decide to start `gnat` using the command:

```
% gnat -p gnatdir=/users/bob/lib/gnat
```

The SWI-Prolog library A

This chapter documents the SWI-Prolog library. As SWI-Prolog provides auto-loading, there is little difference between library predicates and built-in predicates. Part of the library is therefore documented in the rest of the manual. Library predicates differ from built-in predicates in the following ways:

- User definition of a built-in leads to a permission error, while using the name of a library predicate is allowed.
- If autoloading is disabled explicitly or because trapping unknown predicates is disabled (see `unknown/2` and `current_prolog_flag/2`), library predicates must be loaded explicitly.
- Using libraries reduces the footprint of applications that don't need them.

> *The documentation of the library has just started. Material from the standard packages should be moved here, some material from other parts of the manual should be moved too and various libraries are not documented at all.*

A.1 library(aggregate): Aggregation operators on backtrackable predicates

Compatibility Quintus, SICStus 4. The `forall/2` is a SWI-Prolog built-in and `term_variables/3` is a SWI-Prolog with a **different definition**.

To be done
- Analysing the aggregation template and compiling a predicate for the list aggregation can be done at compile time.
- `aggregate_all/3` can be rewritten to run in constant space using non-backtrackable assignment on a term.

This library provides aggregating operators over the solutions of a predicate. The operations are a generalisation of the `bagof/3`, `setof/3` and `findall/3` built-in predicates. The defined aggregation operations are counting, computing the sum, minimum, maximum, a bag of solutions and a set of solutions. We first give a simple example, computing the country with the smallest area:

```
smallest_country(Name, Area) :-
        aggregate(min(A, N), country(N, A), min(Area, Name)).
```

There are four aggregation predicates, distinguished on two properties.

aggregate vs. aggregate_all The aggregate predicates use `setof/3` (`aggregate/4`) or `bagof/3` (`aggregate/3`), dealing with existential qualified variables (Var^Goal) and providing multiple solutions for the remaining free variables in Goal. The `aggregate_all/3` predicate uses `findall/3`, implicitly qualifying all free variables and providing exactly one solution, while `aggregate_all/4` uses `sort/2` over solutions and Distinguish (see below) generated using `findall/3`.

The Distinguish argument The versions with 4 arguments provide a Distinguish argument that allow for keeping duplicate bindings of a variable in the result. For example, if we wish to compute the total population of all countries we do not want to lose results because two countries have the same population. Therefore we use:

```
    aggregate(sum(P), Name, country(Name, P), Total)
```

All aggregation predicates support the following operator below in Template. In addition, they allow for an arbitrary named compound term where each of the arguments is a term from the list below. I.e. the term r(min(X), max(X)) computes both the minimum and maximum binding for X.

count
Count number of solutions. Same as sum(1).

sum(*Expr*)
Sum of Expr for all solutions.

min(*Expr*)
Minimum of Expr for all solutions.

min(*Expr, Witness*)
A term min(Min, Witness), where Min is the minimal version of Expr over all Solution and Witness is any other template applied to Solution that produced Min. If multiple solutions provide the same minimum, Witness corresponds to the first solution.

max(*Expr*)
Maximum of Expr for all solutions.

max(*Expr, Witness*)
As min(Expr, Witness), but producing the maximum result.

set(*X*)
An ordered set with all solutions for X.

bag(*X*)
A list of all solutions for X.

Acknowledgements

The development of this library was sponsored by SecuritEase, `http://www.securitease.com`

aggregate(*+Template, :Goal, -Result***)** *[nondet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate/3` version performs `bagof/3` on *Goal*.

aggregate(*+Template, +Discriminator, :Goal, -Result***)** *[nondet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate/3` version performs `setof/3` on *Goal*.

aggregate_all(*+Template, :Goal, -Result***)** *[semidet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate_all/3` version performs `findall/3` on *Goal*.

aggregate_all(*+Template, +Discriminator, :Goal, -Result***)** *[semidet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate_all/3` version performs `findall/3` followed by `sort/2` on *Goal*.

foreach(*:Generator, :Goal***)**

True if the conjunction of instances of *Goal* using the bindings from *Generator* is true. Unlike `forall/2`, which runs a failure-driven loop that proves *Goal* for each solution of *Generator*, foreach creates a conjunction. Each member of the conjunction is a copy of *Goal*, where the variables it shares with *Generator* are filled with the values from the corresponding solution.

The implementation executes `forall/2` if *Goal* does not contain any variables that are not shared with *Generator*.

Here is an example:

```
?- foreach(between(1,4,X), dif(X,Y)), Y = 5.
Y = 5
?- foreach(between(1,4,X), dif(X,Y)), Y = 3.
No
```

bug *Goal* is copied repeatetly, which may cause problems if attributed variables are involved.

free_variables(*:Generator, +Template, +VarList0, -VarList***)** *[det]*

In order to handle variables properly, we have to find all the universally quantified variables in the *Generator*. All variables as yet unbound are universally quantified, unless

1. they occur in the template
2. they are bound by X^P, setof, or bagof

free_variables(*Generator*, *Template*, OldList, NewList) finds this set, using OldList as an accumulator.

author
- Richard O'Keefe
- Jan Wielemaker (made some SWI-Prolog enhancements)

license Public domain (from DEC10 library).

To be done
- Distinguish between control-structures and data terms.
- Exploit our built-in `term_variables/2` at some places?

A.2 library(apply): Apply predicates on a list

See also
- `apply_macros.pl` provides compile-time expansion for part of this library.
- `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm`

To be done Add `include/4`, `include/5`, `exclude/4`, `exclude/5`

This module defines meta-predicates that apply a predicate on all members of a list.

include(*:Goal, +List1, ?List2***)** *[det]*
Filter elements for which *Goal* succeed. True if *List2* contains those elements Xi of *List1* for which call(*Goal*, Xi) succeeds.

See also Older versions of SWI-Prolog had `sublist/3` with the same arguments and semantics.

exclude(*:Goal, +List1, ?List2***)** *[det]*
Filter elements for which *Goal* fails. True if *List2* contains those elements Xi of *List1* for which call(*Goal*, Xi) fails.

partition(*:Pred, +List, ?Included, ?Excluded***)** *[det]*
Filter elements of *List* according to *Pred*. True if *Included* contains all elements for which call(*Pred*, X) succeeds and *Excluded* contains the remaining elements.

partition(*:Pred, +List, ?Less, ?Equal, ?Greater***)** *[semidet]*
Filter list according to *Pred* in three sets. For each element Xi of *List*, its destination is determined by call(*Pred*, Xi, Place), where Place must be unified to one of <, = or >. *Pred* must be deterministic.

maplist(*:Goal, ?List***)**
True if *Goal* can succesfully be applied on all elements of *List*. Arguments are reordered to gain performance as well as to make the predicate deterministic under normal circumstances.

maplist(*:Goal, ?List1, ?List2***)**
True if *Goal* can succesfully be applied to all succesive pairs of elements of *List1* and *List2*.

maplist(*:Goal, ?List1, ?List2, ?List3***)**
True if *Goal* can succesfully be applied to all succesive triples of elements of *List1..List3*.

maplist(*:Goal, ?List1, ?List2, ?List3, ?List4***)**
True if *Goal* can succesfully be applied to all succesive quadruples of elements of *List1..List4*

A.3 `assoc`: Association lists

Authors: *Richard A. O'Keefe, L.Damas, V.S.Costa and Markus Triska*

Elements of an association list have 2 components: A (unique) *key* and a *value*. Keys should be ground, values need not be. An association list can be used to fetch elements via their keys and to enumerate its elements in ascending order of their keys. The `assoc` module uses AVL trees to implement association lists. This makes inserting, changing and fetching a single element an O(log(N)) (where N denotes the number of elements in the list) expected time (and worst-case time) operation.

assoc_to_list(*+Assoc, -List***)**
List is a list of Key-Value pairs corresponding to the associations in *Assoc* in ascending order of keys.

assoc_to_keys(*+Assoc, -List***)**
List is a list of Keys corresponding to the associations in *Assoc* in ascending order.

assoc_to_values(*+Assoc, -List***)**
List is a list of Values corresponding to the associations in *Assoc* in ascending order of the keys they are associated to.

empty_assoc(*-Assoc***)**
Assoc is unified with an empty association list.

gen_assoc(*?Key, +Assoc, ?Value***)**
Enumerate matching elements of *Assoc* in ascending order of their keys via backtracking.

get_assoc(*+Key, +Assoc, ?Value***)**
Value is the value associated with *Key* in the association list *Assoc*.

get_assoc(*+Key, +Assoc, ?Old, ?NewAssoc, ?New***)**
NewAssoc is an association list identical to *Assoc* except that the value associated with *Key* is *New* instead of *Old*.

list_to_assoc(*+List, ?Assoc***)**
Assoc is an association list corresponding to the Key-Value pairs in *List*.

map_assoc(*:Goal, +Assoc***)**
Goal(V) is true for every value V in *Assoc*.

map_assoc(*:Goal, +AssocIn, ?AssocOut***)**
AssocOut is *AssocIn* with *Goal* applied to all corresponding pairs of values.

max_assoc(*+Assoc, ?Key, ?Value***)**
Key and *Value* are key and value of the element with the largest key in *Assoc*.

min_assoc(*+Assoc, ?Key, ?Value***)**
Key and *Value* are key and value of the element with the smallest key in *Assoc*.

ord_list_to_assoc(*+List, ?Assoc***)**
Assoc is an association list correspond to the Key-Value pairs in *List*, which must occur in ascending order of their keys.

put_assoc(*+Key, +Assoc, +Value, ?NewAssoc***)**
NewAssoc is an association list identical to *Assoc* except that *Key* is associated with *Value*. This can be used to insert and change associations.

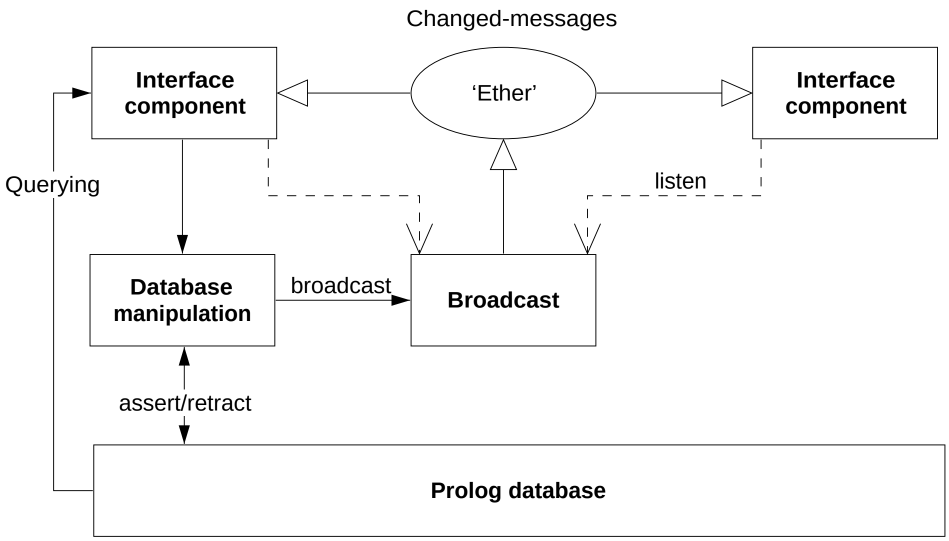


Figure A.1: Information-flow using broadcasting service

A.4 `broadcast`: Broadcast and receive event notifications

The `broadcast` library was invented to realise GUI applications consisting of stand-alone components that use the Prolog database for storing the application data. Figure A.1 illustrates the flow of information using this design

The broadcasting service provides two services. Using the 'shout' service, an unknown number of agents may listen to the message and act. The broadcaster is not (directly) aware of the implications. Using the 'request' service, listening agents are asked for an answer one-by-one and the broadcaster is allowed to reject answers using normal Prolog failure.

Shouting is often used to inform about changes made to a common database. Other messages can be "save yourself" or "show this".

Requesting is used to get information while the broadcaster is not aware who might be able to answer the question. For example "who is showing X?".

broadcast(+*Term*)

Broadcast *Term*. There are no limitations to *Term*, though being a global service, it is good practice to use a descriptive and unique principal functor. All associated goals are started and regardless of their success or failure, `broadcast/1` always succeeds. Exceptions are passed.

broadcast_request(+*Term*)

Unlike `broadcast/1`, this predicate stops if an associated goal succeeds. Backtracking causes it to try other listeners. A broadcast request is used to fetch information without knowing the identity of the agent providing it. C.f. "Is there someone who knows the age of John?" could be asked using

```
        ...,
        broadcast_request(age_of('John', Age)),
```

If there is an agent (*listener*) that registered an 'age-of' service and knows about the age of 'John' this question will be answered.

listen(+*Template, :Goal***)**

Register a *listen* channel. Whenever a term unifying *Template* is broadcasted, call *Goal*. The following example traps all broadcasted messages as a variable unifies to any message. It is commonly used to debug usage of the library.

```
?- listen(Term, (writeln(Term),fail)).
?- broadcast(hello(world)).
hello(world)

Yes
```

listen(+*Listener, +Template, :Goal***)**

Declare *Listener* as the owner of the channel. Unlike a channel opened using `listen/2`, channels that have an owner can terminate the channel. This is commonly used if an object is listening to broadcast messages. In the example below we define a 'name-item' displaying the name of an identifier represented by the predicate `name_of/2`.

```
:- pce_begin_class(name_item, text_item).

variable(id,    any,    get, "Id visualised").

initialise(NI, Id:any) :->
        name_of(Id, Name),
        send_super(NI, initialise, name, Name,
                   message(NI, set_name, @arg1)),
        send(NI, slot, id, Id),
        listen(NI, name_of(Id, Name),
               send(NI, selection, Name)).

unlink(NI) :->
        unlisten(NI),
        send_super(NI, unlink).

set_name(NI, Name:name) :->
        get(NI, id, Id),
        retractall(name_of(Id, _)),
        assert(name_of(Id, Name)),
        broadcast(name_of(Id, Name)).

:- pce_end_class.
```

unlisten(+*Listener***)**

Deregister all entries created with `listen/3` whose *Listener* unify.

unlisten(+*Listener, +Template***)**

Deregister all entries created with `listen/3` whose *Listener* and *Template* unify.

unlisten(*+Listener, +Template, :Goal***)**
Deregister all entries created with `listen/3` whose *Listener*, *Template* and *Goal* unify.

listening(*?Listener, ?Template, ?Goal***)**
Examine the current listeners. This predicate is useful for debugging purposes.

A.5 library(charsio): I/O on Lists of Character Codes

Compatibility The naming of this library is not in line with the ISO standard. We believe that the SWI-Prolog native predicates form a more elegant alternative for this library.

This module emulates the Quintus/SICStus library `charsio.pl` for reading and writing from/to lists of character codes. Most of these predicates are straight calls into similar SWI-Prolog primitives. Some can even be replaced by ISO standard predicates.

format_to_chars(*+Format, +Args, -Codes***)** *[det]*
Use `format/2` to write to a list of character codes.

format_to_chars(*+Format, +Args, -Codes***)** *[det]*
Use `format/2` to write to a difference list of character codes.

write_to_chars(*+Term, -Codes***)**
Codes is a list of character codes produced by `write/1` on *Term*.

write_to_chars(*+Term, -Codes, ?Tail***)**
Codes is a difference-list of character codes produced by `write/1` on *Term*.

atom_to_chars(*+Atom, -Codes***)** *[det]*
Convert *Atom* into a list of character codes.

deprecated Use ISO `atom_codes/2`.

atom_to_chars(*+Atom, -Codes, ?Tail***)** *[det]*
Convert *Atom* into a difference-list of character codes.

number_to_chars(*+Number, -Codes***)** *[det]*
Convert Atom into a list of character codes.

deprecated Use ISO `number_codes/2`.

number_to_chars(*+Atom, -Codes, ?Tail***)** *[det]*
Convert Number into a difference-list of character codes.

read_from_chars(*+Codes, -Term***)** *[det]*
Read *Codes* into *Term*.

Compatibility The SWI-Prolog version does not require *Codes* to end in a full-stop.

read_term_from_chars(*+Codes, -Term, +Options***)** *[det]*
Read *Codes* into *Term*. *Options* are processed by `read_term/3`.

Compatibility sicstus

open_chars_stream(+*Codes, -Stream***)** *[det]*

Open *Codes* as an input stream.

bug Depends on autoloading library(memfile). As many applications do not need this predicate we do not want to make the entire library dependent on autoloading.

with_output_to_chars(:*Goal, -Codes***)** *[det]*

Run *Goal* with as `once/1`. Output written to `current_output` is collected in *Codes*.

with_output_to_chars(:*Goal, -Codes, ?Tail***)** *[det]*

Run *Goal* with as `once/1`. Output written to `current_output` is collected in *Codes**Tail*.

with_output_to_chars(:*Goal, -Stream, -Codes, ?Tail***)** *[det]*

As `with_output_to_chars/2`, but *Stream* is unified with the temporary stream.

A.6 `check`: Elementary completeness checks

This library defines the predicate `check/0` and a few friends that allow for a quick-and-dirty cross-referencing.

check

Performs the three checking passes implemented by `list_undefined/0`, `list_autoload/0` and `list_redefined/0`. Please check the definition of these predicates for details.

The typical usage of this predicate is right after loading your program to get a quick overview on the completeness and possible conflicts in your program.

list_undefined

Scans the database for predicates that have no definition. A predicate is considered defined if it has clauses or is declared using `dynamic/1` or `multifile/1`. As a program is compiled, calls are translated to predicates. If the called predicate is not yet defined it is created as a predicate without definition. The same happens with runtime generated calls. This predicate lists all such undefined predicates that are referenced and not defined in the library. See also `list_autoload/0`. Below is an example from a real program and an illustration of how to edit the referencing predicate using `edit/1`.

```
?- list_undefined.
Warning: The predicates below are not defined. If these are defined
Warning: at runtime using assert/1, use :- dynamic Name/Arity.
Warning:
Warning: rdf_edit:rdfe_retract/4, which is referenced by
Warning:         1-st clause of rdf_edit:undo/4
Warning: rdf_edit:rdfe_retract/3, which is referenced by
Warning:         1-st clause of rdf_edit:delete_object/1
Warning:         1-st clause of rdf_edit:delete_subject/1
Warning:         1-st clause of rdf_edit:delete_predicate/1
```

```
?- edit(rdf_edit:undo/4).
```

list_autoload
Lists all undefined (see `list_undefined/0`) predicates that have a definition in the library along with the file from which they will be autoloaded when accessed. See also `autoload/0`.

list_redefined
Lists predicates that are defined in the global module `user` as well as in a normal module; that is, predicates for which the local definition overrules the global default definition.

A.7 library(clpfd): Constraint Logic Programming over Finite Domains

author Markus Triska

Constraint programming is a declarative formalism that lets you describe conditions a solution must satisfy. This library provides CLP(FD), Constraint Logic Programming over Finite Domains. It can be used to model and solve various combinatorial problems such as planning, scheduling and allocation tasks.

Most predicates of this library are finite domain *constraints*, which are relations over integers. They generalise arithmetic evaluation of integer expressions in that propagation can proceed in all directions. This library also provides *enumeration predicates*, which let you systematically search for solutions on variables whose domains have become finite. A finite domain *expression* is one of:

an integer	Given value
a variable	Unknown value
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr ^ Expr	Exponentiation
min(Expr,Expr)	Minimum of two expressions
max(Expr,Expr)	Maximum of two expressions
Expr mod Expr	Modulo induced by floored division
Expr rem Expr	Modulo induced by truncated division
abs(Expr)	Absolute value
Expr / Expr	Truncated integer division

The most important finite domain constraints are:

Expr1 #>= Expr2	Expr1 is larger than or equal to Expr2
Expr1 #=< Expr2	Expr1 is smaller than or equal to Expr2
Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #> Expr2	Expr1 is strictly larger than Expr2
Expr1 #< Expr2	Expr1 is strictly smaller than Expr2

The constraints `in/2`, `#=/2`, `#\=/2`, `#</2`, `#>/2`, `#=</2`, and `#>=/2` can be *reified*, which means reflecting their truth values into Boolean values represented by the integers 0 and 1. Let P and Q denote reifiable constraints or Boolean variables, then:

#\ Q	True iff Q is false
P #\/ Q	True iff either P or Q
P #/\ Q	True iff both P and Q
P #<==> Q	True iff P and Q are equivalent
P #==> Q	True iff P implies Q
P #<== Q	True iff Q implies P

The constraints of this table are reifiable as well. If a variable occurs at the place of a constraint that is being reified, it is implicitly constrained to the Boolean values 0 and 1. Therefore, the following queries all fail: `?- #\` 2., `?- #\ #\` 2. etc.

Here is an example session with a few queries and their answers:

```
?- [library(clpfd)].
% library(clpfd) compiled into clpfd 0.06 sec, 3,308 bytes
true.

?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19\/21..sup.

?- 2*X #= 10.
X = 5.

?- X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs), X = 1, Y #\= 2.
Vs = [1, 3, 2],
X = 1,
Y = 3,
Z = 2.

?- X #= Y #<==> B, X in 0..3, Y in 4..5.
B = 0,
X in 0..3,
Y in 4..5.
```

In each case (and as for all pure programs), the answer is declaratively equivalent to the original query, and in many cases the constraint solver has deduced additional domain restrictions.

A common usage of this library is to first post the desired constraints among the variables of a model, and then to use enumeration predicates to search for solutions. As an example of a constraint satisfaction problem, consider the cryptoarithmetic puzzle SEND + MORE = MONEY, where different letters denote distinct integers between 0 and 9. It can be modeled in CLP(FD) as follows:

```
:- use_module(library(clpfd)).

puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
        all_different(Vars),
                  S*1000 + E*100 + N*10 + D +
                  M*1000 + O*100 + R*10 + E #=
        M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0.
```

Sample query and its result:

```
?- puzzle(As+Bs=Cs).
As = [9, _G10107, _G10110, _G10113],
Bs = [1, 0, _G10128, _G10107],
Cs = [1, 0, _G10110, _G10107, _G10152],
_G10107 in 4..7,
1000*9+91*_G10107+ -90*_G10110+_G10113+ -9000*1+ -900*0+10*_G10128+ -1*_G10152#=0
all_different([_G10107, _G10110, _G10113, _G10128, _G10152, 0, 1, 9]),
_G10110 in 5..8,
_G10113 in 2..8,
_G10128 in 2..8,
_G10152 in 2..8.
```

Here, the constraint solver has deduced more stringent bounds for all variables. Keeping the modeling part separate from the search lets you view these residual goals, observe termination and determinism properties of the modeling part in isolation from the search, and more easily experiment with different search strategies. Labeling can then be used to search for solutions:

```
?- puzzle(As+Bs=Cs), label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.
```

In this case, it suffices to label a subset of variables to find the puzzle's unique solution, since the constraint solver is strong enough to reduce the domains of remaining variables to singleton sets. In general though, it is necessary to label all variables to obtain ground solutions.

You can also use CLP(FD) constraints as a more declarative alternative for ordinary integer arithmetic with `is/2`, `>/2` etc. For example:

```
:- use_module(library(clpfd)).

n_factorial(0, 1).
n_factorial(N, F) :- N #> 0, N1 #= N - 1, F #= N * F1, n_factorial(N1, F1).
```

This predicate can be used in all directions. For example:

```
?- n_factorial(47, F).
F = 258623241511168180642964355153611979969197632389120000000000 ;
false.

?- n_factorial(N, 1).
N = 0 ;
N = 1 ;
false.

?- n_factorial(N, 3).
false.
```

To make the predicate terminate if any argument is instantiated, add the (implied) constraint F `#\=` 0 before the recursive call. Otherwise, the query n_factorial(N, 0) is the only non-terminating case of this kind.

This library uses `goal_expansion/2` to rewrite constraints at compilation time. The expansion's aim is to transparently bring the performance of CLP(FD) constraints close to that of conventional arithmetic predicates (`</2`, `=:=/2`, `is/2` etc.) when the constraints are used in modes that can also be handled by built-in arithmetic. To disable the expansion, set the flag clpfd_goal_expansion to false.

Use `call_residue_vars/2` and `copy_term/3` to inspect residual goals and the constraints in which a variable is involved. This library also provides *reflection* predicates (like `fd_dom/2`, `fd_size/2` etc.) with which you can inspect a variable's current domain. These predicates can be useful if you want to implement your own labeling strategies.

You can also define custom constraints. The mechanism to do this is not yet finalised, and we welcome suggestions and descriptions of use cases that are important to you. As an example of how it can be done currently, let us define a new custom constraint ”oneground(X,Y,Z)”, where Z shall be 1 if at least one of X and Y is instantiated:

```
:- use_module(library(clpfd)).

:- multifile clpfd:run_propagator/2.

oneground(X, Y, Z) :-
        clpfd:make_propagator(oneground(X, Y, Z), Prop),
        clpfd:init_propagator(X, Prop),
        clpfd:init_propagator(Y, Prop),
        clpfd:trigger_once(Prop).

clpfd:run_propagator(oneground(X, Y, Z), MState) :-
        (   integer(X) -> clpfd:kill(MState), Z = 1
        ;   integer(Y) -> clpfd:kill(MState), Z = 1
        ;   true
        ).
```

First, `clpfd:make_propagator/2` is used to transform a user-defined representation of the new constraint to an internal form. With `clpfd:init_propagator/2`, this internal form is then attached to X and Y. From now on, the propagator will be invoked whenever the domains of X or Y are changed. Then, `clpfd:trigger_once/1` is used to give the propagator its first chance for propagation even though the variables' domains have not yet changed. Finally, `clpfd:run_propagator/2` is extended to define the actual propagator. As explained, this predicate is automatically called by the constraint solver. The first argument is the user-defined representation of the constraint as used in `clpfd:make_propagator/2`, and the second argument is a mutable state that can be used to prevent further invocations of the propagator when the constraint has become entailed, by using `clpfd:kill/1`. An example of using the new constraint:

```
?- oneground(X, Y, Z), Y = 5.
Y = 5,
Z = 1,
X in inf..sup.
```

You can cite this library in your publications as:

```
@inproceedings{Triska08,
  author    = {Markus Triska},
  title     = {Generalising Constraint Solving over Finite Domains},
  booktitle = {ICLP},
  year      = {2008},
  pages     = {820-821}
}
```

?Var **in** *+Domain*

Var is an element of *Domain*. *Domain* is one of:

Integer

Singleton set consisting only of *Integer*.

Lower **..** *Upper*

All integers *I* such that *Lower* =< *I* =< *Upper*. *Lower* must be an integer or the atom **inf**, which denotes negative infinity. *Upper* must be an integer or the atom **sup**, which denotes positive infinity.

Domain1 **\/** *Domain2*
The union of Domain1 and Domain2.

+Vars **ins** *+Domain*
The variables in the list *Vars* are elements of *Domain*.

indomain(*?Var***)**
Bind *Var* to all feasible values of its domain on backtracking. The domain of *Var* must be finite.

label(*+Vars***)**
Equivalent to labeling([], *Vars*).

labeling(*+Options, +Vars***)**
Labeling means systematically trying out values for the finite domain variables *Vars* until all of them are ground. The domain of each variable in *Vars* must be finite. *Options* is a list of options that let you exhibit some control over the search process. Several categories of options exist:

The variable selection strategy lets you specify which variable of *Vars* is labeled next and is one of:

leftmost
Label the variables in the order they occur in *Vars*. This is the default.

ff
First fail. Label the leftmost variable with smallest domain next, in order to detect infeasibility early. This is often a good strategy.

ffc
Of the variables with smallest domains, the leftmost one participating in most constraints is labeled next.

min
Label the leftmost variable whose lower bound is the lowest next.

max
Label the leftmost variable whose upper bound is the highest next.

The value order is one of:

up
Try the elements of the chosen variable's domain in ascending order. This is the default.

down
Try the domain elements in descending order.

The branching strategy is one of:

step
For each variable X, a choice is made between X = V and X #\= V, where V is determined by the value ordering options. This is the default.

enum
For each variable X, a choice is made between X = V_1, X = V_2 etc., for all values V_i of the domain of X. The order is determined by the value ordering options.

bisect
For each variable X, a choice is made between X `#=<` M and X `#>` M, where M is the midpoint of the domain of X.

At most one option of each category can be specified, and an option must not occur repeatedly.

The order of solutions can be influenced with:

min(*Expr***)**

max(*Expr***)**

This generates solutions in ascending/descending order with respect to the evaluation of the arithmetic expression Expr. Labeling *Vars* must make Expr ground. If several such options are specified, they are interpreted from left to right, e.g.:

```
?- [X,Y] ins 10..20, labeling([max(X),min(Y)],[X,Y]).
```

This generates solutions in descending order of X, and for each binding of X, solutions are generated in ascending order of Y. To obtain the incomplete behaviour that other systems exhibit with "maximize(Expr)" and "minimize(Expr)", use `once/1`, e.g.:

```
once(labeling([max(Expr)], Vars))
```

Labeling is always complete, always terminates, and yields no redundant solutions.

all_different(*+Vars***)**
Vars are pairwise distinct.

all_distinct(*+Ls***)**
Like `all_different/1`, with stronger propagation. For example, `all_distinct/1` can detect that not all variables can assume distinct values given the following domains:

```
?- maplist(in, Vs, [1\/3..4, 1..2\/4, 1..2\/4, 1..3, 1..3, 1..6]), all_distin
false.
```

sum(*+Vars, +Rel, ?Expr***)**
The sum of elements of the list *Vars* is in relation *Rel* to *Expr*, where *Rel* is #=, #\=, #<, #>, `#=<` or #>=. For example:

```
?- [A,B,C] ins 0..sup, sum([A,B,C], #=, 100).
A in 0..100,
A+B+C#=100,
B in 0..100,
C in 0..100.
```

scalar_product(+*Cs*, +*Vs*, +*Rel*, ?*Expr***)**
Cs is a list of integers, *Vs* is a list of variables and integers. True if the scalar product of *Cs* and *Vs* is in relation *Rel* to *Expr*, where *Rel* is #=, #\=, #<, #>, `#=<` or #>=.

?*X* **#>=** ?*Y*
X is greater than or equal to *Y*.

?*X* **#=<** ?*Y*
X is less than or equal to *Y*.

?*X* **#=** ?*Y*
X equals *Y*.

?*X* **#\=** ?*Y*
X is not *Y*.

?*X* **#>** ?*Y*
X is greater than *Y*.

?*X* **#<** ?*Y*
X is less than *Y*. In addition to its regular use in problems that require it, this constraint can also be useful to eliminate uninteresting symmetries from a problem. For example, all possible matches between pairs built from four players in total:

```
?- Vs = [A,B,C,D], Vs ins 1..4, all_different(Vs), A #< B, C #< D, A #< C,
   findall(pair(A,B)-pair(C,D), label(Vs), Ms).
Ms = [pair(1, 2)-pair(3, 4), pair(1, 3)-pair(2, 4), pair(1, 4)-pair(2, 3)]
```

+*Q*
The reifiable constraint *Q* does *not* hold. For example, to obtain the complement of a domain:

```
:- use_module(library(clpfd)).

vs_n_num(Vs, N, Num) :-
        maplist(eq_b(N), Vs, Bs),
        sum(Bs, #=, Num).

eq_b(X, Y, B) :- X #= Y #<==> B.
```

Sample queries and their results:

```
?- Vs = [X,Y,Z], Vs ins 0..1, vs_n_num(Vs, 4, Num).
Vs = [X, Y, Z],
Num = 0,
X in 0..1,
Y in 0..1,
Z in 0..1.

?- vs_n_num([X,Y,Z], 2, 3).
X = 2,
Y = 2,
Z = 2.
```

?P **#==>** *?Q*
P implies *Q*.

?P **#<==** *?Q*
Q implies *P*.

?P **#/** *?Q*
P and *Q* hold.

?P **#\\/** *?Q*
P or *Q* holds. For example, the sum of natural numbers below 1000 that are multiples of 3 or 5:

```
?- findall(N, (N mod 3 #= 0 #\/ N mod 5 #= 0, N in 0..999, indomain(N)), Ns),
Ns = [0, 3, 5, 6, 9, 10, 12, 15, 18|...],
Sum = 233168.
```

lex_chain(+*Lists***)**
Lists are lexicographically non-decreasing.

tuples_in(+*Tuples*, +*Relation***)**
Relation must be a list of lists of integers. The elements of the list *Tuples* are constrained to be elements of *Relation*. Arbitrary finite relations, such as compatibility tables, can be modeled in this way. For example, if 1 is compatible with 2 and 5, and 4 is compatible with 0 and 3:

```
?- tuples_in([[X,Y]], [[1,2],[1,5],[4,0],[4,3]]), X = 4.
X = 4,
Y in 0\/3.
```

As another example, consider a train schedule represented as a list of quadruples, denoting departure and arrival places and times for each train. In the following program, Ps is a feasible journey of length 3 from A to D via trains that are part of the given schedule.

```
:- use_module(library(clpfd)).

trains([[1,2,0,1],[2,3,4,5],[2,3,0,1],[3,4,5,6],[3,4,2,3],[3,4,8,9]]).

threepath(A, D, Ps) :-
        Ps = [[A,B,_T0,T1],[B,C,T2,T3],[C,D,T4,_T5]],
        T2 #> T1,
?- Vs = [_,_,_], global_cardinality(Vs, [1-2,3-_]), label(Vs).
Vs = [1, 1, 3] ;
Vs = [1, 3, 1] ;
Vs = [3, 1, 1].
```

global_cardinality(+*Vs, +Pairs, +Options***)**

Vs is a list of finite domain variables, *Pairs* is a list of Key-Num pairs, where Key is an integer and Num is a finite domain variable. The constraint holds iff each V in *Vs* is equal to some key, and for each Key-Num pair in *Pairs*, the number of occurrences of Key in *Vs* is Num. *Options* is a list of options. Supported options are:

consistency(*value***)**

A weaker form of consistency is used.

cost(*Cost, Matrix***)**

Matrix is a list of rows, one for each variable, in the order they occur in *Vs*. Each of these rows is a list of integers, one for each key, in the order these keys occur in *Pairs*. When variable v_i is assigned the value of key k_j, then the associated cost is Matrix_{ij}. Cost is the sum of all costs.

circuit(+*Vs***)**

True if the list *Vs* of finite domain variables induces a Hamiltonian circuit, where the k-th element of *Vs* denotes the successor of node k. Node indexing starts with 1. Examples:

```
?- length(Vs, _), circuit(Vs), label(Vs).
Vs = [] ;
Vs = [1] ;
Vs = [2, 1] ;
Vs = [2, 3, 1] ;
Vs = [3, 1, 2] ;
Vs = [2, 3, 4, 1] .
```

cumulative(+*Tasks***)**

Equivalent to cumulative(*Tasks*, [limit(1)]).

cumulative(+*Tasks, +Options***)**

Tasks is a list of tasks, each of the form task(S_i, D_i, E_i, C_i, T_i). S_i denotes the start time, D_i the positive duration, E_i the end time, C_i the non-negative resource consumption, and T_i the task identifier. Each of these arguments must be a finite domain variable with bounded domain, or an integer. The constraint holds if at any time during the start and end of each task, the total resource consumption of all tasks running at that time does not exceed the global resource limit (which is 1 by default). *Options* is a list of options. Currently, the only supported option is:

limit(*L***)**

The integer L is the global resource limit.

automaton(+*Signature,* +*Nodes,* +*Arcs***)**

Equivalent to automaton(_, _, *Signature*, *Nodes*, *Arcs*, [], [], _), a common use case of `automaton/8`. In the following example, a list of binary finite domain variables is constrained to contain at least two consecutive ones:

```
:- use_module(library(clpfd)).

two_consecutive_ones(Vs) :-
        automaton(Vs, [source(a),sink(c)],
                  [arc(a,0,a), arc(a,1,b),
                   arc(b,0,a), arc(b,1,c),
                   arc(c,0,c), arc(c,1,c)]).

?- length(Vs, 3), two_consecutive_ones(Vs), label(Vs).
Vs = [0, 1, 1] ;
Vs = [1, 1, 0] ;
Vs = [1, 1, 1].
```

automaton(?*Sequence,* ?*Template,* +*Signature,* +*Nodes,* +*Arcs,* +*Counters,* +*Initials,* ?*Finals***)**

True if the finite automaton induced by *Nodes* and *Arcs* (extended with *Counters*) accepts *Signature*. *Sequence* is a list of terms, all of the same shape. Additional constraints must link *Sequence* to *Signature*, if necessary. *Nodes* is a list of source(Node) and sink(Node) terms. *Arcs* is a list of arc(Node,Integer,Node) and arc(Node,Integer,Node,Exprs) terms that denote the automaton's transitions. Each node is represented by an arbitrary term. Transitions that are not mentioned go to an implicit failure node. Exprs is a list of arithmetic expressions, of the same length as *Counters*. In each expression, variables occurring in *Counters* correspond to old counter values, and variables occurring in *Template* correspond to the current element of *Sequence*. When a transition containing expressions is taken, counters are updated as stated. By default, counters remain unchanged. *Counters* is a list of variables that must not occur anywhere outside of the constraint goal. *Initials* is a list of the same length as *Counters*. Counter arithmetic on the transitions relates the counter values in *Initials* to *Finals*.

The following example is taken from Beldiceanu, Carlsson, Debruyne and Petit: "Reformulation of Global Constraints Based on Constraints Checkers", Constraints 10(4), pp 339-362 (2005). It relates a sequence of integers and finite domain variables to its number of inflexions, which are switches between strictly ascending and strictly descending subsequences:

```
:- use_module(library(clpfd)).

sequence_inflexions(Vs, N) :-
        variables_signature(Vs, Sigs),
        automaton(_, _, Sigs,
                  [source(s),sink(i),sink(j),sink(s)],
                  [arc(s,0,s), arc(s,1,j), arc(s,2,i),
                   arc(i,0,i), arc(i,1,j,[C+1]), arc(i,2,i),
                   arc(j,0,j), arc(j,1,j), arc(j,2,i,[C+1])], [C], [0], [N]).
```

```
variables_signature([], []).
variables_signature([V|Vs], Sigs) :-
        variables_signature_(Vs, V, Sigs).

variables_signature_([], _, []).
variables_signature_([V|Vs], Prev, [S|Sigs]) :-
        V #= Prev #<==> S #= 0,
        Prev #< V #<==> S #= 1,
        Prev #> V #<==> S #= 2,
        variables_signature_(Vs, V, Sigs).
```

Example queries:

```
?- sequence_inflexions([1,2,3,3,2,1,3,0], N).
N = 3.

?- length(Ls, 5), Ls ins 0..1, sequence_inflexions(Ls, 3), label(Ls).
Ls = [0, 1, 0, 1, 0] ;
Ls = [1, 0, 1, 0, 1].
```

transpose(+*Matrix*, ?*Transpose***)**

Transpose a list of lists of the same length. Example:

```
?- transpose([[1,2,3],[4,5,6],[7,8,9]], Ts).
Ts = [[1, 4, 7], [2, 5, 8], [3, 6, 9]].
```

This predicate is useful in many constraint programs. Consider for instance Sudoku:

```
:- use_module(library(clpfd)).

sudoku(Rows) :-
        length(Rows, 9), maplist(length_(9), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns), maplist(all_distinct, Columns),
        Rows = [A,B,C,D,E,F,G,H,I],
        blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).

length_(L, Ls) :- length(Ls, L).

blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
        all_distinct([A,B,C,D,E,F,G,H,I]),
        blocks(Bs1, Bs2, Bs3).

problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```

Sample query:

```
?- problem(1, Rows), sudoku(Rows), maplist(writeln, Rows).
[9, 8, 7, 6, 5, 4, 3, 2, 1]
[2, 4, 6, 1, 7, 3, 9, 8, 5]
[3, 5, 1, 9, 2, 8, 7, 4, 6]
[1, 2, 8, 5, 3, 7, 6, 9, 4]
[6, 3, 4, 8, 9, 2, 1, 5, 7]
[7, 9, 5, 4, 6, 1, 8, 3, 2]
[5, 1, 9, 2, 8, 6, 4, 7, 3]
[4, 7, 2, 3, 1, 9, 5, 6, 8]
[8, 6, 3, 7, 4, 5, 2, 1, 9]
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], ... , [...|...]].
```

zcompare(*?Order, ?A, ?B***)**

Analogous to `compare/3`, with finite domain variables *A* and *B*. Example:

```
:- use_module(library(clpfd)).

 n_factorial(N, F) :-
         zcompare(C, N, 0),
         n_factorial_(C, N, F).

 n_factorial_(=, _, 1).
 n_factorial_(>, N, F) :- F #= F0*N, N1 #= N - 1, n_factorial(N1, F0).
```

This version is deterministic if the first argument is instantiated:

```
?- n_factorial(30, F).
F = 265252859812191058636308480000000.
```

chain(*+Zs, +Relation***)**

Zs is a list of finite domain variables that are a chain with respect to the partial order *Relation*, in the order they appear in the list. *Relation* must be #=, #=<, #>=, #< or #>. For example:

```
?- chain([X,Y,Z], #>=).
X#>=Y,
Y#>=Z.
```

fd_var(+*Var***)**
True iff *Var* is a CLP(FD) variable.

fd_inf(+*Var, -Inf***)**
Inf is the infimum of the current domain of *Var*.

fd_sup(+*Var, -Sup***)**
Sup is the supremum of the current domain of *Var*.

fd_size(+*Var, -Size***)**
Size is the number of elements of the current domain of *Var*, or the atom **sup** if the domain is unbounded.

fd_dom(+*Var, -Dom***)**
Dom is the current domain (see `in/2`) of *Var*. This predicate is useful if you want to reason about domains. It is not needed if you only want to display remaining domains; instead, separate your model from the search part and let the toplevel display this information via residual goals.

A.8 `clpqr`: Constraint Logic Programming over Rationals and Reals

Author: *Christian Holzbaur*, ported to SWI-Prolog by *Leslie De Koninck*, K.U. Leuven

This CLP(Q,R) system is a port of the CLP(Q,R) system of Sicstus Prolog by Christian Holzbaur: Holzbaur C.: OFAI clp(q,r) Manual, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.[1] This manual is roughly based on the manual of the above mentioned CLP(Q,R) implementation.

The CLP(Q,R) system consists of two components: the CLP(Q) library for handling constraints over the rational numbers and the CLP(R) library for handling constraints over the real numbers (using floating point numbers as representation). Both libraries offer the same predicates (with exception of `bb_inf/4` in CLP(Q) and `bb_inf/5` in CLP(R)). It is allowed to use both libraries in one program, but using both CLP(Q) and CLP(R) constraints on the same variable will result in an exception.

Please note that the `clpqr` library is *not* an *autoload* library and therefore this library must be loaded explicitly before using it:

```
:- use_module(library(clpq)).
```

or

```
:- use_module(library(clpr)).
```

---

[1] http://www.ai.univie.ac.at/cgi-bin/tr-online?number+95-09

A.8.1 Solver predicates

The following predicates are provided to work with constraints:

{}(+*Constraints***)**

Adds the constraints given by *Constraints* to the constraint store.

entailed(+*Constraint***)**

Succeeds if *Constraint* is necessarily true within the current constraint store. This means that adding the negation of the constraint to the store results in failure.

inf(+*Expression, -Inf***)**

Computes the infimum of *Expression* within the current state of the constraint store and returns that infimum in *Inf*. This predicate does not change the constraint store.

sup(+*Expression, -Sup***)**

Computes the supremum of *Expression* within the current state of the constraint store and returns that supremum in *Sup*. This predicate does not change the constraint store.

minimize(+*Expression***)**

Minimizes *Expression* within the current constraint store. This is the same as computing the infimum and equating the expression to that infimum.

maximize(+*Expression***)**

Maximizes *Expression* within the current constraint store. This is the same as computing the supremum and equating the expression to that supremum.

bb_inf(+*Ints, +Expression, -Inf, -Vertex, +Eps***)**

This predicate is offered in CLP(R) only. It computes the infimum of *Expression* within the current constraint store, with the additional constraint that in that infimum, all variables in *Ints* have integral values. *Vertex* will contain the values of *Ints* in the infimum. *Eps* denotes how much a value may differ from an integer to be considered an integer. E.g. when *Eps* = 0.001, then X = 4.999 will be considered as an integer (5 in this case). *Eps* should be between 0 and 0.5.

bb_inf(+*Ints, +Expression, -Inf, -Vertex***)**

This predicate is offered in CLP(Q) only. It behaves the same as `bb_inf/5` but does not use an error margin.

bb_inf(+*Ints, +Expression, -Inf***)**

The same as `bb_inf/5` or `bb_inf/4` but without returning the values of the integers. In CLP(R), an error margin of 0.001 is used.

dump(+*Target, +Newvars, -CodedAnswer***)**

Returns the constraints on *Target* in the list *CodedAnswer* where all variables of *Target* have been replaced by *NewVars*. This operation does not change the constraint store. E.g. in

```
dump([X,Y,Z],[x,y,z],Cons)
```

`Cons` will contain the constraints on X, Y and Z, where these variables have been replaced by atoms x, y and z.

⟨*Constraints*⟩	::=	⟨*Constraint*⟩	single constraint
	|	⟨*Constraint*⟩ , ⟨*Constraints*⟩	conjunction
	|	⟨*Constraint*⟩ ; ⟨*Constraints*⟩	disjunction
⟨*Constraint*⟩	::=	⟨*Expression*⟩ < ⟨*Expression*⟩	less than
	|	⟨*Expression*⟩ > ⟨*Expression*⟩	greater than
	|	⟨*Expression*⟩ =< ⟨*Expression*⟩	less or equal
	|	<=(⟨*Expression*⟩, ⟨*Expression*⟩)	less or equal
	|	⟨*Expression*⟩ >= ⟨*Expression*⟩	greater or equal
	|	⟨*Expression*⟩ =\= ⟨*Expression*⟩	not equal
	|	⟨*Expression*⟩ =:= ⟨*Expression*⟩	equal
	|	⟨*Expression*⟩ = ⟨*Expression*⟩	equal
⟨*Expression*⟩	::=	⟨*Variable*⟩	Prolog variable
	|	⟨*Number*⟩	Prolog number (float, integer)
	|	+⟨*Expression*⟩	unary plus
	|	-⟨*Expression*⟩	unary minus
	|	⟨*Expression*⟩ + ⟨*Expression*⟩	addition
	|	⟨*Expression*⟩ - ⟨*Expression*⟩	substraction
	|	⟨*Expression*⟩ * ⟨*Expression*⟩	multiplication
	|	⟨*Expression*⟩ / ⟨*Expression*⟩	division
	|	abs(⟨*Expression*⟩)	absolute value
	|	sin(⟨*Expression*⟩)	sine
	|	cos(⟨*Expression*⟩)	cosine
	|	tan(⟨*Expression*⟩)	tangent
	|	exp(⟨*Expression*⟩)	exponent
	|	pow(⟨*Expression*⟩)	exponent
	|	⟨*Expression*⟩ ^ ⟨*Expression*⟩	exponent
	|	min(⟨*Expression*⟩, ⟨*Expression*⟩)	minimum
	|	max(⟨*Expression*⟩, ⟨*Expression*⟩)	maximum

Table A.1: CLP(Q,R) constraint BNF

A.8.2 Syntax of the predicate arguments

The arguments of the predicates defined in the subsection above are defined in table A.1. Failing to meet the syntax rules will result in an exception.

A.8.3 Use of unification

Instead of using the `{}/1` predicate, you can also use the standard unification mechanism to store constraints. The following code samples are equivalent:

- *Unification with a variable*

```
{X = Y}
X = Y
```

$A = B * C$	B or C is ground	A = 5 * C or A = B * 4
	A and (B or C) are ground	20 = 5 * C or 20 = B * 4
$A = B=C$	C is ground	A = B / 3
	A and B are ground	4 = 12 / C
$X = min(Y;Z)$	Y and Z are ground	X = min(4,3)
$X = max(Y;Z)$	Y and Z are ground	X = max(4,3)
$X = abs(Y)$	Y is ground	X = abs(-7)
$X = pow(Y;Z)$	X and Y are ground	8 = 2 ^ Z
$X = exp(Y;Z)$	X and Z are ground	8 = Y ^ 3
$X = Y \hat{} Z$	Y and Z are ground	X = 2 ^ 3
$X = sin(Y)$	X is ground	1 = sin(Y)
$X = cos(Y)$	Y is ground	X = sin(1.5707)
$X = tan(Y)$		

Table A.2: CLP(Q,R) isolating axioms

- *Unification with a number*

```
{X =:= 5.0}
{X = 5.0}
X = 5.0
```

A.8.4 Non-linear constraints

The CLP(Q,R) system deals only passively with non-linear constraints. They remain in a passive state until certain conditions are satisfied. These conditions, which are called the isolation axioms, are given in table A.2.

A.8.5 Status and known problems

The clpq and clpr libraries are ‘orphaned’, i.e., they currently have no maintainer.

- *Top-level output*
  The top-level output may contain variables not present in the original query:

```
?- {X+Y>=1}.
{Y=1-X+_G2160, _G2160>=0}.

?-
```

  Nonetheless, for linear constraints this kind of answer means unconditional satisfiability.

- *Dumping constraints*
  The first argument of `dump/3` has to be a list of free variables at call-time:

```
?- {X=1},dump([X],[Y],L).
ERROR: Unhandled exception: Unknown message:
        instantiation_error(dump([1],[_G11],_G6),1)
?-
```

A.9 library(csv): Process CSV (Comma-Separated Values) data

See also RFC 4180

To be done

- Implement immediate assert of the data to avoid possible stack overflows.
- Writing creates an intermediate code-list, possibly overflowing resources. This waits for pure output!

This library parses and generates CSV data. CSV data is represented in Prolog as a list of rows. Each row is a compound term, where all rows have the same name and arity.

csv_read_file(+*File*, -*Rows***)** *[det]*
csv_read_file(+*File*, -*Rows*, +*Options***)** *[det]*

Read a CSV file into a list of rows. Each row is a Prolog term with the same arity. *Options* is handed to `csv//2`. Remaining options are processed by `phrase_from_file/3`. The default separator depends on the file name extension and is `\t` for `.tsv` files and `,` otherwise.

Suppose we want to create a predicate `table/6` from a CSV file that we know contains 6 fields per record. This can be done using the code below. Without the option arity(6), this would generate a predicate table/N, where N is the number of fields per record in the data.

```
?- csv_read_file(File, Rows, [functor(table), arity(6)]),
   maplist(assert, Rows).
```

csv(?*Rows***) //** *[det]*
csv(?*Rows*, +*Options***) //** *[det]*

Prolog DCG to 'read/write' CSV data. *Options*:

separator(+*Code***)**
The comma-separator. Must be a character code. Default is (of course) the comma. Character codes can be specified using the 0' notion. E.g., `separator(0';)`.

strip(+*Boolean***)**
If `true` (default `false`), strip leading and trailing blank-space. RFC4180 says that blank space is part of the data.

convert(+*Boolean***)**
if `true` (Default), use `name/2` on the field-data. This translates the field into a number if possible.

functor(+*Atom***)**
Functor to use for creating row-terms. Default is `row`.

arity(?*Arity***)**
Number of fields in each row. This predicate raises a domain_error(row_arity(Expected), Found) if a row is found with different arity.

match_arity(+*Boolean***)**
If `false` (default `true`), do not reject CSV files where lines provide a varying number of fields (columns). This can be a work-around to use some incorrect CSV files.

csv_read_file_row(+*File, -Row, +Options***)** *[nondet]*
True when *Row* is a row in *File*. First unifies *Row* with the first row in *File*. Backtracking yields the second, ... row. This interface is an alternative to `csv_read_file/3` that avoids loading all rows in memory. Note that this interface does not guarantee that all rows in *File* have the same arity.

In addition to the options of `csv_read_file/3`, this predicate processes the option:

line(-*Line***)**
Line is unified with the 1-based line-number from which *Row* is read.

csv_write_file(+*File, +Data***)** *[det]*
csv_write_file(+*File, +Data, +Options***)** *[det]*
Write a list of Prolog terms to a CSV file. *Options* are given to `csv//2`. Remaining options are given to `open/4`. The default separator depends on the file name extension and is `\t` for `.tsv` files and `,` otherwise.

csv_write_stream(+*Stream, +Data, +Options***)** *[det]*
Write the rows in *Data* to *Stream*. This is similar to `csv_write_file/3`, but can deal with data that is produced incrementally. The example below saves all answers from the predicate `data/3` to File.

```
save_data(File) :-
   setup_call_cleanup(
       open(File, write, Out),
       forall(data(C1,C2,C3),
              csv_write_file(Out, [row(C1,C2,C3)], [])),
       close(Out)),
```

A.10 library(debug): Print debug messages and test assertions

author Jan Wielemaker

This library is a replacement for `format/3` for printing debug messages. Messages are assigned a *topic*. By dynamically enabling or disabling topics the user can select desired messages. Debug statements are removed when the code is compiled for optimization.

See manual for details. With XPCE, you can use the call below to start a graphical monitorring tool.

```
?- prolog_ide(debug_monitor).
```

Using the predicate `assertion/1` you can make assumptions about your program explicit, trapping the debugger if the condition does not hold.

debugging(+*Topic***)** *[semidet]*
debugging(-*Topic***)** *[nondet]*
debugging(*?Topic, ?Bool***)** *[nondet]*

Check whether we are debugging *Topic* or enumerate the topics we are debugging.

debug(+*Topic***)** *[det]*
nodebug(+*Topic***)** *[det]*

Add/remove a topic from being printed. nodebug(_) removes all topics. Gives a warning if the topic is not defined unless it is used from a directive. The latter allows placing debug topics at the start a a (load-)file without warnings.

For `debug/1`, *Topic* can be a term *Topic* > Out, where Out is either a stream or stream-alias or a filename (atom). This redirects debug information on this topic to the given output.

list_debug_topics *[det]*

List currently known debug topics and their setting.

debug_message_context(+*What***)** *[det]*

Specify additional context for debug messages. *What* is one of +Context or -Context and Context is one of `thread`. `time` or time(Format), where Format is a format-specification for `format_time/3` (default is `%T.%3f`). Initially, `debug/3` show only thread information.

debug(+*Topic*, +*Format*, :*Args***)** *[det]*

Similar to `format/3` to `user_error`, but only prints if *Topic* is activated through `debug/1`. *Args* is a meta-argument to deal with goal for the @-command. Output is first handed to the hook `prolog:debug_print_hook/3`. If this fails, *Format*+*Args* is translated to text using the message-translation (see `print_message/2`) for the term debug(*Format*, *Args*) and then printed to every matching destination (controlled by `debug/1`) using `print_message_lines/3`.

The message is preceeded by '% ' and terminated with a newline.

See also `format/3`.

prolog:debug_print_hook(+*Topic*, +*Format*, +*Args***)** *[semidet,multifile]*

Hook called by `debug/3`. This hook is used by the graphical frontend that can be activated using `prolog_ide/1`:

```
?- prolog_ide(debug_monitor).
```

assertion(:*Goal***)** *[det]*

Acts similar to C assert() macro. It has no effect if *Goal* succeeds. If *Goal* fails or throws and exception, the following steps are taken:

- call `prolog:assertion_failed/2`. If `prolog:assertion_failed/2` fails, then:
  - If this is an interactive toplevel thread, print a message, the stack-trace and finally traps the debugger.
  - Otherwise, throw error(assertion_error(Reason, G),_) where Reason is one of `fail` or the exception raised.

prolog:assertion_failed(+*Reason*, +*Goal***)** *[semidet,multifile]*

This hook is called if the *Goal* of `assertion/1` fails. *Reason* is unified with either `fail` if *Goal* simply failed or an exception ball otherwise. If this hook fails, the default behaviour is activated. If the hooks throws an exception it will be propagated into the caller of `assertion/1`.

A.11 gensym: Generate unique identifiers

Gensym (**Gen**erate **Sym**bols) is an old library for generating unique symbols (atoms). Such symbols are generated from a base atom which gets a sequence number appended. Of course there is no guarantee that 'catch22' is not an already defined atom and therefore one must be aware these atoms are only unique in an isolated context.

The SWI-Prolog gensym library is thread-safe. The sequence numbers are global over all threads and therefore generated atoms are unique over all threads.

gensym(+*Base*, -*Unique***)**

Generate a unique atom from base *Base* and unify it with *Unique*. *Base* should be an atom. The first call will return ⟨*base*⟩1, the next ⟨*base*⟩2, etc. Note that this is no guarantee that the atom is unique in the system.

reset_gensym(+*Base***)**

Restart generation of identifiers from *Base* at ⟨*Base*⟩1. Used to make sure a program produces the same results on subsequent runs. Use with care.

reset_gensym

Reset gensym for all registered keys. This predicate is available for compatibility only. New code is strongly advised to avoid the use of reset_gensym or at least to reset only the keys used by your program to avoid unexpected side effects on other components.

A.12 library(lists): List Manipulation

Compatibility Virtually every Prolog system has library(lists), but the set of provided predicates is diverse. There is a fair agreement on the semantics of most of these predicates, although error handling may vary.

This library provides commonly accepted basic predicates for list manipulation in the Prolog community. Some additional list manipulations are built-in. See e.g., `memberchk/2`, `length/2`.

The implementation of this library is copied from many places. These include: "The Craft of Prolog", the DEC-10 Prolog library (LISTRO.PL) and the YAP lists library. Some predicates are reimplemented based on their specification by Quintus and SICStus.

member(?*Elem*, ?*List***)**

True if *Elem* is a member of *List*. The SWI-Prolog definition differs from the classical one. Our definition avoids unpacking each list element twice and provides determinism on the last element. E.g. this is deterministic:

```
    member(X, [One]).
```

author Gertjan van Noord

append(*?List1, ?List2, ?List1AndList2***)**

List1AndList2 is the concatination of *List1* and *List2*

append(*+ListOfLists, ?List***)**

Concatenate a list of lists. Is true if Lists is a list of lists, and *List* is the concatenation of these lists.

Parameters	
ListOfLists	must be a list of -possibly- partial lists

prefix(*?Part, ?Whole***)**

True iff *Part* is a leading substring of *Whole*. This is the same as append(*Part*, _, *Whole*).

select(*?Elem, ?List1, ?List2***)**

Is true when *List1*, with *Elem* removed results in *List2*.

selectchk(*+Elem, +List, -Rest***)** *[semidet]*

Semi-deterministic removal of first element in *List* that unifies *Elem*.

select(*?X, ?XList, ?Y, ?YList***)** *[nondet]*

Is true when select(*X*, *XList*) and select(*Y*, *YList*) are true, *X* and *Y* appear in the same locations of their respective lists and same_length(*XList*, *YList*) is true. A typical use for this predicate is to *replace* an element:

```
?- select(b, [a,b,c], 2, X).
X = [a, 2, c] ;
X = [a, b, c].
```

selectchk(*?X, ?XList, ?Y, ?YList***)** *[semidet]*

Semi-deterministic version of `select/4`.

nextto(*?X, ?Y, ?List***)**

True of *Y* follows *X* in *List*.

delete(*?List1, ?Elem, ?List2***)** *[det]*

Is true when Lis1, with all occurences of *Elem* deleted results in *List2*.

See also `select/3`, `subtract/3`.

deprecated There are too many ways in which one might want to delete elements from a list to justify the name. Think of matching (= vs. ==), delete first/all, be deterministic or not.

nth0(*?Index, ?List, ?Elem***)**

True when *Elem* is the *Index*-th element of *List*. Counting starts at 0.

Errors type_error(integer, *Index*) if *Index* is not an integer or unbound.

See also `nth1/3`.

nth1(*?Index, ?List, ?Elem***)**
Is true when *Elem* is the *Index*'th element of *List*. Counting starts at 1.

See also `nth0/3`.

nth0(*?N, ?List, ?Elem, ?Rest***)** *[det]*
Select/insert element at index. True when *Elem* is the *N*-th (0-based) element of *List* and *Rest* is the remainder (as in by `select/3`) of *List*. For example:

```
?- nth0(I, [a,b,c], E, R).
I = 0, E = a, R = [b, c] ;
I = 1, E = b, R = [a, c] ;
I = 2, E = c, R = [a, b] ;
false.
```

```
?- nth0(1, L, a1, [a,b]).
L = [a, a1, b].
```

nth1(*?N, ?List, ?Elem, ?Rest***)** *[det]*
As `nth0/4`, but counting starts at 1.

last(*?List, ?Last***)**
Succeeds when '*Last*' is the last element of '*List*'. This predicate is `semidet` if *List* is a list and `multi` if *List* is a partial list.

Compatibility There is no de-facto standard for the argument order of `last/2`. Be careful when porting code or use append(_, [*Last*], *List*) as a portable alternative.

proper_length(*@List, -Length***)** *[semidet]*
True when *Length* is the number of elements in the proper list *List*. This is equivalent to

```
proper_length(List, Length) :-
      is_list(List),
      length(List, Length).
```

same_length(*?List1, ?List2***)**
Is true when *List1* and *List2* are lists with the same number of elements. The predicate is deterministic if at least one of the arguments is a proper list. It is non-deterministic if both arguments are partial lists.

See also `length/2`

reverse(*?List1, ?List2***)**
Is true when the elements of *List2* are in reverse order compared to *List1*.

permutation(*?Xs, ?Ys***)** *[nondet]*
permutation(*Xs*, *Ys*) is true when *Xs* is a permutation of *Ys*. This can solve for *Ys* given *Xs* or *Xs* given *Ys*, or even enumerate *Xs* and *Ys* together. The predicate `permutation/2` is

primarily intended to generate permutations. Note that a list of length N has N! permutations and unbounded permutation generation becomes prohibitively expensive, even for rather short lists (10! = 3,628,800).

If both *Xs* and *Ys* are provided and both lists have equal length the order is `|`*Xs*`|`^2. Simply testing whether *Xs* is a permutation of *Ys* can be achieved in order log(`|`*Xs*`|`) using `msort/2` as illustrated below with the `semidet` predicate `is_permutation/2`:

```
is_permutation(Xs, Ys) :-
  msort(Xs, Sorted),
  msort(Ys, Sorted).
```

The example below illustrate that *Xs* and *Ys* being proper lists is not a sufficient condition to use the above replacement.

```
?- permutation([1,2], [X,Y]).
X = 1, Y = 2 ;
X = 2, Y = 1 ;
false.
```

Errors type_error(list, Arg) if either argument is not a proper or partial list.

flatten(+*List1*, ?*List2***)** *[det]*

Is true it *List2* is a non nested version of *List1*.

See also `append/2`

deprecated Ending up needing `flatten/3` often indicates, like `append/3` for appending two lists, a bad design. Efficient code that generates lists from generated small lists must use difference lists, often possible through grammar rules for optimal readability.

max_member(-*Max*, +*List***)** *[semidet]*

True when *Max* is the largest member in the standard order of terms. Fails if *List* is empty.

See also

- `compare/3`
- `max_list/2` for the maximum of a list of numbers.

min_member(-*Min*, +*List***)** *[semidet]*

True when *Min* is the smallest member in the standard order of terms. Fails if *List* is empty.

See also

- `compare/3`
- `min_list/2` for the minimum of a list of numbers.

sumlist(+*List*, -*Sum***)** *[det]*

Sum is the result of adding all numbers in *List*.

max_list(+*List:list(number)*, -*Max:number***)** *[semidet]*

True if *Max* is the largest number in *List*. Fails if *List* is empty.

See also `max_member/2`.

min_list(+*List:list(number), -Min:number***)** *[semidet]*

True if *Min* is the largest number in *List*. Fails if *List* is empty.

See also `min_member/2`.

numlist(+*Low, +High, -List***)** *[semidet]*

List is a list [*Low*, *Low*+1, ... *High*]. Fails if *High* < *Low*.

Errors

- type_error(integer, *Low*)
- type_error(integer, *High*)

is_set(@*Set***)** *[det]*

True if *Set* is a proper list without duplicates. Equivalence is based on `==/2`. The implementation uses `sort/2`, which implies that the complexity is N*log(N) and the predicate may cause a resource-error. There are no other error conditions.

list_to_set(+*List, ?Set***)** *[det]*

True when *Set* has the same element as *List* in the same order. The left-most copy of the duplicate is retained. The complexity of this operation is |*List*|^2.

See also `sort/2`.

intersection(+*Set1, +Set2, -Set3***)** *[det]*

True if *Set3* unifies with the intersection of *Set1* and *Set2*. The complexity of this predicate is |*Set1*|*|*Set2*|

See also `ord_intersection/3`.

union(+*Set1, +Set2, -Set3***)** *[det]*

True if *Set3* unifies with the union of *Set1* and *Set2*. The complexity of this predicate is |*Set1*|*|*Set2*|

See also `ord_union/3`.

subset(+*SubSet, +Set***)** *[semidet]*

True if all elements of *SubSet* belong to *Set* as well. Membership test is based on `memberchk/2`. The complexity is |*SubSet*|*|*Set*|.

See also `ord_subset/2`.

subtract(+*Set, +Delete, -Result***)** *[det]*

Delete all elements from '*Set*' that occur in '*Delete*' (a set) and unify the result with '*Result*'. Deletion is based on unification using `memberchk/2`. The complexity is |*Delete*|*|*Set*|.

See also `ord_subtract/3`.

A.13 `nb_set`: Non-backtrackable set

The library `nb_set` defines *non-backtrackable sets*, implemented as binary trees. The sets are represented as compound terms and manipulated using `nb_setarg/3`. Non-backtrackable manipulation of datastructures is not supported by a large number of Prolog implementations, but it has several advantages over using the database. It produces less garbage, is thread-safe, reentrant and deals with exceptions without leaking data.

Similar to the `assoc` library, keys can be any Prolog term, but it is not allowed to instantiate or modify a term.

One of the ways to use this library is to generate unique values on backtracking *without* generating *all* solutions first, for example to act as a filter between a generator producing many duplicates and an expensive test routine, as outlined below:

```
generate_and_test(Solution) :-
        empty_nb_set(Set),
        generate(Solution),
        add_nb_set(Solution, Set, true),
        test(Solution).
```

empty_nb_set(*?Set***)**

True if *Set* is a non-backtrackable empty set.

add_nb_set(*+Key, !Set***)**

Add *Key* to *Set*. If *Key* is already a member of *Set*, `add_nb_set/3` succeeds without modifying *Set*.

add_nb_set(*+Key, !Set, ?New***)**

If *Key* is not in *Set* and *New* is unified to `true`, *Key* is added to *Set*. If *Key* is in *Set*, *New* is unified to `false`. It can be used for many purposes:

`add_nb_set(+, +, false)`	Test membership
`add_nb_set(+, +, true)`	Succeed only if new member
`add_nb_set(+, +, Var)`	Succeed, binding *Var*

gen_nb_set(*+Set, -Key***)**

Generate all members of *Set* on backtracking in the standard order of terms. To test membership, use `add_nb_set/3`.

size_nb_set(*+Set, -Size***)**

Unify *Size* with the number of elements in *Set*.

nb_set_to_list(*+Set, -List***)**

Unify *List* with a list of all elements in *Set* in the standard order of terms (i.e., an *ordered list*).

A.14 `www_browser`: Activating your Web-browser

This library deals with the very system-dependent task of opening a web page in a browser. See also `url` and the HTTP package.

www_open_url(+*URL*)

Open *URL* in an external web browser. The reason to place this in the library is to centralise the maintenance on this highly platform- and browser-specific task. It distinguishes between the following cases:

- *MS-Windows*
  If it detects MS-Windows it uses `win_shell/2` to open the *URL*. The behaviour and browser started depends on the version of Windows and Windows-shell configuration, but in general it should be the behaviour expected by the user.
- *Other platforms*
  On other platforms it tests the environment variable (see `getenv/2`) named `BROWSER` or uses `netscape` if this variable is not set. If the browser is either `mozilla` or `netscape`, `www_open_url/1` first tries to open a new window on a running browser using the `-remote` option of Netscape. If this fails or the browser is not `mozilla` or `netscape` the system simply passes the URL as first argument to the program.

A.15 library(option): Option list processing

See also

- library(record)
- Option processing capabilities may be declared using the directive `predicate_options/3`.

To be done We should consider putting many options in an assoc or record with appropriate preprocessing to achieve better performance.

The library(option) provides some utilities for processing option lists. Option lists are commonly used as an alternative for many arguments. Examples built-in predicates are `open/4` and `write_term/3`. Naming the arguments results in more readable code and the list nature makes it easy to extend the list of options accepted by a predicate. Option lists come in two styles, both of which are handled by this library.

Name(Value) This is the preferred style.

Name = Value This is often used, but deprecated.

Processing options inside time critical code (loops) can cause serious overhead. One possibility is to define a record using library(record) and initialise this using make_<record>`/2`. In addition to providing good performance, this also provides type-checking and central declaration of defaults.

```
:- record atts(width:integer=100, shape:oneof([box,circle])=box).

process(Data, Options) :-
        make_atts(Options, Attributes),
        action(Data, Attributes).
action(Data, Attributes) :-
        atts_shape(Attributes, Shape),
        ...
```

Options typically have exactly one argument. The library does support options with 0 or more than one arguments with the following restrictions:

- The predicate `option/3` and `select_option/4`, involving default are meaningless. They perform an arg(1, Option, Default), causing failure without arguments and filling only the first option-argument otherwise.
- `meta_options/3` can only qualify options with exactly one argument.

option(*?Option, +OptionList, +Default***)** *[semidet]*

Get an option from a *OptionList*. *OptionList* can use the Name=Value as well as the Name(Value) convention.

	Parameters
Option	Term of the form Name(?Value).

option(*?Option, +OptionList***)** *[semidet]*

Get an option from a *OptionList*. *OptionList* can use the Name=Value as well as the Name(Value) convention. Fails silently if the option does not appear in *OptionList*.

	Parameters
Option	Term of the form Name(?Value).

select_option(*?Option, +Options, -RestOptions***)** *[semidet]*

Get and remove option from an option list. As `option/2`, removing the matching option from *Options* and unifying the remaining options with *RestOptions*.

select_option(*?Option, +Options, -RestOptions, +Default***)** *[det]*

Get and remove option with default value. As `select_option/3`, but if *Option* is not in *Options*, its value is unified with *Default* and *RestOptions* with *Options*.

merge_options(*+New, +Old, -Merged***)** *[det]*

Merge two option lists. *Merged* is a sorted list of options using the canonical format Name(Value) holding all options from *New* and *Old*, after removing conflicting options from *Old*.

Multi-values options (e.g., proxy(Host, Port)) are allowed, where both option-name and arity define the identity of the option.

meta_options(*+IsMeta, :Options0, -Options***)** *[det]*

Perform meta-expansion on options that are module-sensitive. Whether an option name is module sensitive is determined by calling call(*IsMeta*, Name). Here is an example:

```
        meta_options(is_meta, OptionsIn, Options),
        ...

is_meta(callback).
```

Meta-options must have exactly one argument. This argument will be qualified.

To be done Should be integrated with declarations from `predicate_options/3`.

A.16 library(optparse): command line parsing

author Marcus Uneson
version 0.20 (2011-04-27)
To be done : validation? e.g, numbers; file path existence; one-out-of-a-set-of-atoms

This module helps in building a command-line interface to an application. In particular, it provides functions that take an option specification and a list of atoms, probably given to the program on the command line, and return a parsed representation (a list of the customary Key(Val) by default; or optionally, a list of Func(Key, Val) terms in the style of `current_prolog_flag/2`). It can also synthesize a simple help text from the options specification.

The terminology in the following is partly borrowed from python, see `http://docs.python.org/library/optparse.html#terminology` . Very briefly, *arguments* is what you provide on the command line and for many prologs show up as a list of atoms `Args` in `current_prolog_flag(argv, Args)`. For a typical prolog incantation, they can be divided into

- *runtime arguments*, which controls the prolog runtime; conventionally, they are ended by '–';
- *options*, which are key-value pairs (with a boolean value possibly implicit) intended to control your program in one way or another; and
- *positional arguments*, which is what remains after all runtime arguments and options have been removed (with implicit arguments – true/false for booleans – filled in).

Positional arguments are in particular used for mandatory arguments without which your program won't work and for which there are no sensible defaults (e.g,, input file names). Options, by contrast, offer flexibility by letting you change a default setting. Options are optional not only by etymology: this library has no notion of mandatory or required options (see the python docs for other rationales than laziness).

The command-line arguments enter your program as a list of atoms, but the programs perhaps expects booleans, integers, floats or even prolog terms. You tell the parser so by providing an *options specification*. This is just a list of individual option specifications. One of those, in turn, is a list of ground prolog terms in the customary Name(Value) format. The following terms are recognized (any others raise error).

opt(*Key*)
Key is what the option later will be accessed by, just like for current_prolog_flag(Key, Value). This term is mandatory (an error is thrown if missing).

shortflags(*ListOfFlags*)

ListOfFlags denotes any single-dashed, single letter args specifying the current option (`-s` , `-K`, etc). Uppercase letters must be quoted. Usually ListOfFlags will be a singleton list, but sometimes aliased flags may be convenient.

longflags(*ListOfFlags*)

ListOfFlags denotes any double-dashed arguments specifying the current option (`--verbose`, `--no-debug`, etc). They are basically a more readable alternative to short flags, except

1. long flags can be specified as `--flag value` or `--flag=value` (but not as `--flagvalue`); short flags as `-f val` or `-fval` (but not `-f=val`)
2. boolean long flags can be specified as `--bool-flag` or `--bool-flag=true` or `--bool-flag true`; and they can be negated as `--no-bool-flag` or `--bool-flag=false` or `--bool-flag false`.

Except that shortflags must be single characters, the distinction between long and short is in calling convention, not in namespaces. Thus, if you have shortflags([v]), you can use it as `-v2` or `-v 2` or `--v=2` or `--v 2` (but not `-v=2` or `--v2`).

Shortflags and longflags both default to `[]`. It can be useful to have flagless options – see example below.

meta(*Meta*)

Meta is optional and only relevant for the synthesized usage message and is the name (an atom) of the metasyntactic variable (possibly) appearing in it together with type and default value (e.g, `x:integer=3`, `interest:float=0.11`). It may be useful to have named variables (`x`, `interest`) in case you wish to mention them again in the help text. If not given the `Meta:` part is suppressed – see example below.

type(*Type*)

Type is one of `boolean, atom, integer, float, term`. The corresponding argument will be parsed appropriately. This term is optional; if not given, defaults to `term`.

default(*Default*)

Default value. This term is optional; if not given, or if given the special value ’

```
--mode                  -m     atom=SCAN       data gathering mode, one of
                                                   SCAN: do this
                                                   READ: do that
                                                   MAKE: fabricate some numbers
                                                   WAIT: don’t do anything
--rebuild-cache         -r     boolean=true    rebuild cache in each iteration
--heisenberg-threshold  -t,-h  float=0.1       heisenberg threshold
--depths, --iters       -i,-d  K:integer=3     stop after K iterations
--distances                    term=[1,2,3,5]  initial prolog term
--output-file           -o     FILE:atom=_     write output to FILE
--label                 -l     atom=REPORT     report label
--verbosity             -v     V:integer=2     verbosity level, 1 <= V <= 3
```

We may also have some configuration parameters which we currently think not needs to be controlled from the command line, say path(’/some/file/path’).

This interface is described by the following options specification (order between the specifications of a particular option is irrelevant).

```
ExampleOptsSpec =
    [ [opt(mode    ), type(atom), default(’SCAN’),
        shortflags([m]),   longflags([’mode’] ),
        help([ ’data gathering mode, one of’
             , ’   SCAN: do this’
             , ’   READ: do that’
             , ’   MAKE: fabricate some numbers’
             , ’   WAIT: don’’t do anything’])]

    , [opt(cache), type(boolean), default(true),
        shortflags([r]),   longflags([’rebuild-cache’]),
        help(’rebuild cache in each iteration’)]

    , [opt(threshold), type(float), default(0.1),
        shortflags([t,h]),  longflags([’heisenberg-threshold’]),
        help(’heisenberg threshold’)]

    , [opt(depth), meta(’K’), type(integer), default(3),
        shortflags([i,d]),longflags([depths,iters]),
        help(’stop after K iterations’)]

    , [opt(distances), default([1,2,3,5]),
        longflags([distances]),
        help(’initial prolog term’)]

    , [opt(outfile), meta(’FILE’), type(atom),
        shortflags([o]),  longflags([’output-file’]),
        help(’write output to FILE’)]
    , [opt(label), type(atom), default('REPORT'),
        shortflags([l]), longflags([label]),
        help('report label')]

    , [opt(verbose),  meta('V'), type(integer), default(2),
        shortflags([v]),  longflags([verbosity]),
        help('verbosity level, 1 <= V <= 3')]

    , [opt(path), default('/some/file/path/')]
    ].
```

The help text above was accessed by `opt_help(ExamplesOptsSpec, HelpText)`. The options appear in the same order as in the OptsSpec.

Given `ExampleOptsSpec`, a command line (somewhat syntactically inconsistent, in order to demonstrate different calling conventions) may look as follows

```
ExampleArgs = [ '-d5'
              , '--heisenberg-threshold', '0.14'
              , '--distances=[1,1,2,3,5,8]'
              , '--iters', '7'
              , '-ooutput.txt'
              , '--rebuild-cache', 'true'
              , 'input.txt'
              , '--verbosity=2'
              ].
```

opt_parse(ExampleOptsSpec, ExampleArgs, Opts, PositionalArgs) would then succeed with

```
Opts =     [ mode('SCAN')
           , label('REPORT')
           , path('/some/file/path')
           , threshold(0.14)
           , distances([1,1,2,3,5,8])
           , depth(7)
           , outfile('output.txt')
           , cache(true)
           , verbose(2)
           ],
PositionalArgs = ['input.txt'].
```

Note that path('/some/file/path') showing up in Opts has a default value (of the implicit type 'term'), but no corresponding flags in OptsSpec. Thus it can't be set from the command line. The rest of your program doesn't need to know that, of course. This provides an alternative to the common practice of asserting such hard-coded parameters under a single predicate (for instance setting(path,

'/some/file/path')), with the advantage that you may seamlessly upgrade them to command-line options, should you one day find this a good idea. Just add an appropriate flag or two and a line of help text. Similarly, suppressing an option in a cluttered interface amounts to commenting out the flags.

`opt_parse/5` allows more control through an additional argument list. For instance, opt_parse(ExampleOptsSpec, ExampleArgs, Opts, PositionalArgs, [output_functor(appl_config)]) would instead return

```
Opts =    [ appl_config(verbose, 2),
          , appl_config(label, 'REPORT')
          ...
          ]
```

This representation may be preferable with the empty-flag configuration parameter style above (perhaps with asserting `appl_config/2`).

A.16.1 Notes and tips

- In the example we were mostly explicit about the types. Since the default is `term`, which subsumes `integer, float, atom`, it may be possible to get away cheaper (e.g., by only giving booleans). However, it is recommended practice to always specify types: parsing becomes more reliable and error messages will be easier to interpret.

- Note that `-sbar` is taken to mean `-s bar`, not `-s -b -a -r`, that is, there is no clustering of flags.

- `-s=foo` is disallowed. The rationale is that although some command-line parsers will silently interpret this as `-s =foo`, this is very seldom what you want. To have an option argument start with '=' (very un-recommended), say so explicitly.

- The example specifies the option `depth` twice: once as `-d5` and once as `--iters 7`. The default when encountering duplicated flags is to `keeplast` (this behaviour can be controlled, by ParseOption duplicated_flags).

- The order of the options returned by the parsing functions is the same as given on the command line, with non-overridden defaults prepended and duplicates removed as in previous item. You should not rely on this, however.

- Unknown flags (not appearing in OptsSpec) will throw errors. This is usually a Good Thing. Sometimes, however, you may wish to pass along flags to an external program (say, one called by `shell/2`), and it means duplicated effort and a maintenance headache to have to specify all possible flags for the external program explicitly (if it even can be done). On the other hand, simply taking all unknown flags as valid makes error checking much less efficient and identification of positional arguments uncertain. A better solution is to collect all arguments intended for passing along to an indirectly called program as a single argument, probably as an atom (if you don't need to inspect them first) or as a prolog term (if you do).

opt_arguments(+*OptsSpec, -Opts, -PositionalArgs***)** *[det]*

Convenience predicate, assuming that command-line arguments can be accessed by `current_prolog_flag/2` (as in swi-prolog). For other access mechanisms and/or more control, get the args and pass them as a list of atoms to `opt_parse/4` or `opt_parse/5` instead.

Opts is a list of parsed options in the form Key(Value). Dashed args not in *OptsSpec* are not permitted and will raise error (see tip on how to pass unknown flags in the module description). *PositionalArgs* are the remaining non-dashed args after each flag has taken its argument (filling in `true` or `false` for booleans). There are no restrictions on non-dashed arguments and they may go anywhere (although it is good practice to put them last). Any leading arguments for the runtime (up to and including '–') are discarded.

opt_parse(+*OptsSpec, +ApplArgs, -Opts, -PositionalArgs***)** *[det]*

Equivalent to opt_parse(*OptsSpec*, *ApplArgs*, *Opts*, *PositionalArgs*, []).

opt_parse(+*OptsSpec, +ApplArgs, -Opts, -PositionalArgs, +ParseOptions***)** *[det]*

Parse the arguments Args (as list of atoms) according to *OptsSpec*. Any runtime arguments (typically terminated by '–') are assumed to be removed already.

Opts is a list of parsed options in the form Key(Value), or (with the option functor(Func) given) in the form Func(Key, Value). Dashed args not in *OptsSpec* are not permitted and will raise error (see tip on how to pass unknown flags in the module description). *PositionalArgs* are the remaining non-dashed args after each flag has taken its argument (filling in `true` or `false` for booleans). There are no restrictions on non-dashed arguments and they may go anywhere (although it is good practice to put them last). *ParseOptions* are

output_functor(*Func***)**

Set the functor Func of the returned options Func(Key,Value). Default is the special value 'OPTION' (upper-case), which makes the returned options have form Key(Value).

duplicated_flags(*Keep***)**

Controls how to handle options given more than once on the commad line. Keep is one of `keepfirst, keeplast, keepall` with the obvious meaning. Default is `keeplast`.

allow_empty_flag_spec(*Bool***)**

If true (default), a flag specification is not required (it is allowed that both shortflags and longflags be either [] or absent). Flagless options cannot be manipulated from the command line and will not show up in the generated help. This is useful when you have (also) general configuration parameters in your *OptsSpec*, especially if you think they one day might need to be controlled externally. See example in the module overview. allow_empty_flag_spec(false) gives the more customary behaviour of raising error on empty flags.

opt_help(+*OptsSpec, -Help***)** *[det]*

Returns a help string (atom), synthesized from the help descriptions in *OptsSpec*.

A.17 library(ordsets): Ordered set manipulation

Ordered sets are lists with unique elements sorted to the standard order of terms (see `sort/2`). Exploiting ordering, many of the set operations can be expressed in order N rather than N^2 when dealing with unordered sets that may contain duplicates. The library(ordsets) is available in a number of Prolog implementations. Our predicates are designed to be compatible with common practice in the Prolog community. The implementation is incomplete and relies partly on library(oset), an older ordered set library distributed with SWI-Prolog. New applications are advices to use library(ordsets).

Some of these predicates match directly to corresponding list operations. It is adviced to use the versions from this library to make clear you are operating on ordered sets.

is_ordset(@*Term***)** *[semidet]*

True if *Term* is an ordered set. All predicates in this library expect ordered sets as input arguments. Failing to fullfil this assumption results in undefined behaviour. Typically, ordered sets are created by predicates from this library, `sort/2` or `setof/3`.

ord_empty(*?List***)** *[semidet]*

True when *List* is the empty ordered set. Simply unifies list with the empty list. Not part of Quintus.

ord_seteq(+*Set1*, +*Set2***)** *[semidet]*

True if *Set1* and *Set2* have the same elements. As both are canonical sorted lists, this is the same as `==/2`.

Compatibility sicstus

list_to_ord_set(+*List*, -*OrdSet***)** *[det]*

Transform a list into an ordered set. This is the same as sorting the list.

ord_intersect(+*Set1*, +*Set2***)** *[semidet]*

True if both ordered sets have a non-empty intersection.

ord_disjoint(+*Set1*, +*Set2***)** *[semidet]*

True if *Set1* and *Set2* have no common elements. This is the negation of `ord_intersect/2`.

ord_intersect(+*Set1*, +*Set2*, -*Intersection***)**

Intersection holds the common elements of *Set1* and *Set2*.

deprecated Use `ord_intersection/3`

ord_intersection(+*PowerSet*, -*Intersection***)**

True if *Intersection* is an ordered set holding all elements common to all sets in *PowerSet*.

Compatibility sicstus

ord_intersection(+*Set1*, +*Set2*, -*Intersection***)** *[det]*

Intersection holds the common elements of *Set1* and *Set2*.

ord_intersection(+*Set1, +Set2, ?Intersection, ?Difference***)** *[det]*
Intersection and difference between two ordered sets. *Intersection* is the intersection between *Set1* and *Set2*, while *Difference* is defined by ord_subtract(*Set2*, *Set1*, *Difference*).

See also `ord_intersection/3` and `ord_subtract/3`.

ord_add_element(+*Set1, +Element, ?Set2***)** *[det]*
Insert an element into the set. This is the same as ord_union(*Set1*, [*Element*], *Set2*).

ord_del_element(+*Set, +Element, -NewSet***)** *[det]*
Delete an element from an ordered set. This is the same as ord_subtract(*Set*, [*Element*], *NewSet*).

ord_memberchk(+*Element, +Set***)** *[semidet]*
Check membership. This could stop comparing we have passed the right value, saving scanning (on average) half the list if *Element* is not in *Set*. Probably the built-in `memberchk/2` will be faster.

Compatibility Not part of original Quintus library

ord_member(*?Element, +Set***)** *[nondet]*
True if *Element* is a member of *Set*. Stops if further elements are behind *Element* in the standard order of terms.

Compatibility sicstus

ord_subset(+*Sub, +Super***)** *[semidet]*
Is true if all element of *Sub* are in *Super*

ord_subtract(+*InOSet, +NotInOSet, -Diff***)** *[det]*
Diff is the set holding all elements of *InOSet* that are not in *NotInOSet*.

ord_union(+*SetOfSets, -Union***)** *[det]*
True if *Union* is the union of all elements in the superset *SetOfSets*. Each member of *SetOfSets* must be an ordered set, the sets need not be ordered in any way.

author Copied from YAP, probably originally by Richard O'Keefe.

ord_union(+*Set1, +Set2, ?Union***)** *[det]*
Union is the union of *Set1* and *Set2*

ord_union(+*Set1, +Set2, -Union, -New***)** *[det]*
True if *Union* iff ord_union(*Set1*, *Set2*, *Union*) and ord_subtract(*Set2*, *Set1*, *New*).

ord_symdiff(+*Set1, +Set2, ?Difference***)** *[det]*
Is true when *Difference* is the symmetric difference of *Set1* and *Set2*. I.e., *Difference* contains all elements that are not in the intersection of *Set1* and *Set2*. The semantics is the same as the sequence below (but the actual implementation requires only a single scan).

```
        ord_union(Set1, Set2, Union),
        ord_intersection(Set1, Set2, Intersection),
        ord_subtract(Union, Intersection, Difference).
```

For example:

```
?- ord_symdiff([1,2], [2,3], X).
X = [1,3].
```

A.18 library(pairs): Operations on key-value lists

author Jan Wielemaker
See also `keysort/2`, library(assoc)

This module implements common operations on Key-Value lists, also known as *Pairs*. Pairs have great practical value, especially due to `keysort/2` and the library `assoc.pl`.

This library is based on disussion in the SWI-Prolog mailinglist, including specifications from Quintus and a library proposal by Richard O'Keefe.

pairs_keys_values(*?Pairs, ?Keys, ?Values***)** *[det]*

True if *Keys* holds the keys of *Pairs* and *Values* the values.

Deterministic if any argument is instantiated to a finite list and the others are either free or finite lists. All three lists are in the same order.

See also `pairs_values/2` and `pairs_keys/2`.

pairs_values(*+Pairs, -Values***)** *[det]*

Remove the keys from a list of Key-Value pairs. Same as pairs_keys_values(*Pairs*, _, *Values*)

pairs_keys(*+Pairs, -Keys***)** *[det]*

Remove the values from a list of Key-Value pairs. Same as pairs_keys_values(*Pairs*, *Keys*, _)

group_pairs_by_key(*+Pairs, -Joined:list(Key-Values)***)** *[det]*

Group values with the same key. *Pairs* must be a key-sorted list. For example:

```
?- group_pairs_by_key([a-2, a-1, b-4], X).

X = [a-[2,1], b-[4]]
```

Parameters

Pairs	*Key*-Value list, sorted to the standard order of terms (as `keysort/2` does)
Joined	List of *Key*-Group, where Group is the list of *Values* associated with *Key*.

transpose_pairs(*+Pairs, -Transposed***)** *[det]*

Swap Key-Value to Value-Key and sort the result on Value (the new key) using `keysort/2`.

map_list_to_pairs(*:Function, +List, -Keyed***)**

Create a key-value list by mapping each element of *List*. For example, if we have a list of lists we can create a list of Length-*List* using

```
        map_list_to_pairs(length, ListOfLists, Pairs),
```

A.19 `pio`: Pure I/O

This library provides pure list-based I/O processing for Prolog, where the communication to the actual I/O device is performed transparently through coroutining. This module itself is just an interface to the actual implementation modules.

A.19.1 library(pure_input): Pure Input from files

author
- Ulrich Neumerkel
- Jan Wielemaker

To be done
- Provide support for alternative input readers, e.g. reading terms, tokens, etc.
- Support non-repositioning streams, such as sockets and pipes.

This module is part of `pio.pl`, dealing with *pure input*: processing input streams from the outside world using pure predicates, notably grammar rules (DCG). Using pure predicates makes non-deterministic processing of input much simpler.

Pure input uses coroutining (`freeze/2`) to read input from the external source into a list *on demand*. The overhead of lazy reading is more than compensated for by using block reads based on `read_pending_input/3`.

phrase_from_file(*:Grammar, +File***)** *[nondet]*

Process the content of *File* using the DCG rule *Grammar*. The space usage of this mechanism depends on the length of the not committed part of *Grammar*. Committed parts of the temporary list are reclaimed by the garbage collector, while the list is extended on demand. Here is a very simple definition for searching a string in a file:

```
... --> []|[_],... .

file_contains(File, Pattern) :-
        phrase_from_file((..., Pattern, ...), File).

match_count(File, Pattern, Count) :-
        findall(x, file_contains(File, Pattern), Xs),
        length(Xs, Count).
```

This can be called as (note that the pattern must be a string (code list)):

```
?- match_count('pure_input.pl', "file", Count).
```

phrase_from_file(*:Grammar, +File, +Options***)** *[nondet]*

As `phrase_from_file/2`, providing additional *Options*. *Options* are passed to `open/4`, except for `buffer_size`, which is passed to `set_stream/2`. If not specified, the default buffer size is 512 bytes. Of particular importance are the `open/4` options `type` and `encoding`.

stream_to_lazy_list(*+Stream, -List***)** *[det]*

Create a lazy list representing the character codes in *Stream*. It must be possible to reposition *Stream*. *List* is a list that ends in a delayed goal. *List* can be unified completely transparent to a (partial) list and processed transparently using DCGs, but please be aware that a lazy list is not the same as a materialized list in all respects.

Typically, this predicate is used as a building block for more high level safe predicates such as `phrase_from_file/2`.

To be done Enhance of lazy list throughout the system.

A.20 `predicate_options`: Declare option-processing of predicates

Discussions with Jeff Schultz helped shaping this library

A.20.1 The strength and weakness of predicate options

Many ISO predicates accept options, e.g., `open/4`, `write_term/3`. Options offer an attractive alternative to proliferation into many predicates and using high-arity predicates. Properly defined and used, they also form a mechanism for extending the API of both system and application predicates without breaking portability. I.e., previously fixed behaviour can be replaced by dynamic behaviour controlled by an option where the default is the previously defined fixed behaviour. The alternative to using options is to add an additional argument and maintain the previous definition. While a series of predicates with increasing arity is adequate for a small number of additional parameters, the untyped positional argument handling of Prolog quickly makes this unmanageable.

The ISO standard uses the extensibility offered by options by allowing implementations to extend the set of accepted options. While options form a perfect solution to maintain backward portability in a linear development model, it is not well equipped to deal with concurrent branches because

1. There is no API to find which options are supported in a particular implementation.
2. While the portability problem caused by a missing predicate in Prolog *A* can easily be solved by implementing this predicate, it is much harder to add processing of an additional option to an already existing predicate.

Different Prolog implementations can be seen as concurrent development branches of the Prolog language. Different sets of supported options pose a serious portability issue. Using an option *O* that establishes the desired behaviour on system *A* leads (on most systems) to an error or system *B*. Porting may require several actions:

- Drop *O* (if the option is not vital, such as the layout options to `write_term/3`)
- Replace *O* by *O2* (i.e., a differently named option doing the same)

- Something else (cannot be ported; requires a totally different approach, etc.)

Predicates that process options are particularly a problem when writing a compatibility layer to run programs developed for System *A* on System *B* because complete emulation is often hard, may cause a serious slowdown and is often not needed because the application-to-be-ported only uses options that are shared by all target Prolog implementations. Unfortunately, the consequences of a partial emulation cannot be assessed by tools.

A.20.2 Options as arguments or environment?

We distinguish two views on options. One is to see them as additional parameters that require strict existence, type and domain-checking and the other is to consider them 'locally scoped environment variables'. Most systems adopt the first option. SWI-Prolog adopts the second: it silently ignores options that are not supported but does type and domain checking of option-values. The 'environment' view is commonly used in applications to create predicates supporting more options using the skeleton below. This way of programming requires that *pred1* and *pred2* do not interpret the same option differently. In cases where this is not true, the options must be distributed by *some_pred*. We have been using this programming style for many years and in practice it turns out that the need for active distribution of options is rare. I.e., options either have distinct names or multiple predicates implement the same option but this has the desired effect. An example of the latter is the `encoding` option, which typically needs to be applied consistently.

```
some_pred(..., Options) :-
      pred1(..., Options),
      pred2(..., Options).
```

As stated before, options provide a readable alternative to high-arity predicates and offer a robust mechanism to evolve the API, but at the cost of some runtime overhead and weaker consistency checking, both at compiletime and runtime. From our experience, the 'environment' approach is productive, but the consequence is that mistyped options are silently ignored. The option infrastructure described in this section tries to remedy these problems.

A.20.3 Improving on the current situation

Whether we see options as arguments or locally scoped environment variables, the most obvious way to improve on the current situation is to provide reflective support for options: discover that an argument is an option-list and find what options are supported. Reflective access to options can be used by the compiler and development environment as well as by the runtime system to warn or throw errors.

Options as types

An obvious approach to deal with options is to define the different possible option values as a type and type the argument that processes the option as list(<option_type>), as illustrated below. Considering options as types fully covers the case where we consider options as additional parameters.

```
:- type open_option ---> type(stream_type) |
                         alias(atom) | ... .
:- pred open(source_sink, open_mode, stream, list(open_option)).
```

There are three reasons for considering a different approach:

- There is no consensus about types in the Prolog world, neither about what types should look like, nor whether or not they are desirable. It is not likely that this debate will be resolved shortly.
- Considering options as types does not support the ‘environment’ view, which we consider the most productive.
- Even when using types, we need reflective access to what options are provided in order to be able to write compile or runtime conditional code.

Reflective access to options

From the above, we conclude that we require reflective access to find out whether an option is supported and valid for a particular predicate. Possible option values must be described by types. Due to lack of a type system, we use `library(error)` to describe allowed option values. Predicate options are declared using `predicate_options/3`:

predicate_options(*:PI, +Arg, +Options***)** *[det]*

Declare that the predicate *PI* processes options on *Arg*. *Options* is a list of options processed. Each element is one of:

- Option(ModeAndType) *PI* processes Option. The option-value must comply to ModeAndType. Mode is one of + or - and Type is a type as accepted by `must_be/2`.
- pass_to(:*PI*,*Arg*) The option-list is passed to the indicated predicate.

Below is an example that processes the option header(boolean) and passes all options to `open/4`:

```
:- predicate_options(write_xml_file/3, 3,
                     [ header(boolean),
                       pass_to(open/4, 4)
                     ]).

write_xml_file(File, XMLTerm, Options) :-
    open(File, write, Out, Options),
    (   option(header(true), Option, true)
    ->  write_xml_header(Out)
    ;   true
    ),
    ...
```

This predicate may only be used as a *directive* and is processed by `expand_term/2`. Option processing can be be specified at runtime using `assert_predicate_options/3`, which is intended to support program analysis.

assert_predicate_options(*:PI, +Arg, +Options, ?New***)** *[semidet]*
As predicate_options(:*PI*, +*Arg*, +*Options*). *New* is a boolean indicating whether the declarations have changed. If new is provided and `false`, the predicate becomes semidet and fails without modifications if modifications are required.

The predicates below realise the support for compile and runtime checking for supported options.

current_predicate_option(*:PI, ?Arg, ?Option***)** *[nondet]*
True when *Arg* of *PI* processes *Option*. For example, the following is true:

```
?- current_predicate_option(open/4, 4, type(text)).
true.
```

This predicate is intended to support conditional compilation using `if/1` ... `endif/0`. The predicate `current_predicate_options/3` can be used to access the full capabilities of a predicate.

check_predicate_option(*:PI, +Arg, +Option***)** *[det]*
Similar to `current_predicate_option/3`, but intended to support runtime checking.

Errors
- existence_error(option, OptionName) if the option is not supported by *PI*.
- type_error(Type, Value) if the option is supported but the value does not match the option type. See `must_be/2`.

The predicates below can be used in a development environment to inform the user about supported options. PceEmacs uses this for colouring option names and values.

current_option_arg(*:PI, ?Arg***)** *[nondet]*
True when *Arg* of *PI* processes predicate options. Which options are processed can be accessed using `current_predicate_option/3`.

current_predicate_options(*:PI, ?Arg, ?Options***)** *[nondet]*
True when *Options* is the current active option declaration for *PI* on *Arg*. See `predicate_options/3` for the argument descriptions. If *PI* is ground and refers to an undefined predicate, the autoloader is used to obtain a definition of the predicate.

The library can execute a complete check of your program using `check_predicate_options/0`:

check_predicate_options *[det]*
Analyse loaded program for errornous options. This predicate decompiles the current program and searches for calls to predicates that process options. For each option list, it validates whether the provided options are supported and validates the argument type. This predicate performs partial dataflow analysis to track option-lists inside a clause.

See also `derive_predicate_options/0` can be used to derive declarations for predicates that pass options. This predicate should normally be called before `check_predicate_options/0`.

The library offers predicates that may be used to create declarations for your application. These predicates are designed to cooperate with the module system.

derive_predicate_options *[det]*

Derive new predicate option declarations. This predicate analyses the loaded program to find clauses that process options using one of the predicates from `library(option)` or passes options to other predicates that are known to process options. The process is repeated until no new declarations are retrieved.

See also `autoload/0` may be used to complete the loaded program.

retractall_predicate_options *[det]*

Remove all dynamically (derived) predicate options.

derived_predicate_options(*:PI, ?Arg, ?Options***)** *[nondet]*

True when *Options* is the current *derived* active option declaration for *PI* on *Arg*.

derived_predicate_options(+*Module***)** *[det]*

Derive predicate option declarations for the given module and print them to the current output.

A.21 `prolog_xref`: Cross-reference data collection library

This library collects information on defined and used objects in Prolog source files. Typically these are predicates, but we expect the library to deal with other types of objects in the future. The library is a building block for tools doing dependency tracking in applications. Dependency tracking is useful to reveal the structure of an unknown program or detect missing components at compile time, but also for program transformation or minimising a program saved state by only saving the reachable objects.

This section gives a partial description of the library API, providing some insight in how you can use it for analysing your program. The library should be further modularized, moving its knowledge about, for example, XPCE into a different file and allowing for adding knowledge about other libraries such as Logtalk. **Please do not consider this interface rock-solid.**

The library is exploited by two graphical tools in the SWI-Prolog environment: the XPCE front-end started by `gxref/0` and described in section 3.7, and PceEmacs (section 3.4), which exploits this library for its syntax colouring.

For all predicates described below, *Source* is the source that is processed. This is normally a filename in any notation acceptable to the file loading predicates (see `load_files/2`). Using the hooks defined in section A.21.1 it can be anything else that can be translated into a Prolog stream holding Prolog source text. *Callable* is a callable term (see `callable/1`). Callables do not carry a module qualifier unless the referred predicate is not in the module defined *Source*.

xref_source(+*Source***)**

Gather information on *Source*. If *Source* has already been processed and is still up-to-date according to the file timestamp, no action is taken. This predicate must be called on a file before information can be gathered.

xref_current_source(*?Source*)
Source has been processed.

xref_clean(*+Source*)
Remove the information gathered for *Source*

xref_defined(*?Source, ?Callable, -How*)
Callable is defined in *Source*. *How* is one of

`dynamic`(*Line*)	Declared dynamic at *Line*
`thread_local`(*Line*)	Declared thread local at *Line*
`multifile`(*Line*)	Declared multifile at *Line*
`local`(*Line*)	First clause at *Line*
`foreign`(*Line*)	Foreign library loaded at *Line*
`constraint`(*Line*)	CHR Constraint at *Line*
`imported`(*File*)	Imported from *File*

xref_called(*?Source, ?Callable, ?By*)
Callable is called in *Source* by *By*.

xref_exported(*?Source, ?Callable*)
Callable is public (exported from the module).

xref_module(*?Source, ?Module*)
Source is a module file defining the given module.

xref_built_in(*?Callable*)
True if *Callable* is a built-in predicate. Currently this is assumed for all predicates defined in the `system` module and having the property `built_in`. Built-in predicates are not registered as 'called'.

A.21.1 Extending the library

The library provides hooks for extending the rules it uses for finding predicates called by some programming construct.

prolog:called_by(*+Goal, -Called*)
Goal is a non-var subgoal appearing in the called object (typically a clause body). If it succeeds it must return a list of goals called by *Goal*. As a special construct, if a term $Callable + N$ is returned, N variable arguments are added to *Callable* before further processing. For simple meta-calls a single fact suffices. Complex rules as used in the `html_write` library provided by the HTTP package examine the arguments and create a list of called objects.

The current system cannot deal with the same name/arity in different modules that behave differently with respect to called arguments.

A.22 library(random): Random numbers

author R.A. O'Keefe, V.S. Costa, L. Damas, Jan Wielemaker
See also Built-in function `random/1`: A is random(10)

This library is derived from the DEC10 library random. Later, the core random generator was moved to C. The current version uses the SWI-Prolog arithmetic functions to realise this library. These functions are based on the GMP library.

random(-*R:float***)** *[det]*

Binds *R* to a new random float in the *open* interval (0.0,1.0).

See also

- `setrand/1`, `getrand/1` maye be used to fetch/set the state.
- In SWI-Prolog, `random/1` is implemented by the function `random_float/0`.

random_between(+*L:int*, +*U:int*, -*R:int***)** *[semidet]*

Binds *R* to a random integer in [*L*,*U*] (i.e., including both *L* and *U*). Fails silently if *U*<*L*.

random(+*L:int*, +*U:int*, -*R:int***)** *[det]*
random(+*L:float*, +*U:float*, -*R:float***)** *[det]*

If *L* and *U* are both integers, *R* is a random integer in the half open interval [*L*,*U*). If *L* and *U* are both floats, *R* is a float in the open interval (*L*,*U*).

deprecated Please use `random/1` for generating a random float and `random_between/3` for generating a random integer. Note that the `random_between/3` includes the upper bound, while this predicate excludes the upper bound.

setrand(+*State***)** *[det]*
getrand(-*State***)** *[det]*

Query/set the state of the random generator. This is intended for restarting the generator at a known state only. The predicate `setrand/1` accepts an opaque term returned by `getrand/1`. This term may be asserted, written and read. The application may not make other assumptions about this term.

For compatibility reasons with older versions of this library, `setrand/1` also accepts a term rand(A,B,C), where A, B and C are integers in the range 1..30,000. This argument is used to seed the random generator. Deprecated.

Errors existence_error(random_state, _) is raised if the underlying infrastructure cannot fetch the random state. This is currently the case if SWI-Prolog is not compiled with the GMP library.

See also `set_random/1` and `random_property/1` provide the SWI-Prolog native implementation.

maybe *[semidet]*

Succeed/fail with equal probability (variant of `maybe/1`).

maybe(+*P***)** *[semidet]*

Succeed with probability *P*, fail with probability 1-*P*

maybe(+*K*, +*N***)** *[semidet]*

Succeed with probability *K*/*N* (variant of `maybe/1`)

random_perm2(?*A*, ?*B*, ?*X*, ?*Y***)** *[semidet]*

Does *X*=*A*,*Y*=*B* or *X*=*B*,*Y*=*A* with equal probability.

random_member(*-X, +List:list***)** *[semidet]*

X is a random member of *List*. Equivalent to random_between(1, |*List*|), followed by `nth1/3`. Fails of *List* is the empty list.

Compatibility Quintus and SICStus libraries.

random_select(*-X, +List, -Rest***)** *[semidet]*
random_select(*+X, -List, +Rest***)** *[det]*

Randomly select or insert an element. Either *List* or *Rest* must be a list. Fails if *List* is the empty list.

Compatibility Quintus and SICStus libraries.

randset(*+K:int, +N:int, -S:list(int)***)** *[det]*

S is a sorted list of *K* unique random integers in the range 1..*N*. Implemented by enumerating 1..*N* and deciding whether or not the number should be part of the set. For example:

```
?- randset(5, 5, S).
S = [1, 2, 3, 4, 5].                (always)
?- randset(5, 20, S).
S = [2, 7, 10, 19, 20].
```

See also `randseq/3`.
bug Slow if *N* is large and *K* is small.

randseq(*+K:int, +N:int, -List:list(int)***)** *[det]*

S is a list of *K* unique random integers in the range 1..*N*. The order is random. Works as if defined by the following code.

```
randseq(K, N, List) :-
        randset(K, N, Set),
        random_permutation(Set, List).
```

See also `randset/3`.

random_permutation(*+List, -Permutation***)** *[det]*
random_permutation(*-List, +Permutation***)** *[det]*

Permutation is a random permutation of *List*. This is intended to process the elements of *List* in random order. The predicate is symetric.

Errors instantiation_error, type_error(list, _).

A.23 `readutil`: Reading lines, streams and files

This library contains primitives to read lines, files, multiple terms, etc. The package `clib` provides a shared object (DLL) named `readutil`. If the library can locate this shared object it will use the foreign implementation for reading character codes. Otherwise it will use a Prolog implementation. Distributed applications should make sure to deliver the `readutil` shared object if performance of these predicates is critical.

read_line_to_codes(+*Stream, -Codes***)**

Read the next line of input from *Stream* and unify the result with *Codes after* the line has been read. A line is ended by a newline character or end-of-file. Unlike `read_line_to_codes/3`, this predicate removes a trailing newline character.

On end-of-file the atom `end_of_file` is returned. See also `at_end_of_stream/[0,1]`.

read_line_to_codes(+*Stream, -Codes, ?Tail***)**

Difference-list version to read an input line to a list of character codes. Reading stops at the newline or end-of-file character, but unlike `read_line_to_codes/2`, the newline is retained in the output. This predicate is especially useful for reading a block of lines up to some delimiter. The following example reads an HTTP header ended by a blank line:

```
read_header_data(Stream, Header) :-
        read_line_to_codes(Stream, Header, Tail),
        read_header_data(Header, Stream, Tail).

read_header_data("\r\n", _, _) :- !.
read_header_data("\n", _, _) :- !.
read_header_data("", _, _) :- !.
read_header_data(_, Stream, Tail) :-
        read_line_to_codes(Stream, Tail, NewTail),
        read_header_data(Tail, Stream, NewTail).
```

read_stream_to_codes(+*Stream, -Codes***)**

Read all input until end-of-file and unify the result to *Codes*.

read_stream_to_codes(+*Stream, -Codes, ?Tail***)**

Difference-list version of `read_stream_to_codes/2`.

read_file_to_codes(+*Spec, -Codes, +Options***)**

Read a file to a list of character codes. *Spec* is a file specification for `absolute_file_name/3`. *Codes* is the resulting code list. *Options* is a list of options for `absolute_file_name/3` and `open/4`. In addition, the option `tail`(*Tail*) is defined, forming a difference-list.

read_file_to_terms(+*Spec, -Terms, +Options***)**

Read a file to a list of Prolog terms (see `read/1`). *Spec* is a file specification for `absolute_file_name/3`. *Terms* is the resulting list of Prolog terms. *Options* is a list of options for `absolute_file_name/3` and `open/4`. In addition, the option `tail`(*Tail*) is defined, forming a difference-list.

A.24 `record`: Access named fields in a term

The library `record` provides named access to fields in a record represented as a compound term such as `point(X, Y)`. The Prolog world knows various approaches to solve this problem, unfortunately with no consensus. The approach taken by this library is proposed by Richard O'Keefe on the SWI-Prolog mailinglist.

The approach automates a technique commonly described in Prolog text-books, where access and modification predicates are defined for the record type. Such predicates are subject to normal import/export as well as analysis by cross-referencers. Given the simple nature of the access predicates, an optimizing compiler can easily inline them for optimal preformance.

A record is defined using the directive `record/1`. We introduce the library with a short example:

```
:- record point(x:integer=0, y:integer=0).

        ...,
        default_point(Point),
        point_x(Point, X),
        set_x_of_point(10, Point, Point1),

        make_point([y(20)], YPoint),
```

The principal functor and arity of the term used defines the name and arity of the compound used as records. Each argument is described using a term of the format below.

⟨*name*⟩[:⟨*type*⟩][=⟨*default*⟩]

In this definition, ⟨*name*⟩ is an atom defining the name of the argument, ⟨*type*⟩ is an optional type specification as defined by `must_be/2` from library `error`, and ⟨*default*⟩ is the default initial value. The ⟨*type*⟩ defaults to `any`. If no default value is specified the default is an unbound variable.

A record declaration creates a set of predicates through *term-expansion*. We describe these predicates below. In this description, ⟨*constructor*⟩ refers to the name of the record ('point' in the example above) and ⟨*name*⟩ to the name of an argument (field).

- *default_⟨constructor⟩(-Record)*
  Create a new record where all fields have their default values. This is the same as make_⟨*constructor*⟩([], Record).

- *make_⟨constructor⟩(+Fields, -Record)*
  Create a new record where specified fields have the specified values and remaining fields have their default value. Each field is specified as a term ⟨*name*⟩(⟨*value*⟩). See example in the introduction.

- *make_⟨constructor⟩(+Fields, -Record, -RestFields)*
  Same as make_⟨*constructor*⟩/2, but named fields that do not appear in *Record* are returned in *RestFields*. This predicate is motivated by option-list processing. See library `option`.

- *⟨constructor⟩_⟨name⟩(Record, Value)*
  Unify *Value* with argument in *Record* named ⟨*name*⟩.[2]

- *set_⟨name⟩_of_⟨constructor⟩(+Value, +OldRecord, -NewRecord)*
  Replace the value for ⟨*name*⟩ in *OldRecord* by *Value* and unify the result with *NewRecord*.

---

[2] Note this is not called 'get_' as it performs unification and can perfectly well instantiate the argument.

- *set_⟨name⟩_of_⟨constructor⟩(+Value, !Record)*
  Destructively replace the argument ⟨*name*⟩ in *Record* by *Value* based on `setarg/3`. Use with care.

- *nb_set_⟨name⟩_of_⟨constructor⟩(+Value, !Record)*
  As above, but using non-backtrackable assignment based on `nb_setarg/3`. Use with *extreme* care.

- *set_⟨constructor⟩_fields(+Fields, +Record0, -Record)*
  Set multiple fields using the same syntax as make_⟨*constructor*⟩/2, but starting with *Record0* rather than the default record.

- *set_⟨constructor⟩_fields(+Fields, +Record0, -Record, -RestFields)*
  Similar to set_⟨*constructor*⟩_fields/4, but fields not defined by ⟨*constructor*⟩ are returned in *RestFields*.

- *set_⟨constructor⟩_field(+Field, +Record0, -Record)*
  Set a single field specified as a term ⟨*name*⟩(⟨*value*⟩).

record(+*Spec***)**
The construct `:- record Spec, ...` is used to define access to named fields in a compound. It is subject to term-expansion (see `expand_term/2`) and cannot be called as a predicate. See section A.24 for details.

A.25 `registry`: Manipulating the Windows registry

The `registry` is only available on the MS-Windows version of SWI-Prolog. It loads the foreign extension `plregtry.dll`, providing the predicates described below. This library only makes the most common operations on the registry available through the Prolog user. The underlying DLL provides a more complete coverage of the Windows registry API. Please consult the sources in `pl/src/win32/foreign/plregtry.c` for further details.

In all these predicates, *Path* refers to a '/' separated path into the registry. This is *not* an atom containing '/'-characters as used for filenames, but a term using the functor `//2`. Windows defines the following roots for the registry: `classes_root`, `current_user`, `local_machine` and `users`.

registry_get_key(+*Path*, -*Value***)**
Get the principal (default) value associated to this key. Fails silently if the key does not exist.

registry_get_key(+*Path*, +*Name*, -*Value***)**
Get a named value associated to this key.

registry_set_key(+*Path*, +*Value***)**
Set the principal (default) value of this key. Creates (a path to) the key if it does not already exist.

registry_set_key(+*Path*, +*Name*, +*Value***)**
Associate a named value to this key. Creates (a path to) the key if it does not already exist.

registry_delete_key(+*Path***)**
Delete the indicated key.

shell_register_file_type(+*Ext*, +*Type*, +*Name*, +*OpenAction***)**
Register a file-type. *Ext* is the extension to associate. *Type* is the type name, often something like `prolog.type`. *Name* is the name visible in the Windows file-type browser. Finally, *OpenAction* defines the action to execute when a file with this extension is opened in the Windows explorer.

shell_register_dde(+*Type*, +*Action*, +*Service*, +*Topic*, +*Command*, +*IfNotRunning***)**
Associate DDE actions to a type. *Type* is the same type as used for the 2nd argument of `shell_register_file_type/4`, *Action* is the action to perform, *Service* and *Topic* specify the DDE topic to address, and *Command* is the command to execute on this topic. Finally, *IfNotRunning* defines the command to execute if the required DDE server is not present.

shell_register_prolog(+*Ext***)**
Default registration of SWI-Prolog, which is invoked as part of the initialisation process on Windows systems. As the source also includes the above predicates, it is given as an example:

```
shell_register_prolog(Ext) :-
        current_prolog_flag(argv, [Me|_]),
        concat_atom(['"', Me, '" "%1"'], OpenCommand),
        shell_register_file_type(Ext, 'prolog.type', 'Prolog Source',
                                 OpenCommand),
        shell_register_dde('prolog.type', consult,
                           prolog, control, 'consult(''%1'')', Me),
        shell_register_dde('prolog.type', edit,
                           prolog, control, 'edit(''%1'')', Me).
```

A.26 `simplex`: Solve linear programming problems

Author: *Markus Triska*

A linear programming problem consists of a set of (linear) constraints, a number of variables and a linear objective function. The goal is to assign values to the variables so as to maximize (or minimize) the value of the objective function while satisfying all constraints.

Many optimization problems can be modeled in this way. Consider having a knapsack with fixed capacity C, and a number of items with sizes s(i) and values v(i). The goal is to put as many items as possible in the knapsack (not exceeding its capacity) while maximizing the sum of their values.

As another example, suppose you are given a set of coins with certain values, and you are to find the minimum number of coins such that their values sum up to a fixed amount. Instances of these problems are solved below.
The `simplex` module provides the following predicates:

assignment(+*Cost*, -*Assignment***)**
Cost is a list of lists representing the quadratic cost matrix, where element (i,j) denotes the cost of assigning entity i to entity j. An assignment with minimal cost is computed and unified with *Assignment* as a list of lists, representing an adjacency matrix.

constraint(*+Constraint, +S0, -S*)

Adds a linear or integrality constraint to the linear program corresponding to state *S0*. A linear constraint is of the form ”Left Op C”, where ”Left” is a list of Coefficient*Variable terms (variables in the context of linear programs can be atoms or compound terms) and C is a non-negative numeric constant. The list represents the sum of its elements. *Op* can be =, =¡ or ¿=. The coefficient ”1” can be omitted. An integrality constraint is of the form integral(Variable) and constrains Variable to an integral value.

constraint(*+Name, +Constraint, +S0, -S*)

Like `constraint/3`, and attaches the name *Name* (an atom or compound term) to the new constraint.

constraint_add(*+Name, +Left, +S0, -S*)

Left is a list of Coefficient*Variable terms. The terms are added to the left-hand side of the constraint named *Name*. *S* is unified with the resulting state.

gen_state(*-State*)

Generates an initial state corresponding to an empty linear program.

maximize(*+Objective, +S0, -S*)

Maximizes the objective function, stated as a list of ”Coefficient*Variable” terms that represents the sum of its elements, with respect to the linear program corresponding to state *S0*. *S* is unified with an internal representation of the solved instance.

minimize(*+Objective, +S0, -S*)

Analogous to `maximize/3`.

objective(*+State, -Objective*)

Unifies *Objective* with the result of the objective function at the obtained extremum. *State* must correspond to a solved instance.

shadow_price(*+State, +Name, -Value*)

Unifies *Value* with the shadow price corresponding to the linear constraint whose name is *Name*. *State* must correspond to a solved instance.

transportation(*+Supplies, +Demands, +Costs, -Transport*)

Supplies and *Demands* are both lists of positive numbers. Their respective sums must be equal. *Costs* is a list of lists representing the cost matrix, where an entry (i,j) denotes the cost of transporting one unit from *i* to *j*. A transportation plan having minimum cost is computed and unified with *Transport* in the form of a list of lists that represents the transportation matrix, where element (i,j) denotes how many units to ship from *i* to *j*.

variable_value(*+State, +Variable, -Value*)

Value is unified with the value obtained for *Variable*. *State* must correspond to a solved instance.

All numeric quantities are converted to rationals via `rationalize/1`, and rational arithmetic is used throughout solving linear programs. In the current implementation, all variables are implicitly constrained to be non-negative. This may change in future versions, and non-negativity constraints should therefore be stated explicitly.

A.26.1 Example 1

This is the ”radiation therapy” example, taken from ”Introduction to Operations Research” by Hillier and Lieberman. DCG notation is used to implicitly thread the state through posting the constraints:

```
:- use_module(library(simplex)).

post_constraints -->
        constraint([0.3*x1, 0.1*x2] =< 2.7),
        constraint([0.5*x1, 0.5*x2] = 6),
        constraint([0.6*x1, 0.4*x2] >= 6),
        constraint([x1] >= 0),
        constraint([x2] >= 0).

radiation(S) :-
        gen_state(S0),
        post_constraints(S0, S1),
        minimize([0.4*x1, 0.5*x2], S1, S).
```

An example query:

```
?- radiation(S), variable_value(S, x1, Val1), variable_value(S, x2, Val2).

Val1 = 15 rdiv 2
Val2 = 9 rdiv 2 ;
```

A.26.2 Example 2

Here is an instance of the knapsack problem described above, where C = 8, and we have two types of items: One item with value 7 and size 6, and 2 items each having size 4 and value 4. We introduce two variables, x(1) and x(2) that denote how many items to take of each type.

```
knapsack_constrain(S) :-
        gen_state(S0),
        constraint([6*x(1), 4*x(2)] =< 8, S0, S1),
        constraint([x(1)] =< 1, S1, S2),
        constraint([x(2)] =< 2, S2, S).

knapsack(S) :-
        knapsack_constrain(S0),
        maximize([7*x(1), 4*x(2)], S0, S).
```

An example query yields:

```
?- knapsack(S), variable_value(S, x(1), X1), variable_value(S, x(2), X2).
```

```
X1 = 1
X2 = 1 rdiv 2 ;
```

That is, we are to take the one item of the first type, and half of one of the items of the other type to maximize the total value of items in the knapsack.

If items can not be split, integrality constraints have to be imposed:

```
knapsack_integral(S) :-
        knapsack_constrain(S0),
        constraint(integral(x(1)), S0, S1),
        constraint(integral(x(2)), S1, S2),
        maximize([7*x(1), 4*x(2)], S2, S).
```

Now the result is different:

```
?- knapsack_integral(S), variable_value(S, x(1), X1), variable_value(S, x(2), X2)

X1 = 0
X2 = 2
```

That is, we are to take only the two items of the second type. Notice in particular that always choosing the remaining item with best performance (ratio of value to size) that still fits in the knapsack does not necessarily yield an optimal solution in the presence of integrality constraints.

A.26.3 Example 3

We are given 3 coins each worth 1, 20 coins each worth 5, and 10 coins each worth 20 units of money. The task is to find a minimal number of these coins that amount to 111 units of money. We introduce variables c(1), c(5) and c(20) denoting how many coins to take of the respective type:

```
coins -->
        constraint([c(1), 5*c(5), 20*c(20)] = 111),
        constraint([c(1)] =< 3),
        constraint([c(5)] =< 20),
        constraint([c(20)] =< 10),
        constraint([c(1)] >= 0),
        constraint([c(5)] >= 0),
        constraint([c(20)] >= 0),
        constraint(integral(c(1))),
        constraint(integral(c(5))),
        constraint(integral(c(20))),
        minimize([c(1), c(5), c(20)]).

coins(S) :-
        gen_state(S0),
        coins(S0, S).
```

An example query:

```
?- coins(S), variable_value(S, c(1), C1), variable_value(S, c(5), C5), variable_v

C1 = 1
C5 = 2
C20 = 5
```

A.27 library(thread_pool): Resource bounded thread management

See also `http_handler/3` and `http_spawn/2`.

The module library(thread_pool) manages threads in pools. A pool defines properties of its member threads and the maximum number of threads that can coexist in the pool. The call `thread_create_in_pool/4` allocates a thread in the pool, just like `thread_create/3`. If the pool is fully allocated it can be asked to wait or raise an error.

The library has been designed to deal with server application that recieve a variety of requests, such as HTTP servers. Simply starting a thread for each request is a bit too simple minded for such servers:

- Creating many CPU intensive threads often leads to a slow-down rather than a speedup.
- Creating many memory intensive threads may exhaust resources
- Tasks that require little CPU and memory but take long waiting for external resources can run many threads.

Using this library, one can define a pool for each set of tasks with comparable characteristics and create threads in this pool. Unlike the worker-pool model, threads are not started immediately. Depending on the design, both approaches can be attractive.

The library is implemented by means of a manager thread with the fixed thread id `__thread_pool_manager`. All state is maintained in this manager thread, which receives and processes requests to create and destroy pools, create threads in a pool and handle messages from terminated threads. Thread pools are *not* saved in a saved state and must therefore be recreated using the `initialization/1` directive or otherwise during startup of the application.

thread_pool_create(+*Pool*, +*Size*, +*Options***)** *[det]*

Create a pool of threads. A pool of threads is a declaration for creating threads with shared properties (stack sizes) and a limited number of threads. Threads are created using `thread_create_in_pool/4`. If all threads in the pool are in use, the behaviour depends on the `wait` option of `thread_create_in_pool/4` and the `backlog` option described below. *Options* are passed to `thread_create/3`, except for

backlog(+*MaxBackLog***)**
Maximum number of requests that can be suspended. Default is `infinite`. Otherwise it must be a non-negative integer. Using backlog(0) will never delay thread creation for this pool.

The pooling mechanism does *not* interact with the `detached` state of a thread. Threads can be created both `detached` and normal and must be joined using `thread_join/2` if they are not detached.

bug The thread creation option `at_exit` is reserved for internal use by this library.

thread_pool_destroy(+*Name***)** *[det]*
Destroy the thread pool named *Name*.

Errors existence_error(thread_pool, *Name*).

current_thread_pool(*?Name***)** *[nondet]*
True if *Name* refers to a defined thread pool.

thread_pool_property(*?Name, ?Property***)** *[nondet]*
True if *Property* is a property of thread pool *Name*. Defined properties are:

options(*Options***)**
Thread creation options for this pool

free(*Size***)**
Number of free slots on this pool

size(*Size***)**
Total number of slots on this pool

members(*ListOfIDs***)**
ListOfIDs is the list or threads running in this pool

running(*Running***)**
Number of running threads in this pool

backlog(*Size***)**
Number of delayed thread creations on this pool

thread_create_in_pool(+*Pool, :Goal, -Id,* +*Options***)** *[det]*
Create a thread in *Pool*. *Options* overrule default thread creation options associated to the pool. In addition, the following option is defined:

wait(+*Boolean***)**
If `true` (default) and the pool is full, wait until a member of the pool completes. If `false`, throw a resource_error.

Errors
- resource_error(threads_in_pool(*Pool*)) is raised if wait is `false` or the backlog limit has been reached.
- existence_error(thread_pool, *Pool*) if *Pool* does not exist.

A.28 `ugraphs`: Unweighted Graphs

Authors: *Richard O'Keefe & Vitor Santos Costa*

> *Implementation and documentation are copied from YAP 5.0.1. The* `ugraph` *library is based on code originally written by Richard O'Keefe. The code was then extended to be compatible with the SICStus Prolog ugraphs library. Code and documentation have been cleaned and style has been changed to be more in line with the rest of SWI-Prolog.*
>
> *The ugraphs library was originally released in the public domain. The YAP version is covered by the Perl Artistic license, version 2.0. This code is dual-licensed under the modified GPL as used for all SWI-Prolog libraries or the Perl Artistic license, version 2.0.*

The routines assume directed graphs; undirected graphs may be implemented by using two edges.

Originally graphs were represented in two formats. The SICStus library and this version of `ugraphs.pl` only use the *S-representation*. The S-representation of a graph is a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by keysort) and the neighbors of each vertex are also in standard order (as produced by sort). This form is convenient for many calculations. Each vertex appears in the S-representation, even if it has no neighbors.

vertices_edges_to_ugraph(+*Vertices*, +*Edges*, -*Graph***)**

Given a graph with a set of *Vertices* and a set of *Edges*, *Graph* must unify with the corresponding S-representation. Note that vertices without edges will appear in *Vertices* but not in *Edges*. Moreover, it is sufficient for a vertex to appear in *Edges*.

```
?- vertices_edges_to_ugraph([],[1-3,2-4,4-5,1-5], L).
L = [1-[3,5], 2-[4], 3-[], 4-[5], 5-[]]
```

In this case all vertices are defined implicitly. The next example shows three unconnected vertices:

```
?- vertices_edges_to_ugraph([6,7,8],[1-3,2-4,4-5,1-5], L).
L = [1-[3,5], 2-[4], 3-[], 4-[5], 5-[], 6-[], 7-[], 8-[]] ?
```

vertices(+*Graph*, -*Vertices***)**

Unify *Vertices* with all vertices appearing in *Graph*. Example:

```
?- vertices([1-[3,5],2-[4],3-[],4-[5],5-[]], L).
L = [1, 2, 3, 4, 5]
```

edges(+*Graph*, -*Edges***)**

Unify *Edges* with all edges appearing in *Graph*. Example:

```
?- edges([1-[3,5],2-[4],3-[],4-[5],5-[]], L).
L = [1-3, 1-5, 2-4, 4-5]
```

add_vertices(*+Graph, +Vertices, -NewGraph***)**

Unify *NewGraph* with a new graph obtained by adding the list of *Vertices* to *Graph*. Example:

```
?- add_vertices([1-[3,5],2-[]], [0,1,2,9], NG).
NG = [0-[], 1-[3,5], 2-[], 9-[]]
```

del_vertices(*+Graph, +Vertices, -NewGraph***)**

Unify *NewGraph* with a new graph obtained by deleting the list of *Vertices* and all edges that start from or go to a vertex in *Vertices* from *Graph*. Example:

```
?- del_vertices([2,1],
                [1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[2,6],8-[]],
                NL).
NL = [3-[],4-[5],5-[],6-[],7-[6],8-[]]
```

add_edges(*+Graph, +Edges, -NewGraph***)**

Unify *NewGraph* with a new graph obtained by adding the list of *Edges* to *Graph*. Example:

```
?- add_edges([1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[],8-[]],
             [1-6,2-3,3-2,5-7,3-2,4-5],
             NL).
NL = [1-[3,5,6], 2-[3,4], 3-[2], 4-[5], 5-[7], 6-[], 7-[], 8-[]]
```

del_edges(*+Graph, +Edges, -NewGraph***)**

Unify *NewGraph* with a new graph obtained by removing the list of *Edges* from *Graph*. Notice that no vertices are deleted. Example:

```
?- del_edges([1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[],8-[]],
             [1-6,2-3,3-2,5-7,3-2,4-5,1-3],
             NL).
NL = [1-[5],2-[4],3-[],4-[],5-[],6-[],7-[],8-[]]
```

transpose(*+Graph, -NewGraph***)**

Unify *NewGraph* with a new graph obtained from *Graph* by replacing all edges of the form V1-V2 by edges of the form V2-V1. The cost is $O(|V|^2)$. Notice that an undirected graph is its own transpose. Example:

```
?- transpose([1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[],8-[]], NL).
NL = [1-[],2-[],3-[1],4-[2],5-[1,4],6-[],7-[],8-[]]
```

neighbours(*+Vertex, +Graph, -Vertices***)**

Unify *Vertices* with the list of neighbours of vertex *Vertex* in *Graph*. Example:

```
?- neighbours(4,[1-[3,5],2-[4],3-[],
                 4-[1,2,7,5],5-[],6-[],7-[],8-[]], NL).
NL = [1,2,7,5]
```

neighbors(+*Vertex*, +*Graph*, -*Vertices***)**

American version of `neighbours/3`.

complement(+*Graph*, -*NewGraph***)**

Unify *NewGraph* with the graph complementary to *Graph*. Example:

```
?- complement([1-[3,5],2-[4],3-[],
               4-[1,2,7,5],5-[],6-[],7-[],8-[]], NL).
NL = [1-[2,4,6,7,8],2-[1,3,5,6,7,8],3-[1,2,4,5,6,7,8],
      4-[3,5,6,8],5-[1,2,3,4,6,7,8],6-[1,2,3,4,5,7,8],
      7-[1,2,3,4,5,6,8],8-[1,2,3,4,5,6,7]]
```

compose(+*LeftGraph*, +*RightGraph*, -*NewGraph***)**

Compose *NewGraph* by connecting the *drains* of *LeftGraph* to the *sources* of *RightGraph*. Example:

```
?- compose([1-[2],2-[3]],[2-[4],3-[1,2,4]],L).
L = [1-[4], 2-[1,2,4], 3-[]]
```

ugraph_union(+*Graph1*, +*Graph2*, -*NewGraph***)**

NewGraph is the union of *Graph1* and *Graph2*. Example:

```
?- ugraph_union([1-[2],2-[3]],[2-[4],3-[1,2,4]],L).
L = [1-[2], 2-[3,4], 3-[1,2,4]]
```

top_sort(+*Graph*, -*Sort***)**

Generate the set of nodes *Sort* as a topological sorting of *Graph*, if one is possible. A toplogical sort is possible if the graph is connected and acyclic. In the example we show how topological sorting works for a linear graph:

```
?- top_sort([1-[2], 2-[3], 3-[]], L).
L = [1, 2, 3]
```

top_sort(+*Graph*, -*Sort0*, -*Sort***)**

Generate the difference list Sort-Sort0 as a topological sorting of *Graph*, if one is possible.

transitive_closure(+*Graph*, -*Closure***)**

Generate the graph Closure as the transitive closure of *Graph*. Example:

```
 ?- transitive_closure([1-[2,3],2-[4,5],4-[6]],L).
L = [1-[2,3,4,5,6], 2-[4,5,6], 4-[6]]
```

reachable(+*Vertex*, +*Graph*, -*Vertices***)**
Unify *Vertices* with the set of all vertices in *Graph* that are reachable from *Vertex*. Example:

```
?- reachable(1,[1-[3,5],2-[4],3-[],4-[5],5-[]],V).
V = [1, 3, 5]
```

A.29 library(url): Analysing and constructing URL

author
- Jan Wielemaker
- Lukas Faulstich

deprecated New code should use library(uri), provided by the `clib` package.

This library deals with the analysis and construction of a URL, Universal Resource Locator. URL is the basis for communicating locations of resources (data) on the web. A URL consists of a protocol identifier (e.g. HTTP, FTP, and a protocol-specific syntax further defining the location. URLs are standardized in RFC-1738.

The implementation in this library covers only a small portion of the defined protocols. Though the initial implementation followed RFC-1738 strictly, the current is more relaxed to deal with frequent violations of the standard encountered in practical use.

global_url(+*URL*, +*Base*, -*Global***)** *[det]*
Translate a possibly relative *URL* into an absolute one.

Errors syntax_error(illegal_url) if *URL* is not legal.

is_absolute_url(+*URL***)**
True if *URL* is an absolute *URL*. That is, a *URL* that starts with a protocol identifier.

http_location(?*Parts*, ?*Location***)**
Construct or analyze an HTTP location. This is similar to `parse_url/2`, but only deals with the location part of an HTTP URL. That is, the path, search and fragment specifiers. In the HTTP protocol, the first line of a message is

```
<Action> <Location> HTTP/<version>
```

Parameters	
Location	Atom or list of character codes.

parse_url(+*URL*, -*Attributes***)** *[det]*
Construct or analyse a *URL*. *URL* is an atom holding a *URL* or a variable. Parts is a list of components. Each component is of the format Name(Value). Defined components are:

protocol(*Protocol***)**
The used protocol. This is, after the optional `url:`, an identifier separated from the remainder of the *URL* using :. `parse_url/2` assumes the `http` protocol if no protocol is specified and the *URL* can be parsed as a valid HTTP url. In addition to the RFC-1738 specified protocols, the `file` protocol is supported as well.

host(*Host***)**
Host-name or IP-address on which the resource is located. Supported by all network-based protocols.

port(*Port***)**
Integer port-number to access on the \arg{Host}. This only appears if the port is explicitly specified in the *URL*. Implicit default ports (e.g., 80 for HTTP) do *not* appear in the part-list.

path(*Path***)**
(File-) path addressed by the *URL*. This is supported for the `ftp`, `http` and `file` protocols. If no path appears, the library generates the path `/`.

search(*ListOfNameValue***)**
Search-specification of HTTP *URL*. This is the part after the `?`, normally used to transfer data from HTML forms that use the `GET` protocol. In the *URL* it consists of a www-form-encoded list of Name=Value pairs. This is mapped to a list of Prolog Name=Value terms with decoded names and values.

fragment(*Fragment***)**
Fragment specification of HTTP *URL*. This is the part after the `#` character.

The example below illustrates the all this for an HTTP *URL*.

```
?- parse_url('http://swi.psy.uva.nl/message.cgi?msg=Hello+World%21#x', P).

P = [ protocol(http),
      host('swi.psy.uva.nl'),
      fragment(x),
      search([ msg = 'Hello World!'
             ]),
      path('/message.cgi')
    ]
```

By instantiating the parts-list this predicate can be used to create a *URL*.

parse_url(*+URL, +BaseURL, -Attributes***)** *[det]*
Similar to `parse_url/2` for relative URLs. If *URL* is relative, it is resolved using the absolute *URL BaseURL*.

www_form_encode(*+Value, -XWWWFormEncoded***)** *[det]*
www_form_encode(*-Value, +XWWWFormEncoded***)** *[det]*
En/decode to/from application/x-www-form-encoded. Encoding encodes all characters except RFC 3986 *unreserved* (ASCII `alnum` (see `code_type/2`)), and one of ”-._ ” using percent

encoding. Newline is mapped to `%OD%OA`. When decoding, newlines appear as a single newline (10) character.

Note that a space is encoded as `%20` instead of `+`. Decoding decodes both to a space.

deprecated Use `uri_encoded/3` for new code.

set_url_encoding(*?Old, +New***)** *[semidet]*

Query and set the encoding for URLs. The default is `utf8`. The only other defined value is `iso_latin_1`.

To be done Having a global flag is highly inconvenient, but a work-around for old sites using ISO Latin 1 encoding.

url_iri(*+Encoded, -Decoded***)** *[det]*
url_iri(*-Encoded, +Decoded***)** *[det]*

Convert between a URL, encoding in US-ASCII and an IRI. An IRI is a fully expanded Unicode string. Unicode strings are first encoded into UTF-8, after which %-encoding takes place.

parse_url_search(*?Spec, ?Fields:list(Name=Value)***)** *[det]*

Construct or analyze an HTTP search specification. This deals with form data using the MIME-type =application/x-www-form-urlencoded= as used in HTTP GET requests.

file_name_to_url(*+File, -URL***)** *[det]*
file_name_to_url(*-File, +URL***)** *[semidet]*

Translate between a filename and a file:`//` *URL*.

To be done Current implementation does not deal with paths that need special encoding.

A.30 library(varnumbers): Utilities for numbered terms

See also `numbervars/4`, `=@=/2` (`variant/2`).

Compatibility This library was introduced by Quintus and available in many related implementations, although not with exactly the same set of predicates.

This library provides the inverse functionality of the built-in `numbervars/3`. Note that this library suffers from the known issues that '$VAR'(X) is a normal Prolog term and, -unlike the built-in numbervars-, the inverse predicates do *not* process cyclic terms. The following predicate is true for any acyclic term that contains no '$VAR'(X), integer(X) terms and no constraint variables:

```
always_true(X) :-
      copy_term(X, X2),
      numbervars(X),
      varnumbers(X, Copy),
      Copy =@= X2.
```

numbervars(+*Term***)** *[det]*

Number variables in *Term* using $VAR(N). Equivalent to numbervars(*Term*, 0, _).

See also `numbervars/3`, `numbervars/4`

varnumbers(+*Term*, -*Copy***)** *[det]*

Inverse of `numbervars/1`. Equivalent to varnumbers(*Term*, 0, *Copy*).

varnumbers(+*Term*, +*Start*, -*Copy***)** *[det]*

Inverse of `numbervars/3`. True when copy is a copy of *Term* with all variables numbered >= *Start* consistently replaced by fresh variables. Variables in *Term* are *shared* with *Copy* rather than replaced by fresh variables.

Errors domain_error(acyclic_term, *Term*) if *Term* is cyclic.
Compatibility Quintus, SICStus. Not in YAP version of this library

max_var_number(+*Term*, +*Start*, -*Max***)** *[det]*

True when *Max* is the max of *Start* and the highest numbered $VAR(N) term.

author Vitor Santos Costa
Compatibility YAP

Hackers corner B

This appendix describes a number of predicates which enable the Prolog user to inspect the Prolog environment and manipulate (or even redefine) the debugger. They can be used as entry points for experiments with debugging tools for Prolog. The predicates described here should be handled with some care as it is easy to corrupt the consistency of the Prolog system by misusing them.

B.1 Examining the Environment Stack

prolog_current_frame(*-Frame***)**

Unify *Frame* with an integer providing a reference to the parent of the current local stack frame. A pointer to the current local frame cannot be provided as the predicate succeeds deterministically and therefore its frame is destroyed immediately after succeeding.

prolog_frame_attribute(*+Frame, +Key, -Value***)**

Obtain information about the local stack frame *Frame*. *Frame* is a frame reference as obtained through `prolog_current_frame/1`, `prolog_trace_interception/4` or this predicate. The key values are described below.

alternative

Value is unified with an integer reference to the local stack frame in which execution is resumed if the goal associated with *Frame* fails. Fails if the frame has no alternative frame.

has_alternatives

Value is unified with `true` if *Frame* still is a candidate for backtracking; `false` otherwise.

goal

Value is unified with the goal associated with *Frame*. If the definition module of the active predicate is not `user` the goal is represented as $\langle module \rangle : \langle goal \rangle$. Do not instantiate variables in this goal unless you **know** what you are doing! Note that the returned term may contain references to the frame and should be discarded before the frame terminates.[1]

parent_goal

If *Value* is instantiated to a callable term, find a frame executing the predicate described by *Value* and unify the arguments of *Value* to the goal arguments associated with the frame. This is intended to check the current execution context. The user must ensure the checked parent goal is not removed from the stack due to last-call optimisation and be aware of the slow operation on deeply nested calls.

---

[1] The returned term is actually an illegal Prolog term that may hold references from the global- to the local stack to preserve the variable names.

predicate_indicator
Similar to `goal`, but only returning the [⟨*module*⟩:]⟨*name*⟩/⟨*arity*⟩ term describing the term, not the actual arguments. It avoids creating an illegal term as `goal` and is used by the library `prolog_stack`.

clause
Value is unified with a reference to the currently running clause. Fails if the current goal is associated with a foreign (C) defined predicate. See also `nth_clause/3` and `clause_property/2`.

level
Value is unified with the recursion level of *Frame*. The top level frame is at level '0'.

parent
Value is unified with an integer reference to the parent local stack frame of *Frame*. Fails if *Frame* is the top frame.

context_module
Value is unified with the name of the context module of the environment.

top
Value is unified with `true` if *Frame* is the top Prolog goal from a recursive call back from the foreign language. `false` otherwise.

hidden
Value is unified with `true` if the frame is hidden from the user, either because a parent has the hide-childs attribute (all system predicates), or the system has no trace-me attribute.

skipped
Value is `true` if this frame was skipped in the debugger.

pc
Value is unified with the program-pointer saved on behalf of the parent-goal if the parent-goal is not owned by a foreign predicate.

argument(*N*)
Value is unified with the *N*-th slot of the frame. Argument 1 is the first argument of the goal. Arguments above the arity refer to local variables. Fails silently if *N* is out of range.

prolog_choice_attribute(*+ChoicePoint, +Key, -Value***)**
Extract attributes of a choice-point. *ChoicePoint* is a reference to a choice-point as passed to `prolog_trace_interception/4` on the 3-th argument. *Key* specifies the requested information:

parent
Requests a reference to the first older choice-point.

frame
Requests a reference to the frame to which the choice-point refers.

type
Requests the type. Defined values are `clause` (the goal has alternative clauses), `foreign` (non-deterministic foreign predicate), `jump` (clause internal choice-point), `top` (first dummy choice-point), `catch` (`catch/3` to allow for undo), `debug` (help the debugger), or `none` (has been deleted).

This predicate is used for the graphical debugger to show the choice-point stack.

deterministic(*-Boolean*)

Unifies its argument with `true` if no choicepoint exists that is more recent than the entry of the clause in which it appears. There are few realistic situations for using this predicate. It is used by the `prolog/0` toplevel to check whether Prolog should prompt the user for alternatives. Similar results can be achieved in a more portable fashion using `call_cleanup/2`.

B.2 Intercepting the Tracer

prolog_trace_interception(*+Port, +Frame, +Choice, -Action*)

Dynamic predicate, normally not defined. This predicate is called from the SWI-Prolog debugger just before it would show a port. If this predicate succeeds the debugger assumes the trace action has been taken care of and continues execution as described by *Action*. Otherwise the normal Prolog debugger actions are performed.

Port denotes the reason to activate the tracer ('port' in the 4/5-port, but with some additions):

call

Normal entry through the call-port of the 4-port debugger.

redo

Normal entry through the call-port of the 4-port debugger. The `redo` port signals resuming a predicate to generate alternative solutions.

unify

The unify-port represents the *neck* instruction, signalling the end of the head-matching process. This port is normally invisible. See `leash/1` and `visible/1`.

exit

The exit-port signals the goal is proved. It is possible for the goal to have alternatives. See `prolog_frame_attribute/3` to examine the goal-stack.

fail

The fail-port signals final failure of the goal.

exception(*Except*)

An exception is raised and still pending. This port is activated on each parent frame of the frame generating the exception until the exception is caught or the user restarts normal computation using `retry`. *Except* is the pending exception term.

break(*PC*)

A `break` instruction is executed. *PC* is program counter. This port is used by the graphical debugger.

cut_call(*PC*)

A cut is encountered at *PC*. This port is used by the graphical debugger to visualise the effect of the cut.

cut_exit(*PC*)

A cut has been executed. See `cut_call`(*PC*) for more information.

Frame is a reference to the current local stack frame, which can be examined using `prolog_frame_attribute/3`. *Choice* is a reference to the last choice-point and can be examined using `prolog_choice_attribute/3`. *Action* must be unified with a term that specifies how execution must continue. The following actions are defined:

abort
Abort execution. See `abort/0`.

continue
Continue (i.e., *creep* in the commandline debugger).

fail
Make the current goal fail.

ignore
Step over the current goal without executing it.

nodebug
Continue execution in normal nodebugging mode. See `nodebug/0`.

retry
Retry the current frame.

retry(*Frame***)**
Retry the given frame. This must be a parent of the current frame.

skip
Skip over the current goal (i.e., *skip* in the commandline debugger).

Together with the predicates described in section 4.37 and the other predicates of this chapter this predicate enables the Prolog user to define a complete new debugger in Prolog. Besides this, it enables the Prolog programmer to monitor the execution of a program. The example below records all goals trapped by the tracer in the database.

```
prolog_trace_interception(Port, Frame, _PC, continue) :-
        prolog_frame_attribute(Frame, goal, Goal),
        prolog_frame_attribute(Frame, level, Level),
        recordz(trace, trace(Port, Level, Goal)).
```

To trace the execution of 'go' this way the following query should be given:

```
?- trace, go, notrace.
```

prolog_skip_frame(*-Frame***)**
Indicate *Frame* as a skipped frame and set the 'skip level' (see `prolog_skip_level/2` to the recursion depth of *Frame*. The effect of the skipped flag is that a redo on a child of this frame is handled differently. First, a `redo` trace is called for the child, where the skip-level is set to `redo_in_skip`. Next, the skip level is set to skip-level of the skipped frame.

prolog_skip_level(*-Old, +New***)**
Unify *Old* with the old value of 'skip level' and then set this level according to *New*. *New* is an integer, the atom `very_deep` (meaning don't skip) or the atom `skip_in_redo` (see `prolog_skip_frame/1`). The 'skip level' is a setting of each Prolog thread that disables the debugger on all recursion levels deeper than the level of the variable. See also `prolog_skip_frame/1`.

B.3 Adding context to errors: prolog_exception_hook

The hook `prolog_exception_hook/4` has been introduced in SWI-Prolog 5.6.5 to provide dedicated exception handling facilities for application frameworks, for example non-interactive server applications that wish to provide extensive context for exceptions for offline debugging.

prolog_exception_hook(+*ExceptionIn, -ExceptionOut, +Frame, +CatcherFrame***)**

This hook predicate, if defined in the module `user`, is between raising an exception and handling it. It is intended to allow a program adding additional context to an exception to simplify diagnosing the problem. *ExceptionIn* is the exception term as raised by `throw/1` or one of the built-in predicates. The output argument *ExceptionOut* describes the exception that is actually raised. *Frame* is the innermost frame. See `prolog_frame_attribute/3` and the library `prolog_stack` for getting information from this. *CatcherFrame* is a reference to the frame calling the matching `catch/3` or `none` if the exception is not caught.

The hook is run in 'nodebug' mode. If it succeeds, *ExceptionOut* is considered the current exception. If it fails, *ExceptionIn* is used for further processing. The hook is *never* called recursively. The hook is *not* allowed to modify *ExceptionOut* in such a way that it no longer unifies with the catching frame.

Typically, `prolog_exception_hook/4` is used to fill the second argument of `error`(*Formal, Context*) exceptions. *Formal* is defined by the ISO standard, while SWI-Prolog defines *Context* as a term `context`(*Location, Message*). *Location* is bound to a term ⟨*name*⟩/⟨*arity*⟩ by the kernel. This hook can be used to add more information on the calling context, such as a full stack trace.

Applications that use exceptions as part of normal processing must do a quick test of the environment before starting expensive gathering information on the state of the program.

The hook can call `trace/0` to enter trace mode immediately. For example imagine an application performing an unwanted division by zero while all other errors are expected and handled. We can force the debugger using the hook definition below. Run the program in debug mode (see `debug/0`) to preserve as much as possible of the error context.

```
user:prolog_exception_hook(error(evaluation_error(zero_divisor), _),
                          _, _, _) :-
        trace, fail.
```

B.4 Hooks using the exception predicate

This section describes the predicate `exception/3`, which can be defined by the user in the module `user` as a multifile predicate. Unlike the name suggests, this is actually a *hook* predicate that has no relation to Prolog exceptions as defined by the ISO predicates `catch/3` and `throw/1`.

The predicate `exception/3` is called by the kernel on a couple of events, allowing the user to 'fix' errors just-in-time. The mechanism allows for *lazy* creation of objects such as predicates.

exception(+*Exception, +Context, -Action***)**

Dynamic predicate, normally not defined. Called by the Prolog system on run-time exceptions that can be repaired 'just-in-time'. The values for *Exception* are described below. See also `catch/3` and `throw/1`.

If this hook predicate succeeds it must instantiate the *Action* argument to the atom `fail` to make the operation fail silently, `retry` to tell Prolog to retry the operation or `error` to make the system generate an exception. The action `retry` only makes sense if this hook modified the environment such that the operation can now succeed without error.

undefined_predicate

Context is instantiated to a predicate-indicator ([module]:⟨*name*⟩/⟨*arity*⟩). If the predicate fails Prolog will generate an `existence_error` exception. The hook is intended to implement alternatives to the built-in autoloader, such as autoloading code from a database. Do *not* use this hook to suppress existence errors on predicates. See also `unknown` and section 2.13.

undefined_global_variable

Context is instantiated to the name of the missing global variable. The hook must call `nb_setval/2` or `b_setval/2` before returning with the action `retry`.

B.5 Hooks for integrating libraries

Some libraries realise an entirely new programming paradigm on top of Prolog. An example is XPCE which adds an object-system to Prolog as well as an extensive set of graphical primitives. SWI-Prolog provides several hooks to improve the integration of such libraries. See also section 4.5 for editing hooks and section 4.10.3 for hooking into the message system.

prolog_list_goal(*:Goal***)**

Hook, normally not defined. This hook is called by the 'L' command of the tracer in the module `user` to list the currently called predicate. This hook may be defined to list only relevant clauses of the indicated *Goal* and/or show the actual source code in an editor. See also `portray/1` and `multifile/1`.

prolog:debug_control_hook(*:Action***)**

Hook for the debugger-control predicates that allows the creator of more high-level programming languages to use the common front-end predicates to control the debugger. For example, XPCE uses these hooks to allow for spying methods rather than predicates. *Action* is one of:

spy(*Spec***)**

Hook in `spy/1`. If the hook succeeds `spy/1` takes no further action.

nospy(*Spec***)**

Hook in `nospy/1`. If the hook succeeds `nospy/1` takes no further action. If `spy/1` is hooked, it is advised to place a complementary hook for `nospy/1`.

nospyall

Hook in `nospyall/0`. Should remove all spy-points. This hook is called in a failure-driven loop.

debugging

Hook in `debugging/0`. It can be used in two ways. It can report the status of the additional debug points controlled by the above hooks and fail to let the system report the others, or it succeeds, overruling the entire behaviour of `debugging/0`.

prolog:help_hook(+*Action***)**

Hook into `help/0` and `help/1`. If the hook succeeds, the built-in actions are not executed. For example, `?- help(picture).` is caught by the XPCE help-hook to give help on the class *picture*. Defined actions are:

help

User entered plain `help/0` to give default help. The default performs `help(help/1)`, giving help on help.

help(*What***)**

Hook in `help/1` on the topic *What*.

apropos(*What***)**

Hook in `apropos/1` on the topic *What*.

B.6 Hooks for loading files

All loading of source files is achieved by `load_files/2`. The hook `prolog_load_file/2` can be used to load Prolog code from non-files or even load entirely different information, such as foreign files.

prolog_load_file(+*Spec,* +*Options***)**

Load a single object. If this call succeeds, `load_files/2` assumes the action has been taken care of. This hook is only called if *Options* does not contain the `stream`(*Input*) option. The hook must be defined in the module `user`.

The `http_load` provides an example, loading Prolog sources directly from an HTTP server.

prolog:comment_hook(+*Comments,* +*Pos,* +*Term***)**

This hook allows for processing —structured— comments encountered by the compiler. The reader collects all comments found from the current position to the end of the next term. It calls this hook providing a list of *Position-Comment* in *Comments*, the start position of the next term in *Pos*, and the next term itself in *Term*. All positions are stream position terms. This hook is exploited by the documentation system. See `stream_position_data/3`. See also `read_term/3`.

B.7 Readline Interaction

The following predicates are available if SWI-Prolog is linked to the GNU readline library. This is by default the case on non-Windows installations and indicated by the Prolog flag `readline`.[2] See also `readline(3)`.

rl_read_init_file(+*File***)**

Read a readline initialisation file. Readline by default reads `~/.inputrc`. This predicate may be used to read alternative readline initialisation files.

rl_add_history(+*Line***)**

Add a line to the Control-P/Control-N history system of the readline library.

---

[2] `swipl-win.exe` uses its own history system and does *not* support these predicates.

rl_write_history(+*FileName***)**
Write current history to *FileName*. Can be used from `at_halt/1` to save the history.

rl_read_history(+*FileName***)**
Read history from *FileName*, appending to the current history.

C

Compatibility with other Prolog dialects

This chapter explains issues for writing portable Prolog programs. It was started after discussion with Vitor Santos Costa, the leading developer of YAP Prolog[1] YAP and SWI-Prolog have expressed the ambition to enhance the portability beyond the trivial Prolog examples, including complex libraries involving foreign code.

Although it is our aim to enhance compatibility, we are still faced with many incompatibilities between the dialects. As a first step both YAP and SWI will provide some instruments that help developing portable code. A first release of these tools appeared in SWI-Prolog 5.6.43. Some of the facilities are implemented in the base system, others in the library `dialect.pl`.

- The Prolog flag `dialect` is an unambiguous and fast way to find out which Prolog dialect executes your program. It has the value `swi` for SWI-Prolog and `yap` on YAP.
- The Prolog flag `version_data` is bound to a term `swi`(*Major, Minor, Patch, Extra*)
- Conditional compilation using `:- if(Condition) ...:- endif` is supported. See section 4.3.1.
- The predicate `expects_dialect/1` allows for specifying for which Prolog system the code was written.
- The predicates `exists_source/1` and `source_exports/2` can be used to query the library content. The `require/1` directive can be used to get access to predicates without knowing their location.
- The module predicates `use_module/1`, `use_module/2` have been extended with a notion for 'import-except' and 'import-as'. This is particularly useful together with `reexport/1` and `reexport/2` to compose modules from other modules and mapping names.
- Foreign code can expect `__SWI_PROLOG__` when compiled for SWI-Prolog and `__YAP_PROLOG__` when compiled on YAP.

:- expects_dialect(+*Dialect***)**

This directive states that the code following the directive is written for the given Prolog *Dialect*. See also `dialect`. The declaration holds until the end of the file in which it appears. The current dialect is available using `prolog_load_context/2`.

The exact behaviour of this predicate is still subject to discussion. Of course, if *Dialect* matches the running dialect the directive has no effect. Otherwise we check for the existence of `library`(*dialect/Dialect*) and load it if the file is found. Currently, this file has this functionality:

---

[1] `http://yap.sourceforge.net/`

- Define system predicates of the requested dialect we do not have.
- Apply `goal_expansion/2` rules that map conflicting predicates to versions emulating the requested dialect. These expansion rules reside in the dialect compatibility module, but are applied if prolog_load_context(dialect, Dialect) is active.
- Modify the search path for library directories, putting libraries compatible with the target dialect before the native libraries.
- Setup support for the default filename extension of the dialect.

exists_source(+*Spec***)**

Is true if *Spec* exists as a Prolog source. *Spec* uses the same conventions as `load_files/2`. Fails without error if *Spec* cannot be found.

source_exports(+*Spec*, +*Export***)**

Is true if source *Spec* exports *Export*, a predicate indicator. Fails without error otherwise.

C.1 Some considerations for writing portable code

The traditional way to write portable code is to define custom predicates for all potentially non-portable code and define these separately for all Prolog dialects one wishes to support. Here are some considerations.

- Probably the best reason for this is that it allows to define minimal semantics required by the application for the portability predicates. Such functionality can often be mapped efficiently to the target dialect. Contrary, if code was written for dialect X, the defined semantics are those of dialect X. Emulating all extreme cases and full error handling compatibility may be tedious and result in a much slower implementation that needed. Take for example `call_cleanup/2`. The SICStus definition is fundamentally different from the SWI definition, but 99% of the applications just want to make calls like below to guarantee *StreamIn* is closed, even if `process/1` misbehaves.

```
        call_cleanup(process(StreamIn), close(In))
```

- As a drawback, the code becomes full of *my_call_cleanup*, etc. and every potential portability conflict needs to be abstracted. It is hard for people who have to maintain such code later to grasp the exact semantics of the *my_** predicates and applications that combine multiple libraries using this compatibility approach are likely to encounter conflicts between the portability layers. A good start is not to use *my_**, but a prefix derived from the library or application name or names that explain the intended semantics more precisely.

- Another problem is that most code is initially not written with portability in mind. Instead, ports are requested by users or arise from the desire to switch Prolog dialect. Typically, we want to achieve compatibility with the new Prolog dialect with minimal changes, often keeping compatibility with the original dialect(s). This problem is well known from the C/Unix world and we advise anyone to study the philosophy of GNU autoconf, from which we will illustrate some highlights below.

The GNU autoconf suite, known to most people as `configure`, was an answer to the frustrating life of Unix/C programmers when Unix dialects were about as abundant and poorly standardised as Prolog dialects today. Writing a portable C program can only be achieved using cpp, the C preprocessor. The C preprocessor performs two tasks: macro expansion and conditional compilation. Prolog realises macro expansion through `term_expansion/2` and `goal_expansion/2`. Conditional compilation is achieved using `:- if(Condition)` as explained in section 4.3.1. The situation appears similar.

The important lesson learned from GNU autoconf is that the *last* resort for conditional compilation to achieve portability is to switch on the platform or dialect. Instead, GNU autoconf allows you to write tests for specific properties of the platform. Most of these are whether or not some function or file is available. Then there are some standard tests for difficult-to-write-portable situations and finally there is a framework that allows you to write arbitrary C programs and check whether they can be compiled and/or whether they show the intended behaviour. Using a separate `configure` program is needed in C, as you cannot perform C compilation step or run C programs from the C preprocessor. In most Prolog environments we do not need this distinction as the compiler is integrated into the runtime environment and Prolog has excellent reflexion capabilities.

We must learn from the distinction to test for features instead of platform (dialect), as this makes the platform-specific code robust for future changes of the dialect. Suppose we need `compare/3` as defined in this manual. The `compare/3` predicate is not part of the ISO standard, but many systems support it and it is not unlikely it will become ISO standard or the intended dialect will start supporting it. GNU autoconf strongly advises to test for the availability:

```
:- if(\+current_predicate(_, compare(_,_,_))).
compare(<, Term1, Term2) :-
        Term1 @< Term2, !.
compare(>, Term1, Term2) :-
        Term1 @> Term2, !.
compare(=, Term1, Term2) :-
        Term1 == Term2.
:- endif.
```

This code is **much** more robust against changes to the intended dialect and, possibly at least as important, will provide compatibility with dialects you didn't even consider porting to right now.

In a more challenging case, the target Prolog has `compare/3`, but the semantics are different. What to do? One option is to write a `my_compare/3` and change all occurrences in the code. Alternatively you can rename calls using `goal_expansion/2` like below. This construct will not only deal with Prolog dialects lacking `compare/3` as well as those that only implement it for numeric comparison or have changed the argument order. Of course, writing rock-solid code would require a complete test-suite, but this example will probably cover all Prolog dialects that allow for conditional compilation, have core ISO facilities and provide `goal_expansion/2`, the things we claim a Prolog dialect should have to start writing portable code for it.

```
:- if(\+catch(compare(<,a,b), _, fail)).
compare_standard_order(<, Term1, Term2) :-
        Term1 @< Term2, !.
compare_standard_order(>, Term1, Term2) :-
        Term1 @> Term2, !.
```

```
compare_standard_order(=, Term1, Term2) :-
        Term1 == Term2.

goal_expansion(compare(Order, Term1, Term2),
               compare_standard_order(Order, Term1, Term2)).
:- endif.
```

D

Glossary of Terms

anonymous [variable]

The variable _ is called the *anonymous* variable. Multiple occurrences of _ in a single *term* are not *shared*.

arguments

Arguments are *terms* that appear in a *compound term*. *A1* and *a2* are the first and second argument of the term `myterm`(*A1, a2*).

arity

Argument count (= number of arguments) of a *compound term*.

assert

Add a *clause* to a *predicate*. Clauses can be added at either end of the clause-list of a *predicate*. See `asserta/1` and `assertz/1`.

atom

Textual constant. Used as name for *compound* terms, to represent constants or text.

backtracking

Search process used by Prolog. If a predicate offers multiple *clauses* to solve a *goal*, they are tried one-by-one until one *succeeds*. If a subsequent part of the proof is not satisfied with the resulting *variable binding*, it may ask for an alternative *solution* (= *binding* of the *variables*), causing Prolog to reject the previously chosen *clause* and try the next one.

binding [of a variable]

Current value of the *variable*. See also *backtracking* and *query*.

built-in [predicate]

Predicate that is part of the Prolog system. Built-in predicates cannot be redefined by the user, unless this is overruled using `redefine_system_predicate/1`.

body

Part of a *clause* behind the *neck* operator (`:-`).

clause

'Sentence' of a Prolog program. A *clause* consists of a *head* and *body* separated by the *neck* operator (`:-`) or it is a *fact*. For example:

```
parent(X) :-
        father(X, _).
```

Expressed as “X is a parent if X is a father of someone”. See also *variable* and *predicate*.

compile

Process where a Prolog *program* is translated to a sequence of instructions. See also *interpreted*. SWI-Prolog always compiles your program before executing it.

compound [term]

Also called *structure*. It consists of a name followed by *N arguments*, each of which are *terms*. *N* is called the *arity* of the term.

context module

If a *term* is referring to a *predicate* in a *module*, the *context module* is used to find the target module. The context module of a *goal* is the module in which the *predicate* is defined, unless this *predicate* is *module transparent*, in which case the *context module* is inherited from the parent *goal*. See also `module_transparent/1` and *meta-predicate*.

dynamic [predicate]

A *dynamic* predicate is a predicate to which *clauses* may be *assert*ed and from which *clauses* may be *retract*ed while the program is running. See also *update view*.

exported [predicate]

A *predicate* is said to be *exported* from a *module* if it appears in the *public list*. This implies that the predicate can be *imported* into another module to make it visible there. See also `use_module/[1,2]`.

fact

Clause without a *body*. This is called a fact because, interpreted as logic, there is no condition to be satisfied. The example below states `john` is a person.

```
person(john).
```

fail

A *goal* is said to haved failed if it could not be *proven*.

float

Computer’s crippled representation of a real number. Represented as ‘IEEE double’.

foreign

Computer code expressed in languages other than Prolog. SWI-Prolog can only cooperate directly with the C and C++ computer languages.

functor

Combination of name and *arity* of a *compound* term. The term `foo`(*a, b, c*) is said to be a term belonging to the functor `foo/3`. `foo/0` is used to refer to the *atom* `foo`.

goal

Question stated to the Prolog engine. A *goal* is either an *atom* or a *compound* term. A *goal* either succeeds, in which case the *variables* in the *compound* terms have a *binding*, or it *fails* if Prolog fails to prove it.

hashing

Indexing technique used for quick lookup.

head

Part of a *clause* before the *neck* operator (`:-`). This is an *atom* or *compound* term.

imported [predicate]

A *predicate* is said to be *imported* into a *module* if it is defined in another *module* and made available in this *module*. See also chapter 5.

indexing

Indexing is a technique used to quickly select candidate *clauses* of a *predicate* for a specific *goal*. In most Prolog systems, indexing is done (only) on the first *argument* of the *head*. If this argument is instantiated to an *atom*, *integer*, *float* or *compound* term with *functor*, *hashing* is used to quickly select all *clauses* where the first argument may *unify* with the first argument of the *goal*. SWI-Prolog supports just-in-time and multi-argument indexing. See section 2.17.

integer

Whole number. On all implementations of SWI-Prolog integers are at least 64-bit signed values. When linked to the GNU GMP library, integer arithmetic is unbounded. See also `current_prolog_flag/2`, flags `bounded`, `max_integer` and `min_integer`.

interpreted

As opposed to *compiled*, interpreted means the Prolog system attempts to prove a *goal* by directly reading the *clauses* rather than executing instructions from an (abstract) instruction set that is not or only indirectly related to Prolog.

meta-predicate

A *predicate* that reasons about other *predicates*, either by calling them, (re)defining them or querying *properties*.

module

Collection of predicates. Each module defines a name-space for predicates. *built-in* predicates are accessible from all modules. Predicates can be published (*exported*) and *imported* to make their definition available to other modules.

module transparent [predicate]

A *predicate* that does not change the *context module*. Sometimes also called a *meta-predicate*.

multifile [predicate]

Predicate for which the definition is distributed over multiple source files. See `multifile/1`.

neck

Operator (`:-`) separating *head* from *body* in a *clause*.

operator

Symbol (*atom*) that may be placed before its *operand* (prefix), after its *operand* (postfix) or between its two *operands* (infix).

In Prolog, the expression `a+b` is exactly the same as the canonical term `+(a,b)`.

operand

Argument of an *operator*.

precedence

The *priority* of an *operator*. Operator precedence is used to interpret `a+b*c` as `+(a, *(b,c))`.

predicate

Collection of *clauses* with the same *functor* (name/*arity*). If a *goal* is proved, the system looks for a *predicate* with the same functor, then uses *indexing* to select candidate *clauses* and then tries these *clauses* one-by-one. See also *backtracking*.

predicate indicator

Term of the form Name/Arity (traditional) or Name//Arity (ISO DCG proposal), where Name is an atom and Arity a non-negative integer. It acts as an *indicator* (or reference) to a predicate or *DCG* rule.

priority

In the context of *operators* a synonym for *precedence*.

program

Collection of *predicates*.

property

Attribute of an object. SWI-Prolog defines various **_property* predicates to query the status of predicates, clauses. etc.

prove

Process where Prolog attempts to prove a *query* using the available *predicates*.

public list

List of *predicates* exported from a *module*.

query

See *goal*.

retract

Remove a *clause* from a *predicate*. See also *dynamic*, *update view* and *assert*.

shared

Two *variables* are called *shared* after they are *unified*. This implies if either of them is *bound*, the other is bound to the same value:

```
?- A = B, A = a.

A = a,
B = a
```

singleton [variable]

Variable appearing only one time in a *clause*. SWI-Prolog normally warns for this to avoid you making spelling mistakes. If a variable appears on purpose only once in a clause, write it as _ (see *anonymous*). Rules for naming a variable and avoiding a warning are given in section 2.15.1.

solution

Bindings resulting from a successfully *prove*n *goal*.

structure

Synonym for *compound* term.

string

Used for the following representations of text: a packed array (see section 4.23, SWI-Prolog specific), a list of character codes or a list of one-character *atoms*.

succeed

A *goal* is said to have *succeeded* if it has been *proven*.

term

Value in Prolog. A *term* is either a *variable*, *atom*, *integer*, *float* or *compound* term. In addition, SWI-Prolog also defines the type *string*.

transparent

See *module transparent*.

unify

Prolog process to make two terms equal by assigning variables in one term to values at the corresponding location of the other term. For example:

```
?- foo(a, B) = foo(A, b).

A = a,
B = b
```

Unlike assignment (which does not exist in Prolog), unification is not directed.

update view

How Prolog behaves when a *dynamic predicate* is changed while it is running. There are two models. In most older Prolog systems the change becomes immediately visible to the *goal*, in modern systems including SWI-Prolog, the running *goal* is not affected. Only new *goals* ‘see’ the new definition.

variable

A Prolog variable is a value that ‘is not yet bound’. After *binding* a variable, it cannot be modified. *Backtracking* to a point in the execution before the variable was bound will turn it back into a variable:

```
?- A = b, A = c.
No
?- (A = b; true; A = c).
A = b ;
A = _G283 ;
A = c ;
No
```

See also *unify*.

SWI-Prolog License Conditions and Tools

E

SWI-Prolog licensing aims at a large audience, combining ideas from the Free Software Foundation and the less principal Open Source Initiative. The license aims at the following:

- Make SWI-Prolog and its libraries 'as free as possible'.
- Allow for easy integration of contributions. See section E.2.
- Free software can build on SWI-Prolog without limitations.
- Non-free (open or proprietary) software can be produced using SWI-Prolog, although contributed pure GPL-ed components cannot be used.

To achieve this, different parts of the system have different licenses. SWI-Prolog programs consist of a mixture of 'native' code (source compiled to machine instructions) and 'virtual machine' code (Prolog source compiled to SWI-Prolog virtual machine instructions, covering both compiled SWI-Prolog libraries and your compiled application).

For maximal coherence between free licenses, we start with the two prime licenses from the Free Software Foundation, the GNU General Public License (GPL) and the Lesser GNU General Public License (LGPL), after which we add a proven (used by the GNU-C compiler runtime library as well as the GNU *ClassPath* project) exception to deal with the specific nature of compiled virtual machine code in a saved state.

E.1 The SWI-Prolog kernel and foreign libraries

The SWI-Prolog kernel and our foreign libraries are distributed under the **LGPL**. A Prolog executable consists of the combination of these 'native' code components and Prolog virtual machine code. The SWI-Prolog `swipl-rc` utility allows for disassembling and re-assembling these parts, a process satisfying article **6b** of the LGPL.

Under the LGPL SWI-Prolog can be linked to code distributed under arbitrary licenses, provided a number of requirements are fulfilled. The most important requirement is that, if an application relies on a *modified* version of SWI-Prolog, the modified sources must be made available.

E.1.1 The SWI-Prolog Prolog libraries

Lacking a satisfactory technical solution to handle article **6** of the LGPL, this license cannot be used for the Prolog source code that is part of the SWI-Prolog system (both libraries and kernel code). This situation is comparable to `libgcc`, the runtime library used with the GNU C-compiler. Therefore, we use the same proven license terms as this library. The libgcc license is the with a special exception. Below we rephrase this exception adjusted to our needs:

> *As a special exception, if you link this library with other files, compiled with a Free Software compiler, to produce an executable, this library does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.*

E.2 Contributing to the SWI-Prolog project

To achieve maximal coherence using SWI-Prolog for Free and Non-Free software we advise using LGPL for contributed foreign code and using GPL with the SWI-Prolog exception for Prolog code for contributed modules.

As a rule of thumb it is advised to use the above licenses whenever possible, and use a strict GPL compliant license only if the module contains other code under strict GPL compliant licenses.

E.3 Software support to keep track of license conditions

Given the above, it is possible that SWI-Prolog packages and extensions will rely on the GPL.[1] The predicates below allow for registering license requirements for Prolog files and foreign modules. The predicate `eval_license/0` reports which components from the currently configured system are distributed under copy-left and open source enforcing licenses (the GPL) and therefore must be replaced before distributing linked applications under non-free license conditions.

eval_license

Evaluate the license conditions of all loaded components. If the system contains one or more components that are licenced under GPL-like restrictions the system indicates this program may only be distributed under the `GPL` license as well as which components prohibit the use of other license conditions.

license(+*LicenseId*, +*Component***)**

Register the fact that *Component* is distributed under a license identified by *LicenseId*. The most important *LicenseId*'s are:

swipl

Indicates this module is distributed under the GNU General Public License (GPL) with the SWI-Prolog exception:[2]

> *As a special exception, if you link this library with other files, compiled with SWI-Prolog, to produce an executable, this library does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.*

---

[1] On the Unix version, the default toplevel uses the GNU readline library for command-line editing. This library is distributed under the GPL. In practice this problem is small as most final applications have Prolog embedded, without direct access to the commandline and therefore without need for `libreadline`.

[2] This exception is a straight re-phrasing of the license used for `libgcc`, the GNU-C runtime library facing similar technical issues.

This should be the default for software contributed to the SWI-Prolog project as it allows the community to prosper both in the free and non-free world. Still, people using SWI-Prolog to create non-free applications must contribute sources to improvements they make to the community.

lgpl
This is the default license for foreign libraries linked with SWI-Prolog. Use `PL_license()` to register the condition from foreign code.

gpl
Indicates this module is strictly Free Software, which implies it cannot be used together with any module that is incompatible with the GPL. Please only use these conditions when forced by other code used in the component.

Other licenses known to the system are `guile`, `gnu_ada`, `x11`, `expat`, `sml`, `public_domain`, `cryptix`, `bsd`, `zlib`, `lgpl_compatible` and `gpl_compatible`. New licenses can be defined by adding clauses for the multifile predicate license:license/3. Below is an example. The second argument is either `gpl` or `lgpl` to indicate compatibility with these licenses. Other values cause the license to be interpreted as *proprietary*. Proprietary licenses are reported by `eval_license/0`. See the file `boot/license.pl` for details.

```
:- multifile license:license/3.

license:license(mylicense, lgpl,
                [ comment('My personal license'),
                  url('http://www.mine.org/license.html')
                ]).

:- license(mylicense).
```

license(+*LicenseId***)**
Intended as a directive in Prolog source files. It takes the current filename and calls `license/2`.

`void` **PL_license**(*const char *LicenseId, const char *Component*)
Intended for the install() procedure of foreign libraries. This call can be made *before* `PL_initialise()`.

E.4 License conditions inherited from used code

E.4.1 Cryptographic routines

Cryptographic routines are used in `variant_sha1/2` and `crypt`. These routines are provided under the following conditions.

```
Copyright (c) 2002, Dr Brian Gladman, Worcester, UK.   All rights reserved.

LICENSE TERMS
```

The free distribution and use of this software in both source and binary form is allowed (with or without changes) provided that:

1. distributions of this source code include the above copyright notice, this list of conditions and the following disclaimer;

2. distributions in binary form include the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other associated materials;

3. the copyright holder’s name is not used to endorse products built using this software without specific written permission.

ALTERNATIVELY, provided that this notice is retained in full, this product may be distributed under the terms of the GNU General Public License (GPL), in which case the provisions of the GPL apply INSTEAD OF those given above.

DISCLAIMER

This software is provided ’as is’ with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose.

F

Summary

F.1 Predicates

The predicate summary is used by the Prolog predicate `apropos/1` to suggest predicates from a keyword.

!/0	Cut (discard choicepoints)
,/2	Conjunction of goals
->/2	If-then-else
*->/2	Soft-cut
./2	Consult. Also list constructor
;/2	Disjunction of goals. Same as |/2
</2	Arithmetic smaller
=/2	Unification
=../2	“Univ.” Term to list conversion
=:=/2	Arithmetic equal
=</2	Arithmetic smaller or equal
==/2	Identical
=@=/2	Structural identical
=\=/2	Arithmetic not equal
>/2	Arithmetic larger
>=/2	Arithmetic larger or equal
?=/2	Test of terms can be compared now
@</2	Standard order smaller
@=</2	Standard order smaller or equal
@>/2	Standard order larger
@>=/2	Standard order larger or equal
\+/1	Negation by failure. Same as not/1
\=/2	Not unifiable
\==/2	Not identical
\=@=/2	Not structural identical
^/2	Existential quantification (bagof/3, setof/3)
|/2	Disjunction of goals. Same as ;/2
{}/1	DCG escape; constraints
abolish/1	Remove predicate definition from the database
abolish/2	Remove predicate definition from the database
abort/0	Abort execution, return to top level
absolute_file_name/2	Get absolute path name
absolute_file_name/3	Get absolute path name with options

access_file/2	Check access permissions of a file
acyclic_term/1	Test term for cycles
add_import_module/3	Add module to the auto-import list
add_nb_set/2	Add term to a non-backtrackable set
add_nb_set/3	Add term to a non-backtrackable set
append/1	Append to a file
apply/2	Call goal with additional arguments
apropos/1	`online_help` Search manual
arg/3	Access argument of a term
assoc_to_list/2	Convert association tree to list
assert/1	Add a clause to the database
assert/2	Add a clause to the database, give reference
asserta/1	Add a clause to the database (first)
asserta/2	Add a clause to the database (first)
assertion/1	Make assertions about your program
assertz/1	Add a clause to the database (last)
assertz/2	Add a clause to the database (last)
attach_console/0	Attach I/O console to thread
attribute_goals/3	Project attributes to goals
attr_unify_hook/2	Attributed variable unification hook
attr_portray_hook/2	Attributed variable print hook
attvar/1	Type test for attributed variable
at_end_of_stream/0	Test for end of file on input
at_end_of_stream/1	Test for end of file on stream
at_halt/1	Register goal to run at `halt/1`
atom/1	Type check for an atom
atom_chars/2	Convert between atom and list of characters
atom_codes/2	Convert between atom and list of characters codes
atom_concat/3	Append two atoms
atom_length/2	Determine length of an atom
atom_prefix/2	Test for start of atom
atom_number/2	Convert between atom and number
atom_to_term/3	Convert between atom and term
atomic/1	Type check for primitive
atomic_concat/3	Concatenate two atomic values to an atom
atomic_list_concat/2	Append a list of atoms
atomic_list_concat/3	Append a list of atoms with separator
autoload/0	Autoload all predicates now
autoload_path/1	Add directories for autoloading
b_getval/2	Fetch backtrackable global variable
b_setval/2	Assign backtrackable global variable
bagof/3	Find all solutions to a goal
between/3	Integer range checking/generating
blob/2	Type check for a blob
break/0	Start interactive top-level
byte_count/2	Byte-position in a stream
call/1	Call a goal

call/[2..]	Call with additional arguments
call_cleanup/3	Guard a goal with a cleaup-handler
call_cleanup/2	Guard a goal with a cleaup-handler
call_residue_vars/2	Find residual attributed variables
call_shared_object_function/2	UNIX: Call C-function in shared (.so) file
call_with_depth_limit/3	Prove goal with bounded depth
callable/1	Test for atom or compound term
catch/3	Call goal, watching for exceptions
char_code/2	Convert between character and character code
char_conversion/2	Provide mapping of input characters
char_type/2	Classify characters
character_count/2	Get character index on a stream
chdir/1	Compatibility: change working directory
chr_constraint/1	CHR Constraint declaration
chr_show_store/1	List suspended CHR constraints
chr_trace/0	Start CHR tracer
chr_type/1	CHR Type declaration
chr_notrace/0	Stop CHR tracer
chr_leash/1	Define CHR leashed ports
chr_option/2	Specify CHR compilation options
clause/2	Get clauses of a predicate
clause/3	Get clauses of a predicate
clause_property/2	Get properties of a clause
close/1	Close stream
close/2	Close stream (forced)
close_dde_conversation/1	Win32: Close DDE channel
close_shared_object/1	UNIX: Close shared library (.so file)
collation_key/2	Sort key for locale dependent ordering
comment_hook/3	(hook) handle comments in sources
compare/3	Compare, using a predicate to determine the order
compile_aux_clauses/1	Compile predicates for `goal_expansion/2`
compile_predicates/1	Compile dynamic code to static
compiling/0	Is this a compilation run?
compound/1	Test for compound term
code_type/2	Classify a character-code
consult/1	Read (compile) a Prolog source file
context_module/1	Get context module of current goal
convert_time/8	Break time stamp into fields
convert_time/2	Convert time stamp to string
copy_stream_data/2	Copy all data from stream to stream
copy_stream_data/3	Copy n bytes from stream to stream
copy_predicate_clauses/2	Copy clauses between predicates
copy_term/2	Make a copy of a term
copy_term/3	Copy a term and obtain attribute-goals
copy_term_nat/2	Make a copy of a term without attributes
create_prolog_flag/3	Create a new Prolog flag
current_arithmetic_function/1	Examine evaluable functions

current_atom/1	Examine existing atoms
current_blob/2	Examine typed blobs
current_char_conversion/2	Query input character mapping
current_flag/1	Examine existing flags
current_foreign_library/2	`shlib` Examine loaded shared libraries (.so files)
current_format_predicate/2	Enumerate user-defined format codes
current_functor/2	Examine existing name/arity pairs
current_input/1	Get current input stream
current_key/1	Examine existing database keys
current_module/1	Examine existing modules
current_op/3	Examine current operator declarations
current_output/1	Get the current output stream
current_predicate/1	Examine existing predicates (ISO)
current_predicate/2	Examine existing predicates
current_signal/3	Current software signal mapping
current_stream/3	Examine open streams
cyclic_term/1	Test term for cycles
day_of_the_week/2	Determine ordinal-day from date
date_time_stamp/2	Convert sate structure to time-stamp
date_time_value/3	Extract info from a date structure
dcg_translate_rule/2	Source translation of DCG rules
dde_current_connection/2	Win32: Examine open DDE connections
dde_current_service/2	Win32: Examine DDE services provided
dde_execute/2	Win32: Execute command on DDE server
dde_register_service/2	Win32: Become a DDE server
dde_request/3	Win32: Make a DDE request
dde_poke/3	Win32: POKE operation on DDE server
dde_unregister_service/1	Win32: Terminate a DDE service
debug/0	Test for debugging mode
debug/1	Select topic for debugging
debug/3	Print debugging message on topic
debug_control_hook/1	(hook) Extend `spy/1`, etc.
debugging/0	Show debugger status
debugging/1	Test where we are debugging topic
default_module/2	Query module inheritance
del_attr/2	Delete attribute from variable
del_attrs/1	Delete all attributes from variable
delete_directory/1	Remove a folder from the file system
delete_file/1	Remove a file from the file system
delete_import_module/2	Remove module from import list
deterministic/1	Test deterministicy of current clause
dif/2	Constrain two terms to be different
directory_files/2	Get entries of a directory/folder
discontiguous/1	Indicate distributed definition of a predicate
downcase_atom/2	Convert atom to lower-case
duplicate_term/2	Create a copy of a term
dwim_match/2	Atoms match in “Do What I Mean” sense

dwim_match/3	Atoms match in “Do What I Mean” sense
dwim_predicate/2	Find predicate in “Do What I Mean” sense
dynamic/1	Indicate predicate definition may change
edit/0	Edit current script- or associated file
edit/1	Edit a file, predicate, module (extensible)
elif/1	Part of conditional compilation (directive)
else/0	Part of conditional compilation (directive)
empty_assoc/1	Create/test empty association tree
empty_nb_set/1	Test/create an empty non-backtrackable set
encoding/1	Define encoding inside a source file
endif/0	End of conditional compilation (directive)
ensure_loaded/1	Consult a file if that has not yet been done
erase/1	Erase a database record or clause
eval_license/0	Evaluate licenses of loaded modules
exception/3	(hook) Handle runtime exceptions
exists_directory/1	Check existence of directory
exists_file/1	Check existence of file
exists_source/1	Check existence of a Prolog source
expand_answer/2	Expand answer of query
expand_file_name/2	Wildcard expansion of file names
expand_file_search_path/2	Wildcard expansion of file paths
expand_goal/2	Compiler: expand goal in clause-body
expand_query/4	Expanded entered query
expand_term/2	Compiler: expand read term into clause(s)
expects_dialect/1	For which Prolog dialect is this code written?
explain/1	`explain` Explain argument
explain/2	`explain` 2nd argument is explanation of first
export/1	Export a predicate from a module
fail/0	Always false
false/0	Always false
current_prolog_flag/2	Get system configuration parameters
file_base_name/2	Get file part of path
file_directory_name/2	Get directory part of path
file_name_extension/3	Add, remove or test file extensions
file_search_path/2	Define path-aliases for locating files
find_chr_constraint/1	Returns a constraint from the store
findall/3	Find all solutions to a goal
findall/4	Difference list version of `findall/3`
flag/3	Simple global variable system
float/1	Type check for a floating point number
flush_output/0	Output pending characters on current stream
flush_output/1	Output pending characters on specified stream
forall/2	Prove goal for all solutions of another goal
format/1	Formatted output
format/2	Formatted output with arguments
format/3	Formatted output on a stream
format_time/3	C strftime() like date/time formatter

format_time/4	date/time formatter with explicit locale
format_predicate/2	Program `format/[1,2]`
term_attvars/2	Find attributed variables in a term
term_variables/2	Find unbound variables in a term
term_variables/3	Find unbound variables in a term
freeze/2	Delay execution until variable is bound
frozen/2	Query delayed goals on var
functor/3	Get name and arity of a term or construct a term
garbage_collect/0	Invoke the garbage collector
garbage_collect_atoms/0	Invoke the atom garbage collector
garbage_collect_clauses/0	Invoke clause garbage collector
gen_assoc/3	Enumerate members of association tree
gen_nb_set/2	Generate members of non-backtrackable set
gensym/2	Generate unique atoms from a base
get/1	Read first non-blank character
get/2	Read first non-blank character from a stream
get_assoc/3	Fetch key from association tree
get_assoc/5	Fetch key from association tree
get0/1	Read next character
get0/2	Read next character from a stream
get_attr/3	Fetch named attribute from a variable
get_attrs/2	Fetch all attributes of a variable
get_byte/1	Read next byte (ISO)
get_byte/2	Read next byte from a stream (ISO)
get_char/1	Read next character as an atom (ISO)
get_char/2	Read next character from a stream (ISO)
get_code/1	Read next character (ISO)
get_code/2	Read next character from a stream (ISO)
get_single_char/1	Read next character from the terminal
get_time/1	Get current time
getenv/2	Get shell environment variable
goal_expansion/2	Hook for macro-expanding goals
ground/1	Verify term holds no unbound variables
gdebug/0	Debug using graphical tracer
gspy/1	Spy using graphical tracer
gtrace/0	Trace using graphical tracer
guitracer/0	Install hooks for the graphical debugger
gxref/0	Cross-reference loaded program
halt/0	Exit from Prolog
halt/1	Exit from Prolog with status
term_hash/2	Hash-value of ground term
term_hash/4	Hash-value of term with depth limit
help/0	Give help on help
help/1	Give help on predicates and show parts of manual
help_hook/1	(hook) User-hook in the help-system
if/1	Start conditional compilation (directive)
ignore/1	Call the argument, but always succeed

import/1	Import a predicate from a module
import_module/2	Query import modules
in_pce_thread/1	Run goal in XPCE thread
in_pce_thread_sync/1	Run goal in XPCE thread
include/1	Include a file with declarations
initialization/1	Initialization directive
initialization/2	Initialization directive
instance/2	Fetch clause or record from reference
integer/1	Type check for integer
interactor/0	Start new thread with console and top-level
is/2	Evaluate arithmetic expression
is_absolute_file_name/1	True if arg defines an absolute path
is_list/1	Type check for a list
is_stream/1	Type check for a stream handle
join_threads/0	Join all terminated threads interactively
keysort/2	Sort, using a key
last/2	Last element of a list
leash/1	Change ports visited by the tracer
length/2	Length of a list
library_directory/1	(hook) Directories holding Prolog libraries
license/1	Define license for current file
license/2	Define license for named module
line_count/2	Line number on stream
line_position/2	Character position in line on stream
list_debug_topics/0	List registered topics for debugging
list_to_assoc/2	Create association tree from list
list_to_set/2	Remove duplicates from a list
listing/0	List program in current module
listing/1	List predicate
load_files/2	Load source files with options
load_foreign_library/1	`shlib` Load shared library (.so file)
load_foreign_library/2	`shlib` Load shared library (.so file)
locale_sort/2	Language dependent sort of atoms
make/0	Reconsult all changed source files
make_directory/1	Create a folder on the file system
make_library_index/1	Create autoload file INDEX.pl
make_library_index/2	Create selective autoload file INDEX.pl
map_assoc/2	Map association tree
map_assoc/3	Map association tree
max_assoc/3	Highest key in association tree
memberchk/2	Deterministic `member/2`
message_hook/3	Intercept `print_message/2`
message_line_element/2	(hook) Intercept `print_message_lines/3`
message_queue_create/1	Create queue for thread communication
message_queue_create/2	Create queue for thread communication
message_queue_destroy/1	Destroy queue for thread communication
message_queue_property/2	Query message queue properties

message_to_string/2	Translate message-term to string
meta_predicate/1	Declare access to other predicates
min_assoc/3	Lowest key in association tree
module/1	Query/set current type-in module
module/2	Declare a module
module_property/2	Find properties of a module
module_transparent/1	Indicate module based meta-predicate
msort/2	Sort, do not remove duplicates
multifile/1	Indicate distributed definition of predicate
mutex_create/1	Create a thread-synchronisation device
mutex_create/2	Create a thread-synchronisation device
mutex_destroy/1	Destroy a mutex
mutex_lock/1	Become owner of a mutex
mutex_property/2	Query mutex properties
mutex_statistics/0	Print statistics on mutex usage
mutex_trylock/1	Become owner of a mutex (non-blocking)
mutex_unlock/1	Release ownership of mutex
mutex_unlock_all/0	Release ownership of all mutexes
name/2	Convert between atom and list of character codes
nb_current/2	Enumerate non-backtrackable global variables
nb_delete/1	Delete a non-backtrackable global variable
nb_getval/2	Fetch non-backtrackable global variable
nb_linkarg/3	Non-backtrackable assignment to term
nb_linkval/2	Assign non-backtrackable global variable
nb_set_to_list/2	Convert non-backtrackable set to list
nb_setarg/3	Non-backtrackable assignment to term
nb_setval/2	Assign non-backtrackable global variable
nl/0	Generate a newline
nl/1	Generate a newline on a stream
nodebug/0	Disable debugging
nodebug/1	Disable debug-topic
noguitracer/0	Disable the graphical debugger
nonvar/1	Type check for bound term
noprofile/1	Hide (meta-) predicate for the profiler
noprotocol/0	Disable logging of user interaction
normalize_space/2	Normalize white space
nospy/1	Remove spy point
nospyall/0	Remove all spy points
not/1	Negation by failure (argument not provable). Same as `\+/1`
notrace/0	Stop tracing
notrace/1	Do not debug argument goal
nth_clause/3	N-th clause of a predicate
number/1	Type check for integer or float
number_chars/2	Convert between number and one-char atoms
number_codes/2	Convert between number and character codes
numbervars/3	Number unbound variables of a term
numbervars/4	Number unbound variables of a term

on_signal/3	Handle a software signal
once/1	Call a goal deterministically
op/3	Declare an operator
open/3	Open a file (creating a stream)
open/4	Open a file (creating a stream)
open_dde_conversation/3	Win32: Open DDE channel
open_null_stream/1	Open a stream to discard output
open_resource/3	Open a program resource as a stream
open_shared_object/2	UNIX: Open shared library (.so file)
open_shared_object/3	UNIX: Open shared library (.so file)
ord_list_to_assoc/2	Convert ordered list to assoc
parse_time/2	Parse text to a time-stamp
parse_time/3	Parse text to a time-stamp
pce_dispatch/1	Run XPCE GUI in separate thread
pce_call/1	Run goal in XPCE GUI thread
peek_byte/1	Read byte without removing
peek_byte/2	Read byte without removing
peek_char/1	Read character without removing
peek_char/2	Read character without removing
peek_code/1	Read character-code without removing
peek_code/2	Read character-code without removing
phrase/2	Activate grammar-rule set
phrase/3	Activate grammar-rule set (returning rest)
please/3	Query/change environment parameters
plus/3	Logical integer addition
portray/1	(hook) Modify behaviour of `print/1`
portray_clause/1	Pretty print a clause
portray_clause/2	Pretty print a clause to a stream
predicate_property/2	Query predicate attributes
predsort/3	Sort, using a predicate to determine the order
preprocessor/2	Install a preprocessor before the compiler
print/1	Print a term
print/2	Print a term on a stream
print_message/2	Print message from (exception) term
print_message_lines/3	Print message to stream
profile/1	Obtain execution statistics
profile/3	Obtain execution statistics
profile_count/3	Obtain profile results on a predicate
profiler/2	Obtain/change status of the profiler
prolog/0	Run interactive top-level
prolog_choice_attribute/3	Examine the choice-point stack
prolog_current_frame/1	Reference to goal's environment stack
prolog_edit:locate/2	Locate targets for `edit/1`
prolog_edit:locate/3	Locate targets for `edit/1`
prolog_edit:edit_source/1	Call editor for `edit/1`
prolog_edit:edit_command/2	Specify editor activation
prolog_edit:load/0	Load `edit/1` extensions

prolog_exception_hook/4	Rewrite exceptions
prolog_file_type/2	Define meaning of file extension
prolog_frame_attribute/3	Obtain information on a goal environment
prolog_ide/1	Program access to the development environment
prolog_list_goal/1	(hook) Intercept tracer 'L' command
prolog_load_context/2	Context information for directives
prolog_load_file/2	(hook) Program `load_files/2`
prolog_skip_level/2	Indicate deepest recursion to trace
prolog_skip_frame/1	Perform 'skip' on a frame
prolog_stack_property/2	Query properties of the stacks
prolog_to_os_filename/2	Convert between Prolog and OS filenames
prolog_trace_interception/4	`user` Intercept the Prolog tracer
prompt1/1	Change prompt for 1 line
prompt/2	Change the prompt used by `read/1`
protocol/1	Make a log of the user interaction
protocola/1	Append log of the user interaction to file
protocolling/1	On what file is user interaction logged
public/1	Declaration that a predicate may be called
put/1	Write a character
put/2	Write a character on a stream
put_assoc/4	Add Key-Value to association tree
put_attr/3	Put attribute on a variable
put_attrs/2	Set/replace all attributes on a variable
put_byte/1	Write a byte
put_byte/2	Write a byte on a stream
put_char/1	Write a character
put_char/2	Write a character on a stream
put_code/1	Write a character-code
put_code/2	Write a character-code on a stream
qcompile/1	Compile source to Quick Load File
qcompile/2	Compile source to Quick Load File
qsave_program/1	Create runtime application
qsave_program/2	Create runtime application
random_property/1	Query properties of random generation
rational/1	Type check for a rational number
rational/3	Decompose a rational
read/1	Read Prolog term
read/2	Read Prolog term from stream
read_clause/1	Read clause
read_clause/2	Read clause from stream
read_history/6	Read using history substitution
read_link/3	Read a symbolic link
read_pending_input/3	Fetch buffered input from a stream
read_term/2	Read term with options
read_term/3	Read term with options from stream
recorda/2	Record term in the database (first)
recorda/3	Record term in the database (first)

recorded/2	Obtain term from the database
recorded/3	Obtain term from the database
recordz/2	Record term in the database (last)
recordz/3	Record term in the database (last)
redefine_system_predicate/1	Abolish system definition
reexport/1	Load files and re-export the imported predicates
reexport/2	Load predicates from a file and re-export it
reload_foreign_libraries/0	Reload DLLs/shared objects
reload_library_index/0	Force reloading the autoload index
rename_file/2	Change name of file
repeat/0	Succeed, leaving infinite backtrack points
require/1	This file requires these predicates
reset_gensym/1	Reset a gensym key
reset_gensym/0	Reset all gensym keys
reset_profiler/0	Clear statistics obtained by the profiler
resource/3	Declare a program resource
retract/1	Remove clause from the database
retractall/1	Remove unifying clauses from the database
same_file/2	Succeeds if arguments refer to same file
same_term/2	Test terms to be at the same address
see/1	Change the current input stream
seeing/1	Query the current input stream
seek/4	Modify the current position in a stream
seen/0	Close the current input stream
set_end_of_stream/1	Set physical end of an open file
set_input/1	Set current input stream from a stream
set_module/1	Set properties of a module
set_output/1	Set current output stream from a stream
set_prolog_IO/3	Prepare streams for interactive session
set_prolog_flag/2	Define a system feature
set_prolog_stack/2	Modify stack characteristics
set_random/1	Control random number generation
set_stream/2	Set stream attribute
set_stream_position/2	Seek stream to position
set_tty/2	Set ‘tty’ stream
setup_call_cleanup/3	Undo side-effects safely
setup_call_catcher_cleanup/4	Undo side-effects safely
setarg/3	Destructive assignment on term
setenv/2	Set shell environment variable
setlocale/3	Set/query C-library regional information
setof/3	Find all unique solutions to a goal
shell/0	Execute interactive subshell
shell/1	Execute OS command
shell/2	Execute OS command
show_profile/1	Show results of the profiler
show_profile/2	Show results of the profiler
size_file/2	Get size of a file in characters

size_nb_set/2	Determine size of non-backtrackable set
skip/1	Skip to character in current input
skip/2	Skip to character on stream
rl_add_history/1	Add line to readline(3) history
rl_read_history/1	Read readline(3) history
rl_read_init_file/1	Read readline(3) init file
rl_write_history/1	Write readline(3) history
sleep/1	Suspend execution for specified time
sort/2	Sort elements in a list
source_exports/2	Check whether source exports a predicate
source_file/1	Examine currently loaded source files
source_file/2	Obtain source file of predicate
source_file_property/2	Information about loaded files
source_location/2	Location of last read term
spy/1	Force tracer on specified predicate
stamp_date_time/3	Convert time-stamp to date structure
statistics/0	Show execution statistics
statistics/2	Obtain collected statistics
stream_pair/3	Create/examine a bi-directional stream
stream_position_data/3	Access fields from stream position
stream_property/2	Get stream properties
string/1	Type check for string
string_concat/3	atom_concat/3 for strings
string_length/2	Determine length of a string
string_to_atom/2	Conversion between string and atom
string_to_list/2	Conversion between string and list of character codes
strip_module/3	Extract context module and term
style_check/1	Change level of warnings
sub_atom/5	Take a substring from an atom
sub_string/5	Take a substring from a string
subsumes_term/2	One-sided unification test
succ/2	Logical integer successor relation
swritef/2	Formatted write on a string
swritef/3	Formatted write on a string
tab/1	Output number of spaces
tab/2	Output number of spaces on a stream
tdebug/0	Switch all threads into debug mode
tdebug/1	Switch a thread into debug mode
tell/1	Change current output stream
telling/1	Query current output stream
term_expansion/2	(hook) Convert term before compilation
term_subsumer/3	Most specific generalization of two terms
term_to_atom/2	Convert between term and atom
thread_at_exit/1	Register goal to be called at exit
thread_create/3	Create a new Prolog task
thread_detach/1	Make thread cleanup after completion
thread_exit/1	Terminate Prolog task with value

thread_get_message/1	Wait for message
thread_get_message/2	Wait for message in a queue
thread_get_message/3	Wait for message in a queue
thread_initialization/1	Run action at start of thread
thread_join/2	Wait for Prolog task-completion
thread_local/1	Declare thread-specific clauses for a predicate
thread_peek_message/1	Test for message
thread_peek_message/2	Test for message in a queue
thread_property/2	Examine Prolog threads
thread_self/1	Get identifier of current thread
thread_send_message/2	Send message to another thread
thread_setconcurrency/2	Number of active threads
thread_signal/2	Execute goal in another thread
thread_statistics/3	Get statistics of another thread
threads/0	List running threads
throw/1	Raise an exception (see `catch/3`)
time/1	Determine time needed to execute goal
time_file/2	Get last modification time of file
tmp_file/2	Create a temporary filename
tmp_file_stream/3	Create a temporary file and open it
tnodebug/0	Switch off debug mode in all threads
tnodebug/1	Switch off debug mode in a thread
told/0	Close current output
tprofile/1	Profile a thread for some period
trace/0	Start the tracer
trace/1	Set trace-point on predicate
trace/2	Set/Clear trace-point on ports
tracing/0	Query status of the tracer
trim_stacks/0	Release unused memory resources
true/0	Succeed
tspy/1	Set spy point and enable debugging in all threads
tspy/2	Set spy point and enable debugging in a thread
tty_get_capability/3	Get terminal parameter
tty_goto/2	Goto position on screen
tty_put/2	Write control string to terminal
tty_size/2	Get row/column size of the terminal
ttyflush/0	Flush output on terminal
unify_with_occurs_check/2	Logically sound unification
unifiable/3	Determining binding required for unification
unix/1	OS interaction
unknown/2	Trap undefined predicates
unload_file/1	Unload a source-file
unload_foreign_library/1	`shlib` Detach shared library (.so file)
unload_foreign_library/2	`shlib` Detach shared library (.so file)
unsetenv/1	Delete shell environment variable
upcase_atom/2	Convert atom to upper-case
use_foreign_library/1	Load DLL/shared object (directive)

use_foreign_library/2	Load DLL/shared object (directive)
use_module/1	Import a module
use_module/2	Import predicates from a module
var/1	Type check for unbound variable
var_number/2	Check that var is numbered by numbervars
variant_sha1/2	Term-hash for term-variants
visible/1	Ports that are visible in the tracer
volatile/1	Predicates that are not saved
wait_for_input/3	Wait for input with optional timeout
when/2	Execute goal when condition becomes true
wildcard_match/2	Csh(1) style wildcard match
win_exec/2	Win32: spawn Windows task
win_has_menu/0	Win32: true if console menu is available
win_folder/2	Win32: get special folder by CSIDL
win_insert_menu/2	swipl-win.exe: add menu
win_insert_menu_item/4	swipl-win.exe: add item to menu
win_shell/2	Win32: open document through Shell
win_shell/3	Win32: open document through Shell
win_registry_get_value/3	Win32: get registry value
win_window_pos/1	Win32: change size and position of window
window_title/2	Win32: change title of window
with_mutex/2	Run goal while holding mutex
with_output_to/2	Write to strings and more
working_directory/2	Query/change CWD
write/1	Write term
write/2	Write term to stream
writeln/1	Write term, followed by a newline
write_canonical/1	Write a term with quotes, ignore operators
write_canonical/2	Write a term with quotes, ignore operators on a stream
write_length/3	Dermine #characters to output a term
write_term/2	Write term with options
write_term/3	Write term with options to stream
writef/1	Formatted write
writef/2	Formatted write on stream
writeq/1	Write term, insert quotes
writeq/2	Write term, insert quotes on stream

F.2 Library predicates

F.2.1 `aggregate`

aggregate/3	Aggregate bindings in Goal according to Template.
aggregate/4	Aggregate bindings in Goal according to Template.
aggregate_all/3	Aggregate bindings in Goal according to Template.
aggregate_all/4	Aggregate bindings in Goal according to Template.
foreach/2	True if the conjunction of instances of Goal using the bindings from Generator is true.
free_variables/4	In order to handle variables properly, we have to find all the universally quantified variables in the Ger

F.2.2 `apply`

exclude/3	Filter elements for which Goal fails.
include/3	Filter elements for which Goal succeed.
maplist/2	True if Goal can succesfully be applied on all elements of List.
maplist/3	True if Goal can succesfully be applied to all succesive pairs of elements of List1 and List2.
maplist/4	True if Goal can succesfully be applied to all succesive triples of elements of List1..List3.
maplist/5	True if Goal can succesfully be applied to all succesive quadruples of elements of List1..List4.
partition/4	Filter elements of List according to Pred.
partition/5	Filter list according to Pred in three sets.

F.2.3 `assoc`

assoc_to_list/2	Translate assoc into a pairs list
assoc_to_keys/2	Translate assoc into a key list
assoc_to_values/2	Translate assoc into a value list
empty_assoc/1	Test/create an empty assoc
gen_assoc/3	Non-deterministic enumeration of assoc
get_assoc/3	Get associated value
get_assoc/5	Get and replace associated value
list_to_assoc/2	Translate pair list to assoc
map_assoc/2	Test assoc values
map_assoc/3	Map assoc values
max_assoc/3	Max key-value of an assoc
min_assoc/3	Min key-value of an assoc
ord_list_to_assoc/3	Translate ordered list into an assoc
put_assoc/4	Add association to an assoc

F.2.4 `broadcast`

broadcast/1	Send event notification
broadcast_request/1	Request all agents
listen/2	Listen to event notifications
listen/3	Listen to event notifications
unlisten/1	Stop listening to event notifications

unlisten/2	Stop listening to event notifications
unlisten/3	Stop listening to event notifications
listening/3	Who is listening to event notifications?

F.2.5 `charsio`

atom_to_chars/2	Convert Atom into a list of character codes.
atom_to_chars/3	Convert Atom into a difference-list of character codes.
format_to_chars/3	Use format/2 to write to a list of character codes.
format_to_chars/3	Use format/2 to write to a list of character codes.
number_to_chars/2	Convert Atom into a list of character codes.
number_to_chars/3	Convert Number into a difference-list of character codes.
open_chars_stream/2	Open Codes as an input stream.
read_from_chars/2	Read Codes into Term.
read_term_from_chars/3	Read Codes into Term.
with_output_to_chars/2	Run Goal with as once/1.
with_output_to_chars/3	Run Goal with as once/1.
with_output_to_chars/4	As with_output_to_chars/2, but Stream is unified with the temporary stream.
write_to_chars/2	Codes is a list of character codes produced by write/1 on Term.
write_to_chars/3	Codes is a difference-list of character codes produced by write/1 on Term.

F.2.6 `check`

check/0	Program completeness and consistency
list_undefined/0	List undefined predicates
list_autoload/0	List predicates that require autoload
list_redefined/0	List locally redefined predicates

F.2.7 `csv`

csv_read_file/2	Read a CSV file into a list of rows.
csv_read_file/3	Read a CSV file into a list of rows.
csv_read_file_row/3	True when Row is a row in File.
csv_write_file/2	Write a list of Prolog terms to a CSV file.
csv_write_file/3	Write a list of Prolog terms to a CSV file.
csv_write_stream/3	Write the rows in Data to Stream.
csv/3	Prolog DCG to ‘read/write’ CSV data.
csv/4	Prolog DCG to ‘read/write’ CSV data.

F.2.8 `lists`

append/2	Concatenate a list of lists.
append/3	List1AndList2 is the concatination of List1 and List2.
delete/3	Is true when Lis1, with all occurences of Elem deleted results in List2.
flatten/2	Is true it List2 is a non nested version of List1.
intersection/3	True if Set3 unifies with the intersection of Set1 and Set2.

is_set/1	True if Set is a proper list without duplicates.
last/2	Succeeds when ‘Last’ is the last element of ‘List’.
list_to_set/2	True when Set has the same element as List in the same order.
max_list/2	True if Max is the largest number in List.
max_member/2	True when Max is the largest member in the standard order of terms.
member/2	True if Elem is a member of List.
min_list/2	True if Min is the largest number in List.
min_member/2	True when Min is the smallest member in the standard order of terms.
nextto/3	True of Y follows X in List.
nth0/3	True when Elem is the Index-th element of List.
nth0/4	Select/insert element at index.
nth1/3	Is true when Elem is the Index’th element of List.
nth1/4	As nth0/4, but counting starts at 1.
numlist/3	List is a list [Low, Low+1, ... High].
permutation/2	permutation(Xs, Ys) is true when Xs is a permutation of Ys.
prefix/2	True iff Part is a leading substring of Whole.
proper_length/2	True when Length is the number of elements in the proper list List.
reverse/2	Is true when the elements of List2 are in reverse order compared to List1.
same_length/2	Is true when List1 and List2 are lists with the same number of elements.
select/3	Is true when List1, with Elem removed results in List2.
select/4	Is true when select(X, XList) and select(Y, YList) are true, X and Y appear in the same locations of th
selectchk/3	Semi-deterministic removal of first element in List that unifies Elem.
selectchk/4	Semi-deterministic version of select/4.
subset/2	True if all elements of SubSet belong to Set as well.
subtract/3	Delete all elements from ‘Set’ that occur in ‘Delete’ (a set) and unify the result with ‘Result’.
sumlist/2	Sum is the result of adding all numbers in List.
union/3	True if Set3 unifies with the union of Set1 and Set2.

F.2.9 `debug`

assertion/1	Acts similar to C assert() macro.
assertion_failed/2	This hook is called if the Goal of assertion/1 fails.
debug/1	Add/remove a topic from being printed.
debug/3	Similar to format/3 to =user_error=, but only prints if Topic is activated through debug/1.
debug_message_context/1	Specify additional context for debug messages.
debug_print_hook/3	Hook called by debug/3.
debugging/1	Check whether we are debugging Topic or enumerate the topics we are debugging.
debugging/2	Check whether we are debugging Topic or enumerate the topics we are debugging.
list_debug_topics/0	List currently known debug topics and their setting.
nodebug/1	Add/remove a topic from being printed.

F.2.10 `option`

merge_options/3	Merge two option lists.
meta_options/3	Perform meta-expansion on options that are module-sensitive.
option/2	Get an option from a OptionList.

option/3	Get an option from a OptionList.
select_option/3	Get and remove option from an option list.
select_option/4	Get and remove option with default value.

F.2.11 `optparse`

opt_arguments/3	Convenience predicate, assuming that command-line arguments can be accessed by current_prolog_fl
opt_help/2	Returns a help string (atom), synthesized from the help descriptions in OptsSpec.
opt_parse/4	Equivalent to opt_parse(OptsSpec, ApplArgs, Opts, PositionalArgs, []).
opt_parse/5	Parse the arguments Args (as list of atoms) according to OptsSpec.

F.2.12 `ordsets`

is_ordset/1	True if Term is an ordered set.
list_to_ord_set/2	Transform a list into an ordered set.
ord_add_element/3	Insert an element into the set.
ord_del_element/3	Delete an element from an ordered set.
ord_disjoint/2	True if Set1 and Set2 have no common elements.
ord_empty/1	True when List is the empty ordered set.
ord_intersect/2	True if both ordered sets have a non-empty intersection.
ord_intersect/3	Intersection holds the common elements of Set1 and Set2.
ord_intersection/2	True if Intersection is an ordered set holding all elements common to all sets in PowerSet.
ord_intersection/3	Intersection holds the common elements of Set1 and Set2.
ord_intersection/4	Intersection and difference between two ordered sets.
ord_member/2	True if Element is a member of Set.
ord_memberchk/2	Check membership.
ord_seteq/2	True if Set1 and Set2 have the same elements.
ord_subset/2	Is true if all element of Sub are in Super.
ord_subtract/3	Diff is the set holding all elements of InOSet that are not in NotInOSet.
ord_symdiff/3	Is true when Difference is the symmetric difference of Set1 and Set2.
ord_union/2	True if Union is the union of all elements in the superset SetOfSets.
ord_union/3	Union is the union of Set1 and Set2.
ord_union/4	True if Union iff ord_union(Set1, Set2, Union) and ord_subtract(Set2, Set1, New).

F.2.13 `predicate_options`

assert_predicate_options/4	As predicate_options(:PI, +Arg, +Options).
check_predicate_option/3	Similar to current_predicate_option/3, but intended to support runtime checking.
check_predicate_options/0	Analyse loaded program for errornous options.
current_option_arg/2	True when Arg of PI processes predicate options.
current_predicate_option/3	True when Arg of PI processes Option.
current_predicate_options/3	True when Options is the current active option declaration for PI on Arg.
derive_predicate_options/0	Derive new predicate option declarations.
derived_predicate_options/1	Derive predicate option declarations for the given module and print them to the current
derived_predicate_options/3	True when Options is the current _derived_ active option declaration for PI on Arg.
predicate_options/3	Declare that the predicate PI processes options on Arg.

retractall_predicate_options/0	Remove all dynamically (derived) predicate options.

F.2.14 `prologxref`

prolog:called_by/2	(hook) Extend cross-referencer
xref_built_in/1	Examine defined built-ins
xref_called/3	Examine called predicates
xref_clean/1	Remove analysis of source
xref_current_source/1	Examine cross-referenced sources
xref_defined/3	Examine defined predicates
xref_exported/2	Examine exported predicates
xref_module/2	Module defined by source
xref_source/1	Cross-reference analysis of source

F.2.15 `pairs`

group_pairs_by_key/2	Group values with the same key.
map_list_to_pairs/3	Create a key-value list by mapping each element of List.
pairs_keys/2	Remove the values from a list of Key-Value pairs.
pairs_keys_values/3	True if Keys holds the keys of Pairs and Values the values.
pairs_values/2	Remove the keys from a list of Key-Value pairs.
transpose_pairs/2	Swap Key-Value to Value-Key and sort the result on Value (the new key) using keysort/2.

F.2.16 `pio`

`pure_input`

phrase_from_file/2	Process the content of File using the DCG rule Grammar.
phrase_from_file/3	As phrase_from_file/2, providing additional Options.
stream_to_lazy_list/2	Create a lazy list representing the character codes in Stream.

F.2.17 `random`

getrand/1	Query/set the state of the random generator.
maybe/0	Succeed/fail with equal probability (variant of maybe/1).
maybe/1	Succeed with probability P, fail with probability 1-P.
maybe/2	Succeed with probability K/N (variant of maybe/1).
random/1	Binds R to a new random float in the _open_ interval (0.0,1.0).
random/3	If L and U are both integers, R is a random integer in the half open interval [L,U).
random_between/3	Binds R to a random integer in [L,U] (i.e., including both L and U).
random_member/2	X is a random member of List.
random_perm2/4	Does X=A,Y=B or X=B,Y=A with equal probability.
random_permutation/2	Permutation is a random permutation of List.
random_select/3	Randomly select or insert an element.
randseq/3	S is a list of K unique random integers in the range 1..N.
randset/3	S is a sorted list of K unique random integers in the range 1..N.

setrand/1	Query/set the state of the random generator.

F.2.18 `readutil`

read_line_to_codes/2	Read line from a stream
read_line_to_codes/3	Read line from a stream
read_stream_to_codes/2	Read contents of stream
read_stream_to_codes/3	Read contents of stream
read_file_to_codes/3	Read contents of file
read_file_to_terms/3	Read contents of file to Prolog terms

F.2.19 `record`

record/1	Define named fields in a term

F.2.20 `registry`

This library is only available on Windows systems.

registry_get_key/2	Get principal value of key
registry_get_key/3	Get associated value of key
registry_set_key/2	Set principal value of key
registry_set_key/3	Set associated value of key
registry_delete_key/1	Remove a key
shell_register_file_type/4	Register a file-type
shell_register_dde/6	Register DDE action
shell_register_prolog/1	Register Prolog

F.2.21 `ugraphs`

vertices_edges_to_ugraph/3	Create unweighted graph
vertices/2	Find vertices in graph
edges/2	Find edges in graph
add_vertices/3	Add vertices to graph
del_vertices/3	Delete vertices from graph
add_edges/3	Add edges to graph
del_edges/3	Delete edges from graph
transpose/2	Invert the direction of all edges
neighbors/3	Find neighbors of vertice
neighbours/3	Find neighbors of vertice
complement/2	Inverse presense of edges
compose/3	
top_sort/2	Sort graph topologically
top_sort/3	Sort graph topologically
transitive_closure/2	Create transitive closure of graph
reachable/3	Find all reachable vertices

ugraph_union/3	Union of two graphs

F.2.22 `url`

file_name_to_url/2	Translate between a filename and a file:// URL.
global_url/3	Translate a possibly relative URL into an absolute one.
http_location/2	Construct or analyze an HTTP location.
is_absolute_url/1	True if URL is an absolute URL.
parse_url/2	Construct or analyse a URL.
parse_url/3	Similar to parse_url/2 for relative URLs.
parse_url_search/2	Construct or analyze an HTTP search specification.
set_url_encoding/2	Query and set the encoding for URLs.
url_iri/2	Convert between a URL, encoding in US-ASCII and an IRI.
www_form_encode/2	En/decode to/from application/x-www-form-encoded.

F.2.23 `www_browser`

www_open_url/1	Open a web-page in a browser

F.2.24 `clp/clpfd`

#/\/2	P and Q hold.
#</2	X is less than Y.
#<==/2	Q implies P.
#<==>/2	P and Q are equivalent.
#=/2	X equals Y.
#=</2	X is less than or equal to Y.
#==>/2	P implies Q.
#>/2	X is greater than Y.
#>=/2	X is greater than or equal to Y.
#\/1	The reifiable constraint Q does _not_ hold.
#\//2	P or Q holds.
#\=/2	X is not Y.
all_different/1	Vars are pairwise distinct.
all_distinct/1	Like all_different/1, with stronger propagation.
automaton/3	Equivalent to automaton(_, _, Signature, Nodes, Arcs, [], [], _), a common use case of automaton/8
automaton/8	True if the finite automaton induced by Nodes and Arcs (extended with Counters) accepts Signatu
chain/2	Zs is a list of finite domain variables that are a chain with respect to the partial order Relation, in t
circuit/1	True if the list Vs of finite domain variables induces a Hamiltonian circuit, where the k-th element
cumulative/1	Equivalent to cumulative(Tasks, [limit(1)]).
cumulative/2	Tasks is a list of tasks, each of the form task(S_i, D_i, E_i, C_i, T_i).
element/3	The N-th element of the list of finite domain variables Vs is V.
fd_dom/2	Dom is the current domain (see in/2) of Var.
fd_inf/2	Inf is the infimum of the current domain of Var.
fd_size/2	Size is the number of elements of the current domain of Var, or the atom *sup* if the domain is un
fd_sup/2	Sup is the supremum of the current domain of Var.

fd_var/1	True iff Var is a CLP(FD) variable.
global_cardinality/2	Equivalent to global_cardinality(Vs, Pairs, []).
global_cardinality/3	Vs is a list of finite domain variables, Pairs is a list of Key-Num pairs, where Key is an integer and
in/2	Var is an element of Domain.
indomain/1	Bind Var to all feasible values of its domain on backtracking.
ins/2	The variables in the list Vars are elements of Domain.
label/1	Equivalent to labeling([], Vars).
labeling/2	Labeling means systematically trying out values for the finite domain variables Vars until all of th
lex_chain/1	Lists are lexicographically non-decreasing.
scalar_product/4	Cs is a list of integers, Vs is a list of variables and integers.
serialized/2	Constrain a set of intervals to a non-overlapping sequence.
sum/3	The sum of elements of the list Vars is in relation Rel to Expr, where Rel is #=, #\=, `#<`, `#>`, `#=<`
transpose/2	Transpose a list of lists of the same length.
tuples_in/2	Relation must be a list of lists of integers.
zcompare/3	Analogous to compare/3, with finite domain variables A and B.

F.2.25 `clpqr`

entailed/1	Check if constraint is entailed
inf/2	Find the infimum of an expression
sup/2	Find the supremum of an expression
minimize/1	Minimizes an expression
maximize/1	Maximizes an expression
bb_inf/3	Infimum of expression for mixed-integer problems
bb_inf/4	Infimum of expression for mixed-integer problems
bb_inf/5	Infimum of expression for mixed-integer problems
dump/3	Dump constraints on variables

F.2.26 `clp/simplex`

assignment/2	Solve assignment problem
constraint/3	Add linear constraint to state
constraint/4	Add named linear constraint to state
constraint_add/4	Extend a named constraint
gen_state/1	Create empty linear program
maximize/3	Maximize objective function in to linear constraints
minimize/3	Minimize objective function in to linear constraints
objective/2	Fetch value of objective function
shadow_price/3	Fetch shadow price in solved state
transportation/4	Solve transportation problem
variable_value/3	Fetch value of variable in solved state

F.2.27 `thread_pool`

current_thread_pool/1	True if Name refers to a defined thread pool.
thread_create_in_pool/4	Create a thread in Pool.

thread_pool_create/3	Create a pool of threads.
thread_pool_destroy/1	Destroy the thread pool named Name.
thread_pool_property/2	True if Property is a property of thread pool Name.

F.2.28 `varnumbers`

max_var_number/3	True when Max is the max of Start and the highest numbered $VAR(N) term.
numbervars/1	Number variables in Term using $VAR(N).
varnumbers/2	Inverse of numbervars/1.
varnumbers/3	Inverse of numbervars/3.

F.3 Arithmetic Functions

*/2	Multiplication
**/2	Power function
+/1	Unary plus (No-op)
+/2	Addition
−/1	Unary minus
−/2	Subtraction
//2	Division
///2	Integer division
/\/2	Bitwise and
<</2	Bitwise left shift
>>/2	Bitwise right shift
./2	List of one character: character code
\/1	Bitwise negation
\//2	Bitwise or
^/2	Power function
abs/1	Absolute value
acos/1	Inverse (arc) cosine
asin/1	Inverse (arc) sine
atan/1	Inverse (arc) tangent
atan/2	Rectangular to polar conversion
atan2/2	Rectangular to polar conversion
ceil/1	Smallest integer larger than arg
ceiling/1	Smallest integer larger than arg
cos/1	Cosine
cputime/0	Get CPU time
div/2	Integer division
e/0	Mathematical constant
epsilon/0	Floating point precision
eval/1	Evaluate term as expression
exp/1	Exponent (base e)
float/1	Explicitly convert to float
float_fractional_part/1	Fractional part of a float
float_integer_part/1	Integer part of a float
floor/1	Largest integer below argument
gcd/2	Greatest common divisor
integer/1	Round to nearest integer
log/1	Natural logarithm
log10/1	10 base logarithm
lsb/1	Least significant bit
max/2	Maximum of two numbers
min/2	Minimum of two numbers
msb/1	Most significant bit
mod/2	Remainder of division
powm/3	Integer exponent and modulo

random/1	Generate random number
random_float/0	Generate random number
rational/1	Convert to rational number
rationalize/1	Convert to rational number
rdiv/2	Ration number division
rem/2	Remainder of division
round/1	Round to nearest integer
truncate/1	Truncate float to integer
pi/0	Mathematical constant
popcount/1	Count 1s in a bitvector
sign/1	Extract sign of value
sin/1	Sine
sqrt/1	Square root
tan/1	Tangent
xor/2	Bitwise exclusive or

F.4 Operators

$	1	fx	Bind top-level variable
^	200	xfy	Predicate
^	200	xfy	Arithmetic function
mod	300	xfx	Arithmetic function
*	400	yfx	Arithmetic function
/	400	yfx	Arithmetic function
//	400	yfx	Arithmetic function
<<	400	yfx	Arithmetic function
>>	400	yfx	Arithmetic function
xor	400	yfx	Arithmetic function
+	500	fx	Arithmetic function
-	500	fx	Arithmetic function
?	500	fx	XPCE: obtainer
\	500	fx	Arithmetic function
+	500	yfx	Arithmetic function
-	500	yfx	Arithmetic function
/\	500	yfx	Arithmetic function
\/	500	yfx	Arithmetic function
:	600	xfy	module:term separator
<	700	xfx	Predicate
=	700	xfx	Predicate
=..	700	xfx	Predicate
=:=	700	xfx	Predicate
<	700	xfx	Predicate
==	700	xfx	Predicate
=@=	700	xfx	Predicate
=\=	700	xfx	Predicate
>	700	xfx	Predicate
>=	700	xfx	Predicate
@<	700	xfx	Predicate
@=<	700	xfx	Predicate
@>	700	xfx	Predicate
@>=	700	xfx	Predicate
is	700	xfx	Predicate
\=	700	xfx	Predicate
\==	700	xfx	Predicate
=@=	700	xfx	Predicate
not	900	fy	Predicate
\+	900	fy	Predicate
,	1000	xfy	Predicate
->	1050	xfy	Predicate
*->	1050	xfy	Predicate
;	1100	xfy	Predicate
|	1105	xfy	Predicate

discontiguous	1150	*fx*	Predicate
dynamic	1150	*fx*	Predicate
module_transparent	1150	*fx*	Predicate
meta_predicate	1150	*fx*	Head
multifile	1150	*fx*	Predicate
thread_local	1150	*fx*	Predicate
volatile	1150	*fx*	Predicate
initialization	1150	*fx*	Predicate
`:-`	1200	*fx*	Introduces a directive
`?-`	1200	*fx*	Introduces a directive
`-->`	1200	*xfx*	DCGrammar: rewrite
`:-`	1200	*xfx*	head **`:-`** body. separator

Bibliography

[Anjewierden & Wielemaker, 1989] A. Anjewierden and J. Wielemaker. Extensible objects. ESPRIT Project 1098 Technical Report UvA-C1-TR-006a, University of Amsterdam, March 1989.

[BIM, 1989] BIM sa/nv, Everberg, Belgium. *BIM Prolog release 2.4*, 1989.

[Bowen *et al.*, 1983] D. L. Bowen, L. M. Byrd, and WF. Clocksin. A portable Prolog compiler. In L. M. Pereira, editor, *Proceedings of the Logic Programming Workshop 1983*, Lisabon, Portugal, 1983. Universidade nova de Lisboa.

[Bratko, 1986] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1986.

[Butenhof, 1997] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Reading, MA, USA, 1997.

[Byrd, 1980] L. Byrd. Understanding the control flow of Prolog programs. *Logic Programming Workshop*, 1980.

[Clocksin & Melish, 1987] W. F. Clocksin and C. S. Melish. *Programming in Prolog*. Springer-Verlag, New York, Third, Revised and Extended edition, 1987.

[Demoen, 2002] Bart 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.

[Deransart *et al.*, 1996] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.

[Frühwirth,] T. Frühwirth. Thom Fruehwirth's constraint handling rules website. http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/chr-intro.html.

[Frühwirth, 1998] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.

[Graham *et al.*, 1982] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[Hodgson, 1998]	Jonathan Hodgson. validation suite for conformance with part 1 of the standard, 1998, `http://www.sju.edu/~jhodgson/pub/suite.tar.gz`.
[Holzbaur, 1992]	Christian Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In *PLILP*, volume 631, pages 260–268. Springer-Verlag, 1992. LNCS 631.
[Kernighan & Ritchie, 1978]	B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
[Neumerkel, 1993]	Ulrich Neumerkel. The binary WAM, a simplified Prolog engine. Technical report, Technische Universität Wien, 1993. http://www.complang.tuwien.ac.at/ulrich/papers/PDF/binwam-nov93.pdf.
[O'Keefe, 1990]	R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Massachussetts, 1990.
[Pereira, 1986]	F. Pereira. *C-Prolog User's Manual*. EdCaad, University of Edinburgh, 1986.
[Qui, 1997]	AI International ltd., Berkhamsted, UK. *Quintus Prolog, User Guide and Reference Manual*, 1997.
[Sterling & Shapiro, 1986]	L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.