Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.
Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.
Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.
# Table of Contents

**Introduction** ................................................. 1

1 **Installing YAP** ............................................. 3
   1.1 Tuning the Functionality of YAP .......................... 3
   1.2 Tuning YAP for a Particular Machine and Compiler .... 4
   1.3 Tuning YAP for GCC ........................................ 4
      1.3.1 Compiling Under Visual C++ ......................... 6
      1.3.2 Compiling Under SGI's cc ............................ 7

2 **Running YAP** ................................................. 9
   2.1 Running YAP Interactively ................................. 9
   2.2 Running Prolog Files ...................................... 10

3 **Syntax** ......................................................... 13
   3.1 Syntax of Terms ........................................... 13
   3.2 Prolog Tokens .............................................. 14
      3.2.1 Numbers .............................................. 14
         3.2.1.1 Integers ......................................... 14
         3.2.1.2 Floating-point Numbers ......................... 15
      3.2.2 Character Strings .................................. 15
      3.2.3 Atoms ............................................... 16
      3.2.4 Variables .......................................... 16
      3.2.5 Punctuation Tokens ................................. 17
      3.2.6 Layout ............................................. 17
   3.3 Wide Character Support .................................. 17
      3.3.1 Wide character encodings on streams ............. 17
      3.3.2 BOM: Byte Order Mark ................................ 19

4 **Loading Programs** .......................................... 21
   4.1 Program loading and updating ............................ 21
   4.2 Changing the Compiler’s Behavior ....................... 23
   4.3 Conditional Compilation ................................. 26
   4.4 Saving and Loading Prolog States ....................... 26

5 **The Module System** ......................................... 29
   5.1 Module Concepts .......................................... 29
   5.2 Defining a New Module .................................. 30
   5.3 Using Modules .......................................... 30
   5.4 Meta-Predicates in Modules .............................. 31
   5.5 Re-Exporting Predicates From Other Modules .......... 32
# 6 Built-In Predicates

6.1 Control Predicates ............................................ 33
6.2 Handling Undefined Procedures .............................. 37
6.3 Message Handling .............................................. 39
6.4 Predicates on terms ............................................ 40
6.5 Predicates on Atoms ............................................ 42
6.6 Predicates on Characters ...................................... 44
6.7 Comparing Terms ................................................ 46
6.8 Arithmetic ....................................................... 48
6.9 I/O Predicates ................................................... 53
   6.9.1 Handling Streams and Files ............................... 53
   6.9.2 Handling Streams and Files ............................... 58
   6.9.3 Handling Input/Output of Terms ......................... 59
   6.9.4 Handling Input/Output of Characters .................... 65
   6.9.5 Input/Output Predicates applied to Streams ........... 66
6.9.6 Compatible C-Prolog predicates for Terminal I/O ........ 68
6.9.7 Controlling Input/Output .................................. 68
6.9.8 Using Sockets From YAP ................................... 69
6.10 Using the Clausal Data Base ................................ 71
   6.10.1 Modification of the Data Base ......................... 71
   6.10.2 Looking at the Data Base ............................... 73
   6.10.3 Using Data Base References ............................ 74
6.11 Internal Data Base ............................................. 75
6.12 The Blackboard ............................................... 77
6.13 Collecting Solutions to a Goal .............................. 78
6.14 Grammar Rules ................................................ 79
6.15 Access to Operating System Functionality .................. 81
6.16 Term Modification ............................................. 83
6.17 Global Variables .............................................. 84
6.18 Profiling Prolog Programs .................................... 86
   6.18.1 The Count Profiler ....................................... 87
   6.18.2 Tick Profiler .............................................. 88
6.19 Counting Calls ................................................. 88
6.20 Arrays .......................................................... 89
6.21 Predicate Information ........................................ 92
6.22 Miscellaneous .................................................. 92

# 7 Library Predicates

7.1 Aggregate ...................................................... 105
7.2 Apply Macros ..................................................... 107
7.3 Association Lists .............................................. 107
7.4 AVL Trees ........................................................ 108
7.5 Heaps ............................................................. 109
7.6 List Manipulation ............................................... 110
7.7 Line Manipulation Utilities .................................... 112
7.8 Maplist ............................................................ 113
7.9 Matrix Library .................................................. 115
7.10 MATLAB Package Interface ................................. 118
7.11 Non-Backtrackable Data Structures ....................... 120
7.12 Ordered Sets............................................ 121
7.13 Pseudo Random Number Integer Generator .............. 122
7.14 Queues .................................................. 123
7.15 Random Number Generator ............................. 124
7.16 Read Utilities.......................................... 124
7.17 Red-Black Trees ....................................... 125
7.18 Regular Expressions .................................... 127
7.19 SWI-Prolog's shlib library ............................. 129
7.20 Splay Trees .............................................. 130
7.21 Reading From and Writing To Strings .................. 131
7.22 Calling The Operating System from YAP ............... 132
7.23 Utilities On Terms ...................................... 136
7.24 Trie DataStructure ..................................... 137
7.25 Call Cleanup ........................................... 138
7.26 Calls With Timeout .................................... 140
7.27 Updatable Binary Trees ................................ 140
7.28 Unweighted Graphs .................................... 141
7.29 Directed Graphs ........................................ 144
7.30 Undirected Graphs ...................................... 146
7.31 LAM ...................................................... 147

8  SWI-Prolog Emulation ...................................... 149
8.1 Invoking Predicates on all Members of a List .......... 150
8.2 Forall ..................................................... 150

9  SWI Global variables ...................................... 151
9.1 Compatibility of Global Variables ...................... 152

10 Extensions to Prolog ....................................... 153
10.1 Rational Trees .......................................... 153
10.2 Co-routining ............................................. 153

11 Attributed Variables ..................................... 155
11.1 hProlog and SWI-Prolog style Attribute Declarations .... 155
11.2 SICStus Prolog style Attribute Declarations .......... 158
11.2.1 Attribute Declarations ............................. 158
11.2.2 Attribute Manipulation ............................. 159
11.2.3 Attributed Unification .............................. 159
11.2.4 Displaying Attributes .............................. 160
11.2.5 Projecting Attributes .............................. 160
11.2.6 Attribute Examples ................................. 160
# Constraint Logic Programming over Reals

- 12.1 Solver Predicates .................................................. 165
- 12.2 Syntax of the predicate arguments ................................. 166
- 12.3 Use of unification ..................................................... 167
- 12.4 Non-Linear Constraints ............................................... 167

# CHR: Constraint Handling Rules ........................................ 169

- 13.1 Introduction ............................................................. 169
- 13.2 Syntax and Semantics ................................................ 169
  - 13.2.1 Syntax .............................................................. 169
  - 13.2.2 Semantics ......................................................... 170
    - Rule Types .................................................................. 171
    - Rule Names .................................................................. 171
    - Pragmas ...................................................................... 171
    - Options ....................................................................... 171
- 13.3 CHR in YAP Programs .................................................. 173
- 13.3.1 Embedding in Prolog Programs ................................... 173
- 13.3.2 Constraint declaration ............................................. 173
- 13.3.3 Compilation ............................................................ 173
- 13.4 Debugging .................................................................. 173
- 13.4.1 Ports ..................................................................... 174
- 13.4.2 Tracing .................................................................. 174
- 13.4.3 CHR Debugging Predicates ....................................... 175
- 13.5 Examples .................................................................... 175
- 13.6 Compatibility with SICStus CHR ................................. 176
- 13.7 Guidelines ................................................................. 176

# Logtalk ............................................................................. 179

# Threads ............................................................................. 181

- 15.1 Creating and Destroying Prolog Threads ......................... 181
- 15.2 Monitoring Threads ..................................................... 183
- 15.3 Thread communication ................................................ 184
  - 15.3.1 Message Queues .................................................... 184
  - 15.3.2 Signalling Threads .................................................. 186
  - 15.3.3 Threads and Dynamic Predicates ............................... 187
- 15.4 Thread Synchronisation ............................................... 187

# Parallelism ...................................................................... 191

# Tabling ............................................................................. 193

# Tracing at Low Level .......................................................... 195
19 Profiling the Abstract Machine ............ 197

20 Debugging ........................................ 199
  20.1 Debugging Predicates ......................... 199
  20.2 Interacting with the debugger ............... 200

21 Efficiency Considerations ...................... 203
  21.1 Deterministic Programs ....................... 203
  21.2 Non-Deterministic Programs ................. 203
  21.3 Data-Base Operations ....................... 203
  21.4 Indexing ..................................... 203
  21.5 Profiling .................................... 203

22 C Language interface to YAP ................. 207
  22.1 Terms ........................................ 208
  22.2 Unification .................................. 211
  22.3 Strings ....................................... 212
  22.4 Memory Allocation ............................ 212
  22.5 Controlling YAP Streams from C ............. 212
  22.6 Utility Functions in C ....................... 213
  22.7 From C back to Prolog ....................... 214
  22.8 Module Manipulation in C ................... 216
  22.9 Writing predicates in C ..................... 217
  22.10 Loading Object Files ....................... 219
  22.11 Saving and Restoring ....................... 220
  22.12 Changes to the C-Interface in YAP4 ........ 220

23 Using YAP as a Library ......................... 221

24 Compatibility with Other Prolog systems
  .................................................. 225
  24.1 Compatibility with the C-Prolog interpreter 225
    24.1.1 Major Differences between YAP and C-Prolog. 225
    24.1.2 YAP predicates fully compatible with C-Prolog ... 225
    24.1.3 YAP predicates not strictly compatible with C-Prolog . 227
    24.1.4 YAP predicates not available in C-Prolog .......... 227
    24.1.5 YAP predicates not available in C-Prolog ........... 235
  24.2 Compatibility with the Quintus and SICStus Prolog systems
    .................................................. 235
    24.2.1 Major Differences between YAP and SICStus Prolog.. 235
    24.2.2 YAP predicates fully compatible with SICStus Prolog .. 237
    24.2.3 YAP predicates not strictly compatible with SICStus Prolog
      .................................................. 242
    24.2.4 YAP predicates not available in SICStus Prolog ...... 243
  24.3 Compatibility with the ISO Prolog standard ......... 247
Appendix A   Summary of YAP Predefined Operators ................................. 249

Predicate Index ......................................................... 251

Concept Index............................................................. 263
Introduction

This document provides User information on version 6.0.6 of YAP (Yet Another Prolog). The YAP Prolog System is a high-performance Prolog compiler developed at LIACC, Universidade do Porto. YAP provides several important features:

- Speed: YAP is widely considered one of the fastest available Prolog systems.
- Functionality: it supports stream I/O, sockets, modules, exceptions, Prolog debugger, C-interface, dynamic code, internal database, DCGs, saved states, co-routining, arrays, threads.
- We explicitly allow both commercial and non-commercial use of YAP.

YAP is based on the David H. D. Warren’s WAM (Warren Abstract Machine), with several optimizations for better performance. YAP follows the Edinburgh tradition, and was originally designed to be largely compatible with DEC-10 Prolog, Quintus Prolog, and especially with C-Prolog.

YAP implements most of the ISO-Prolog standard. We are striving at full compatibility, and the manual describes what is still missing. The manual also includes a (largely incomplete) comparison with SICStus Prolog.

The document is intended neither as an introduction to Prolog nor to the implementation aspects of the compiler. A good introduction to programming in Prolog is the book The Art of Prolog, by L. Sterling and E. Shapiro, published by "The MIT Press, Cambridge MA". Other references should include the classical Programming in Prolog, by W.F. Clocksin and C.S. Mellish, published by Springer-Verlag.

YAP 4.3 is known to build with many versions of gcc (<= gcc-2.7.2, >= gcc-2.8.1, >= egcs-1.0.1, gcc-2.95.*) and on a variety of Unixen: SunOS 4.1, Solaris 2.*, Irix 5.2, HP-UX 10, Dec Alpha Unix, Linux 1.2 and Linux 2.* (RedHat 4.0 thru 5.2, Debian 2.*) in both the x86 and alpha platforms. It has been built on Windows NT 4.0 using Cygwin from Cygnus Solutions (see README.nt) and using Visual C++ 6.0.

The overall copyright and permission notice for YAP4.3 can be found in the Artistic file in this directory. YAP follows the Perl Artistic license, and it is thus non-copylefted freeware.

If you have a question about this software, desire to add code, found a bug, want to request a feature, or wonder how to get further assistance, please send e-mail to yap-users AT lists.sourceforge.net. To subscribe to the mailing list, visit the page https://lists.sourceforge.net/lists/listinfo/yap-users.

On-line documentation is available for YAP at:

http://www.ncc.up.pt/~vsc/YAP/

Recent versions of YAP, including both source and selected binaries, can be found from this same URL.

This manual was written by Vítor Santos Costa, Luís Damas, Rogério Reis, and Rúben Azevedo. The manual is largely based on the DECsystem-10 Prolog User’s Manual by D.L. Bowen, L. Byrd, F. C. N. Pereira, L. M. Pereira, and D. H. D. Warren. We have also used comments from the Edinburgh Prolog library written by R. O’Keefe. We would also like to gratefully acknowledge the contributions from Ashwin Srinivasan.
We are happy to include in YAP several excellent packages developed under separate licenses. Our thanks to the authors for their kind authorization to include these packages.

The packages are, in alphabetical order:

- The CHR package developed by Tom Schrijvers, Christian Holzbaur, and Jan Wielemaker.
- The CLP(R) package developed by Leslie De Koninck, Bart Demoen, Tom Schrijvers, and Jan Wielemaker, based on the CLP(Q,R) implementation by Christian Holzbaur.
- The Logtalk Object-Oriented system is developed at the University of Beira Interior, Portugal, by Paulo Moura:
  http://logtalk.org/
  Logtalk is no longer distributed with YAP. Please use the Logtalk standalone installer for a smooth integration with YAP.
- The Pillow WEB library developed at Universidad Politecnica de Madrid by the CLIP group. This package is distributed under the FSF’s LGPL. Documentation on this package is distributed separately from yap.tex.
- The yap2swi library implements some of the functionality of SWI’s PL interface. Please do refer to the SWI-Prolog home page:
  http://www.swi-prolog.org
  for more information on SWI-Prolog and for a detailed description of its foreign language interface.
1 Installing YAP

To compile YAP it should be sufficient to:

1. mkdir ARCH.
2. cd ARCH.
3. ../configure ...options....
   Notice that by default configure gives you a vanilla configuration. For instance, in order to use co-routining and/or CLP you need to do
   
   ../configure --enable-coroutining ...options....
   
   Please see Section 1.1 [Configuration Options], page 3 for extra options.
4. check the Makefile for any extensions or changes you want to make.

YAP uses autoconf. Recent versions of YAP try to follow GNU conventions on where to place software.

- The main executable is placed at BINDIR. This executable is actually a script that calls the Prolog engine, stored at LIBDIR.
- LIBDIR is the directory where libraries are stored. YAPLIBDIR is a subdirectory that contains the Prolog engine and a Prolog library.
- INCLUDEDIR is used if you want to use YAP as a library.
- INFODIR is where to store info files. Usually /usr/local/info, /usr/info, or /usr/share/info.

5. make.
6. If the compilation succeeds, try ./yap.
7. If you feel satisfied with the result, do make install.
8. make install-info will create the info files in the standard info directory.
9. make html will create documentation in html format in the predefined directory.

In most systems you will need to be superuser in order to do make install and make info on the standard directories.

1.1 Tuning the Functionality of YAP

Compiling YAP with the standard options give you a plain vanilla Prolog. You can tune YAP to include extra functionality by calling configure with the appropriate options:

- --enable-rational-trees=yes gives you support for infinite rational trees.
- --enable-coroutining=yes gives you support for coroutining, including freezing of goals, attributed variables, and constraints. This will also enable support for infinite rational trees.
- --enable-depth-limit=yes allows depth limited evaluation, say for implementing iterative deepening.
- --enable-low-level-tracer=yes allows support for tracing all calls, retries, and backtracks in the system. This can help in debugging your application, but results in performance loss.
- --enable-wam-profile=yes allows profiling of abstract machine instructions. This is useful when developing YAP, should not be so useful for normal users.
• `--enable-condor=yes` allows using the Condor system that support High Throughput Computing (HTC) on large collections of distributively owned computing resources.
• `--enable-tabling=yes` allows tabling support. This option is still experimental.
• `--enable-parallelism={env-copy,sba,a-cow}` allows or-parallelism supported by one of these three forms. This option is still highly experimental.
• `--with-max-workers` allows definition of the maximum number of parallel processes (its value can be consulted at runtime using the flag `max_workers`).
• `--with-gmp[=DIR]` give a path to where one can find the GMP library if not installed in the default path.
• `--enable-threads` allows using of the multi-threading predicates provided by YAP. Depending on the operating system, the option `--enable-pthread-locking` may also need to be used.
• `--with-max-threads` allows definition of the maximum number of threads (the default value is 1024; its value can be consulted at runtime using the flag `max_threads`).

Next section discusses machine dependent details.

1.2 Tuning YAP for a Particular Machine and Compiler

The default options should give you best performance under GCC. Although the system is tuned for this compiler we have been able to compile versions of YAP under lcc in Linux, Sun’s cc compiler, IBM’s xlc, SGI’s cc, and Microsoft’s Visual C++ 6.0.

1.3 Tuning YAP for GCC.

YAP has been developed to take advantage of GCC (but not to depend on it). The major advantage of GCC is threaded code and explicit register reservation.

YAP is set by default to compile with the best compilation flags we know. Even so, a few specific options reduce portability. The option

• `--enable-max-performance=yes` will try to support the best available flags for a specific architectural model. Currently, the option assumes a recent version of GCC.
• `--enable-debug-yap` compiles YAP so that it can be debugged by tools such as `dbx` or `gdb`.

Here follow a few hints:

On x86 machines the flags:

```
YAP_EXTRAS= ... -DBP_FREE=1
```

tells us to use the `%bp` register (frame-pointer) as the emulator’s program counter. This seems to be stable and is now default.

On Sparc/Solaris2 use:

```
YAP_EXTRAS= ... -mno-app-reg -DOPTIMISE_ALL_REGS_FOR_SPARC=1
```

and YAP will get two extra registers! This trick does not work on SunOS 4 machines.

Note that versions of GCC can be tweaked to recognize different processors within the same instruction set, e.g. 486, Pentium, and PentiumPro for the x86; or Ultrasparc, and Supersparc for Sparc. Unfortunately, some of these tweaks do may make YAP run slower or not at all in other machines with the same instruction set, so they cannot be made default.
Last, the best options also depends on the version of GCC you are using, and it is a good idea to consult the GCC manual under the menus "Invoking GCC"/"Submodel Options". Specifically, you should check `--march=XXX` for recent versions of GCC/EGCS. In the case of GCC2.7 and other recent versions of GCC you can check:

486: In order to take advantage of 486 specific optimizations in GCC 2.7.4:

```
YAP_EXTRAS= ... -m486 -DBP_FREE=1
```

Pentium:

```
YAP_EXTRAS= ... -m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2
```

PentiumPro and other recent Intel and AMD machines:

PentiumPros are known not to require alignment. Check your version of GCC for the best `--march` option.

Super and UltraSparcs:

```
YAP_EXTRAS= ... -msupersparc
```

MIPS: if have a recent machine and you need a 64 bit wide address
space you can use the abi 64 bits or eabi option, as in:

```
CC="gcc -mabi=64" ./configure --...
```

Be careful. At least for some versions of GCC, compiling with `-g` seems to result in broken code.

WIN32: GCC is distributed in the MINGW32 and CYGWIN packages.
The Mingw32 environment is available from the URL:

http://www.mingw.org
You will need to install the msys and mingw packages. You should be able to do configure, make and make install.

If you use mingw32 you may want to search the contributed packages for the gmp multi-precision arithmetic library. If you do setup YAP with gmp note that libgmp.dll must be in the path, otherwise YAP will not be able to execute.

CygWin environment is available from the URL:

http://www.cygwin.com
and mirrors. We suggest using recent versions of the cygwin shell. The compilation steps under the cygwin shell are as follows:

```
mkdir cyg
$YAPSRC/configure --enable-coroutining \  
   --enable-depth-limit \  
   --enable-max-performance
make
make install
```
By default, YAP will use the `--no-cygwin` option to disable the use of the cygwin dll and to enable the mingw32 subsystem instead. YAP thus will not need the cygwin dll. It instead accesses the system’s CRTDLL.DLL C run time library supplied with Win32 platforms through the mingw32 interface. Note
that some older WIN95 systems may not have CRTDLL.DLL, in this case it should be sufficient to import the file from a newer WIN95 or WIN98 machine. You should check the default installation path which is set to /YAP in the standard Makefile. This string will usually be expanded into c:\YAP by Windows. The cygwin environment does not provide gmp on the MINGW subsystem. You can fetch a dll for the gmp library from http://www.sf.net/projects/mingwrep.

It is also possible to configure YAP to be a part of the cygwin environment. In this case you should use:

```
mkdir cyg
$YAPSRC/configure --enable-max-performance \" --enable-cygwin=yes
make
make install
```

YAP will then compile using the cygwin library and will be installed in cygwin's /usr/local. You can use YAP from a cygwin console, or as a standalone application as long as it can find cygwin1.dll in its path. Note that you may use to use --enable-depth-limit for Aleph compatibility, and that you may want to be sure that GMP is installed.

### 1.3.1 Compiling Under Visual C++

YAP compiles cleanly under Microsoft’s Visual C++ release 6.0. We next give a step-by-step tutorial on how to compile YAP manually using this environment.

First, it is a good idea to build YAP as a DLL:

1. create a project named yapdll using File.New. The project will be a DLL project, initially empty. Notice that either the project is named yapdll or you must replace the preprocessors variable YAPDLL_EXPORTS to match your project names in the files YAPInterface.h and c_interface.c.

2. add all .c files in the $YAPSRC/C directory and in the $YAPSRC/OPTYAP directory to the Project’s Source Files (use FileView).

3. add all .h files in the $YAPSRC/H directory, $YAPSRC\include directory and in the $YAPSRC/OPTYAP subdirectory to the Project’s Header Files.

4. Ideally, you should now use m4 to generate extra .h from .m4 files and use configure to create a config.h. Or, you can be lazy, and fetch these files from $YAPSRC\VC\include.

5. You may want to go to Build.Set Active Configuration and set Project Type to Release.

6. To use YAP’s own include directories you have to set the Project option Project.Project Settings.C/C++.Preprocessor.Additional Include Directories to include the directories $YAPSRC\H, $YAPSRC\VC\include, $YAPSRC\OPTYAP and $YAPSRC\include. The syntax is:

   $YAPSRC\H, $YAPSRC\VC\include, $YAPSRC\OPTYAP, $YAPSRC\include

7. Build: the system should generate an yapdll.dll and an yapdll.lib.
8. Copy the file yapdll.dll to your path. The file yapdll.lib should also be copied to a location where the linker can find it.

Now you are ready to create a console interface for YAP:

1. create a second project say wyap with File.New. The project will be a WIN32 console project, initially empty.
2. add $YAPSRC\console\yap.c to the Source Files.
3. add $YAPSRC\VC\include\config.h and the files in $YAPSRC\include to the Header Files.
4. You may want to go to Build.Set Active Configuration and set Project Type to Release.
5. you will eventually need to bootstrap the system by booting from boot.yap, so write:
   
   -b $YAPSRC\pl\boot.yap
   
6. You need the sockets and yap libraries. Add
   
   ws2_32.lib yapdll.lib to
   
   to Project.Project Settings.Link.Object/Library Modules
   
   You may also need to set the Link Path so that VC++ will find yapdll.lib.
7. set Project.Project Settings.C/C++.Preprocessor.Additional Include Directories to include the $YAPSRC/VC/include and $YAPSRC/include.
   
   The syntax is:
   
   $YAPSRC/VC/include, $YAPSRC/include
8. Build the system.
9. Use Build.Start Debug to boot the system, and then create the saved state with
   
   ['$YAPSRC\pl\init'].
   
   save_program('startup.yss').
   
   That’s it, you’ve got YAP and the saved state!

The $YAPSRC\VC directory has the make files to build YAP4.3.17 under VC++ 6.0.

1.3.2 Compiling Under SGI’s cc

YAP should compile under the Silicon Graphic’s cc compiler, although we advise using the GNUCC compiler, if available.

64 bit Support for 64 bits should work by using (under Bourne shell syntax):

   CC="cc -64" $YAP_SRC_PATH/configure --...
2 Running YAP

We next describe how to invoke YAP in Unix systems.

2.1 Running YAP Interactively

Most often you will want to use YAP in interactive mode. Assuming that YAP is in the user’s search path, the top-level can be invoked under Unix with the following command:

```
yap [-s n] [-h n] [-a n] [-c IP_HOST port ] [filename]
```

All the arguments and flags are optional and have the following meaning:

- `?` print a short error message.
- `-s Size` allocate `Size` K bytes for local and global stacks. The user may specify `M` bytes.
- `-h Size` allocate `Size` K bytes for heap and auxiliary stacks
- `-t Size` allocate `Size` K bytes for the trail stack
- `-L Size` SWI-compatible option to allocate `Size` K bytes for local and global stacks; the local stack cannot be expanded. To avoid confusion with the load option, `Size` must immediately follow the letter L.
- `-G Size` SWI-compatible option to allocate `Size` K bytes for local and global stacks; the global stack cannot be expanded
- `-T Size` SWI-compatible option to allocate `Size` K bytes for the trail stack; the trail cannot be expanded.

- `-l YAP_FILE` compile the Prolog file `YAP_FILE` before entering the top-level.
- `-L YAP_FILE` compile the Prolog file `YAP_FILE` and then halt. This option is useful for implementing scripts.
- `-g Goal` run the goal `Goal` before top-level. The goal is converted from an atom to a Prolog term.
- `-z Goal` run the goal `Goal` as top-level. The goal is converted from an atom to a Prolog term.

- `-b BOOT_FILE` boot code is in Prolog file `BOOT_FILE`. The filename must define the predicate `
`'live'/0
`

- `-c IP_HOST port` connect standard streams to host `IP_HOST` at port `port`

- `filename` restore state saved in the given file
- `-f` do not consult initial files
- `-q` do not print informational messages
- `--` separator for arguments to Prolog code. These arguments are visible through the `unix/1` built-in predicate.
Note that YAP will output an error message on the following conditions:

- a file name was given but the file does not exist or is not a saved YAP state;
- the necessary amount of memory could not be allocated;
- the allocated memory is not enough to restore the state.

When restoring a saved state, YAP will allocate the same amount of memory as that in use when the state was saved, unless a different amount is specified by flags in the command line. By default, YAP restores the file ‘startup.yss’ from the current directory or from the YAP library.

- YAP usually boots from a saved state. The saved state will use the default installation directory to search for the YAP binary unless you define the environment variable YAPBINDIR.
- YAP always tries to find saved states from the current directory first. If it cannot it will use the environment variable YAPLIBDIR, if defined, or search the default library directory.
- YAP will try to find library files from the YAPSHAREDIR/library directory.

### 2.2 Running Prolog Files

YAP can also be used to run Prolog files as scripts, at least in Unix-like environments. A simple example is shown next (do not forget that the shell comments are very important):

```bash
#!/usr/local/bin/yap -L --  
#  
# Hello World script file using YAP  
#  
# put a dot because of syntax errors .  

:- write('Hello World'), nl.
```

The `#!/` characters specify that the script should call the binary file YAP. Notice that many systems will require the complete path to the YAP binary. The `-L` flag indicates that YAP should consult the current file when booting and then halt. The remaining arguments are then passed to YAP. Note that YAP will skip the first lines if they start with `#` (the comment sign for Unix’s shell). YAP will consult the file and execute any commands.

A slightly more sophisticated example is:
Chapter 2: Running YAP

```
#!/usr/bin/yap -L --
#
# Hello World script file using YAP
#
:- initialization(main).
main :- write('Hello World'), nl.
```

The `initialization` directive tells YAP to execute the goal `main` after consulting the file. Source code is thus compiled and `main` executed at the end. The `.` is useful while debugging the script as a Prolog program: it guarantees that the syntax error will not propagate to the Prolog code.

Notice that the `--` is required so that the shell passes the extra arguments to YAP. As an example, consider the following script `dump_args`:

```
#!/usr/bin/yap -L --
#
main( [] ).
main( [H|T] ) :-
    write( H ), nl,
    main( T ).
:- unix( argv(AllArgs) ), main( AllArgs ).
```

If you run this script with the arguments:

```
./dump_args -s 10000
```

the script will start an YAP process with stack size 10MB, and the list of arguments to the process will be empty.

Often one wants to run the script as any other program, and for this it is convenient to ignore arguments to YAP. This is possible by using `L --` as in the next version of `dump_args`:

```
#!/usr/bin/yap -L --
main( [] ).
main( [H|T] ) :-
    write( H ), nl,
    main( T ).
:- unix( argv(AllArgs) ), main( AllArgs ).
```

The `--` indicates the next arguments are not for YAP. Instead, they must be sent directly to the `argv` built-in. Hence, running
./dump_args test
will write test on the standard output.
3 Syntax

We will describe the syntax of YAP at two levels. We first will describe the syntax for Prolog terms. In a second level we describe the tokens from which Prolog terms are built.

3.1 Syntax of Terms

Below, we describe the syntax of YAP terms from the different classes of tokens defined above. The formalism used will be BNF, extended where necessary with attributes denoting integer precedence or operator type.

```
term  ----->  subterm(1200)  end_of_term_marker

subterm(N) ----->  term(M)  [M <= N]

term(N) ----->  op(N, fx) subterm(N-1)
                |  op(N, fy) subterm(N)
                |  subterm(N-1) op(N, xfx) subterm(N-1)
                |  subterm(N-1) op(N, xfy) subterm(N)
                |  subterm(N) op(N, yfx) subterm(N-1)
                |  subterm(N-1) op(N, xf)
                |  subterm(N) op(N, yf)

term(0) ----->  atom '(' arguments ')'
                |  '(' subterm(1200) ')' 
                |  '{' subterm(1200) '}' 
                |  list
                |  string
                |  number
                |  atom
                |  variable

arguments ----->  subterm(999)
                |  subterm(999) ',' arguments

list  ----->  '[]'
            |  '[ ' list_expr ' ]'

list_expr ----->  subterm(999)
                 |  subterm(999) list_tail

list_tail ----->  ',,' list_expr
                 |  ',...' subterm(999)
                 |  '|' subterm(999)
```
Notes:

- \( op(N,T) \) denotes an atom which has been previously declared with type \( T \) and base precedence \( N \).
- Since \( , \) is itself a pre-declared operator with type \( xfy \) and precedence 1000, is subterm starts with a \( , \), \( op \) must be followed by a space to avoid ambiguity with the case of a functor followed by arguments, e.g.:
  
  \[
  + (a,b) \quad \text{[the same as } '+'(',',(a,b)) \text{ of arity one]}
  \]
  versus

  \[
  + (a,b) \quad \text{[the same as } '+'(a,b) \text{ of arity two]}
  \]

- In the first rule for term(0) no blank space should exist between \( atom \) and \( '('. \)
- Each term to be read by the YAP parser must end with a single dot, followed by a blank (in the sense mentioned in the previous paragraph). When a name consisting of a single dot could be taken for the end of term marker, the ambiguity should be avoided by surrounding the dot with single quotes.

### 3.2 Prolog Tokens

Prolog tokens are grouped into the following categories:

#### 3.2.1 Numbers

Numbers can be further subdivided into integer and floating-point numbers.

##### 3.2.1.1 Integers

Integer numbers are described by the following regular expression:

\[
<integer> := \{<digit>+<single-quote>|0\{xXo}\}\alpha_{numeric\_char}+\]

where \{\ldots\} stands for optionality, \(+\) optional repetition (one or more times), \(<digit>\) denotes one of the characters \(0 \ldots 9\), \(1\) denotes or, and \(<single-quote>\) denotes the character "'". The digits before the \(<single-quote>\) character, when present, form the number basis, that can go from 0, 1 and up to 36. Letters from \(A\) to \(Z\) are used when the basis is larger than 10.

Note that if no basis is specified then base 10 is assumed. Note also that the last digit of an integer token can not be immediately followed by one of the characters 'e', 'E', or '}'.

Following the ISO standard, YAP also accepts directives of the form \(0x\) to represent numbers in hexadecimal base and of the form \(0o\) to represent numbers in octal base. For usefulness, YAP also accepts directives of the form \(0X\) to represent numbers in hexadecimal base.

Example: the following tokens all denote the same integer

\[
10 \ 2'1010 \ 3'101 \ 8'12 \ 16'a \ 36'a \ 0xa \ 0o12
\]

Numbers of the form \(0'a\) are used to represent character constants. So, the following tokens denote the same integer:
YAP (version 6.0.6) supports integers that can fit the word size of the machine. This is 32 bits in most current machines, but 64 in some others, such as the Alpha running Linux or Digital Unix. The scanner will read larger or smaller integers erroneously.

### 3.2.1.2 Floating-point Numbers

Floating-point numbers are described by:

\[
<\text{float}> := <\text{digit}>+{<\text{dot}><\text{digit}>+} \\
|<\text{exponent-marker}>({<\text{sign}>}<\text{digit}>+ \\
|<\text{digit}>+{<\text{dot}><\text{digit}>+}
\]

where \(<\text{dot}>\) denotes the decimal-point character \(\cdot\), \(<\text{exponent-marker}>\) denotes one of \('e'\) or \('E'\), and \(<\text{sign}>\) denotes one of \('+\) or \('-\).

Examples:

10.0 10e3 10e-3 3.1415e+3

Floating-point numbers are represented as a double in the target machine. This is usually a 64-bit number.

### 3.2.2 Character Strings

Strings are described by the following rules:

\[
\text{string} \rightarrow '\', \text{string}_\text{quoted}_\text{characters} '"
\]

\[
\text{string}_\text{quoted}_\text{characters} \rightarrow '' , ''\', \text{string}_\text{quoted}_\text{characters} \\
|'\'\', \text{string}_\text{quoted}_\text{characters} \\
|'\\' , \text{escape}_\text{sequence} , \text{string}_\text{quoted}_\text{characters} \\
|'\', \text{string}_\text{character} , \text{string}_\text{quoted}_\text{characters} \\
|'\\' , \text{escape}_\text{sequence} \text{string}_\text{quoted}_\text{characters} \\
|'\', \text{escape}_\text{sequence} \text{string}_\text{quoted}_\text{characters} \\
|'\\' , \text{at}_\text{most}_3_\text{octal}_\text{digit}_\text{seq}_\text{char} '\\' \\
|'\\' , \text{at}_\text{most}_2_\text{hexa}_\text{digit}_\text{seq}_\text{char} '\\'
\]

where \(<\text{string}_\text{character}>\) in any character except the double quote and escape characters.

Examples:

"" "a string" "a double-quote:""

The first string is an empty string, the last string shows the use of double-quoting. The implementation of YAP represents strings as lists of integers. Since YAP 4.3.0 there is no static limit on string size.

Escape sequences can be used to include the non-printable characters \(a\) (alert), \(b\) (backspace), \(r\) (carriage return), \(f\) (form feed), \(t\) (horizontal tabulation), \(n\) (new line), \(v\) (vertical tabulation), \(x\) (null), \(X\) (upper-case null), \(\backslash\) (backslash), \(\\"\) (double quote), \(\\'\) (single quote).
and \ (vertical tabulation). Escape sequences also be include the meta-characters \, ", , ', and '. Last, one can use escape sequences to include the characters either as an octal or hexadecimal number.

The next examples demonstrates the use of escape sequences in YAP:

"\x0c" "\01" "\f" "\\"

The first three examples return a list including only character 12 (form feed). The last example escapes the escape character.

Escape sequences were not available in C-Prolog and in original versions of YAP up to 4.2.0. Escape sequences can be disable by using:

:- yap_flag(character_escapes,off).

### 3.2.3 Atoms

Atoms are defined by one of the following rules:

```prolog
atom --> solo-character
atom --> lower-case-letter name-character*
atom --> symbol-character+
atom --> single-quote single-quote
atom --> ''' atom_quoted_characters '''
```

```prolog
atom_quoted_characters --> ''' ''' atom_quoted_characters
atom_quoted_characters --> '\' atom_sequence string_quoted_characters
atom_quoted_characters --> character string_quoted_characters
```

where:

- `<solo-character>` denotes one of: ! ;
- `<symbol-character>` denotes one of: # & * + - . / : < = > ? @ \ ^ ' ~
- `<lower-case-letter>` denotes one of: a...z
- `<name-character>` denotes one of: _ a...z A...Z 0....9
- `<single-quote>` denotes:

and `<string_character>` denotes any character except the double quote and escape characters. Note that escape sequences in strings and atoms follow the same rules.

Examples:

```
a a12x 'a' ! => '1 2'
```

Version 4.2.0 of YAP removed the previous limit of 256 characters on an atom. Size of an atom is now only limited by the space available in the system.

### 3.2.4 Variables

Variables are described by:

```prolog
<variable-starter><variable-character>+
```

where
<variable-starter> denotes one of: _ A...Z
<variable-character> denotes one of: _ a...z A...Z

If a variable is referred only once in a term, it needs not to be named and one can use the character _ to represent the variable. These variables are known as anonymous variables. Note that different occurrences of _ on the same term represent different anonymous variables.

### 3.2.5 Punctuation Tokens

Punctuation tokens consist of one of the following characters:

( ) , [ ] { } |

These characters are used to group terms.

### 3.2.6 Layout

Any characters with ASCII code less than or equal to 32 appearing before a token are ignored.

All the text appearing in a line after the character % is taken to be a comment and ignored (including %). Comments can also be inserted by using the sequence /* to start the comment and */ to finish it. In the presence of any sequence of comments or layout characters, the YAP parser behaves as if it had found a single blank character. The end of a file also counts as a blank character for this purpose.

### 3.3 Wide Character Support

YAP now implements a SWI-Prolog compatible interface to wide characters and the Universal Character Set (UCS). The following text was adapted from the SWI-Prolog manual.

YAP now 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-bits 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 YAP, following SWI-Prolog, has two representations for atoms. 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 wide chars, which may take 16 or 32 bits. This representational issue is completely transparent to the Prolog user. Users of the foreign language interface 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.

#### 3.3.1 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 influence 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 (note that the Prolog native UTF-8 mode is considerably faster than the generic mbtowc() one). 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 (not yet implemented). For Prolog source files we also provide the encoding/1 directive that can be used to switch between encodings that are compatible to US-ASCII (ascii, iso_latin_1, utf8 and many locales). For additional information and Unicode resources, please visit http://www.unicode.org/.

YAP 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 mbtowc() 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 to 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 open/4 or set_stream/2 (not implemented). Initially the terminal stream write the characters using Prolog escape sequences while other streams generate an I/O exception.
3.3.2 BOM: Byte Order Mark

From Section 3.3.1 [Stream Encoding], page 17, you may have got the impression text-files are complicated. This section deals with a related topic, making live 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 \texttt{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 of as US-ASCII and may contain some encoding mark to switch to UTF-8, such as the \texttt{encoding="UTF-8"} in an XML header. Such formats often explicitly forbid the the use of a UTF-8 BOM. In other cases there is additional information telling the encoding making the use of a BOM redundant or even illegal.

The BOM is handled by the \texttt{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 \texttt{bom(true)} is available through \texttt{stream_property/2}. When opening a file for writing, writing a BOM can be requested using the option \texttt{bom(true)} with \texttt{open/4}. 
4 Loading Programs

4.1 Program loading and updating

consult(+F)
   Adds the clauses written in file F or in the list of files F to the program.
   In YAP consult/1 does not remove previous clauses for the procedures defined in F. Moreover, note that all code in YAP is compiled.

reconsult(+F)
   Updates the program replacing the previous definitions for the predicates defined in F.

[F]
   The same as consult(F).

[-F]
   The same as reconsult(F)

Example:

?- [file1, -file2, -file3, file4].
   will consult file1 file4 and reconsult file2 and file3.

compile(+F)
   In YAP, the same as reconsult/1.

load_files(+Files, +Options)
   General implementation of consult. Execution is controlled by the following flags:

   autoload(+Autoload)
      SWI-compatible option where if Autoload is true predicates are loaded on first call. Currently not supported.

   derived_from(+File)
      SWI-compatible option to control make. Currently not supported.

   encoding(+Encoding)
      Character encoding used in consulting files. Please see Section 3.3 [Encoding], page 17 for supported encodings.

   expand(+Bool)
      Not yet implemented. In SWI-Prolog, 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.

   if(+Condition)
      Load the file only if the specified Condition is satisfied. The value true 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(+ListOrAll)
      If all and the file is a module file, import all public predicates. Otherwise import only the named predicates. Each predicate is
referred to as `<name>/<arity>`. This option has no effect if the file is not a module file.

**must_be_module(+Bool)**

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

**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.

**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.

**compilation_mode(+Mode)**

This extension controls how procedures are compiled. If `Mode` is `compact` clauses are compiled and no source code is stored; if it is `source` clauses are compiled and source code is stored; if it is `assert_all` clauses are asserted into the data-base.

**ensure_loaded(+F) [ISO]**

When the files specified by `F` are module files, `ensure_loaded/1` loads them if they have not been previously loaded, otherwise advertises the user about the existing name clashes and prompts about importing or not those predicates. Predicates which are not public remain invisible.

When the files are not module files, `ensure_loaded/1` loads them if they have not been loaded before, does nothing otherwise.

`F` must be a list containing the names of the files to load.

**make [ISO]**

SWI-Prolog built-in to consult all source files that have been changed since they were consulted. It checks all loaded source files. `make/0` can be combined with the compiler to speed up the development of large packages. In this case compile the package using

```
sun% pl -g make -o my_program -c file ...
```

If `my_program` is started it will first reconsult all source files that have changed since the compilation.

**include(+F) [ISO]**

The `include` directive includes the text files or sequence of text files specified by `F` into the file being currently consulted.
4.2 Changing the Compiler’s Behavior

This section presents a set of built-ins predicates designed to set the environment for the compiler.

**source_mode(-O,+N)**

The state of source mode can either be on or off. When the source mode is on, all clauses are kept both as compiled code and in a "hidden" database. \( O \) is unified with the previous state and the mode is set according to \( N \).

**source**

After executing this goal, YAP keeps information on the source of the predicates that will be consulted. This enables the use of listing/0, listing/1 and clause/2 for those clauses.

The same as source_mode(_,on) or as declaring all newly defined static procedures as public.

**no_source**

The opposite to source.

The same as source_mode(_,off).

**compile_expressions**

After a call to this predicate, arithmetical expressions will be compiled. (see example below). This is the default behavior.

**do_not_compile_expressions**

After a call to this predicate, arithmetical expressions will not be compiled.

```prolog
?- source, do_not_compile_expressions.
yes
?- [user].
| p(X) :- X is 2 * (3 + 8).
| :- end_of_file.
?- compile_expressions.
yes
?- [user].
| q(X) :- X is 2 * (3 + 8).
| :- end_of_file.
:- listing.

p(A):-
    A is 2 * (3 + 8).

q(A):-
    A is 22.
```

**hide(+Atom)**

Make atom *Atom* invisible.

**unhide(+Atom)**

Make hidden atom *Atom* visible.

**hide_predicate(+Pred)**

Make predicate *Pred* invisible to current_predicate/2, listing, and friends.
**expand_exprs(-O,+N)**

Puts YAP in state \(N\) (on or off) and unify \(O\) with the previous state, where \(On\) is equivalent to **compile_expressions** and off is equivalent to **do_not_compile_expressions**. This predicate was kept to maintain compatibility with C-Prolog.

**path(-D)**

Unifies \(D\) with the current directory search-path of YAP. Note that this search-path is only used by YAP to find the files for **consult/1**, **reconsult/1** and **restore/1** and should not be taken for the system search path.

**add_to_path(+D)**

Adds \(D\) to the end of YAP’s directory search path.

**add_to_path(+D,+N)**

Inserts \(D\) in the position, of the directory search path of YAP, specified by \(N\). \(N\) must be either of **first** or **last**.

**remove_from_path(+D)**

Remove \(D\) from YAP’s directory search path.

**style_check(+X)**

Turns on style checking according to the attribute specified by \(X\), which must be one of the following:

- **single_var**
  
  Checks single occurrences of named variables in a clause.

- **discontiguous**
  
  Checks non-contiguous clauses for the same predicate in a file.

- **multiple**
  
  Checks the presence of clauses for the same predicate in more than one file when the predicate has not been declared as **multifile**

- **all**
  
  Performs style checking for all the cases mentioned above.

By default, style checking is disabled in YAP unless we are in **sicstus** or **iso** language mode.

The **style_check/1** built-in is now deprecated. Please use the **set_prolog_flag/1** instead.

**no_style_check(+X)**

Turns off style checking according to the attribute specified by \(X\), which has the same meaning as in **style_check/1**.

The **no_style_check/1** built-in is now deprecated. Please use the **set_prolog_flag/1** instead.

**multifile P [ISO]**

Instructs the compiler about the declaration of a predicate \(P\) in more than one file. It must appear in the first of the loaded files where the predicate is declared, and before declaration of any of its clauses.

Multifile declarations affect **reconsult/1** and **compile/1**: when a multifile predicate is reconsulted, only the clauses from the same file are removed.

Since YAP4.3.0 multifile procedures can be static or dynamic.
discontiguous(+G) [ISO]
Declare that the arguments are discontiguous procedures, that is, clauses for
discontiguous procedures may be separated by clauses from other procedures.

initialization(+G) [ISO]
The compiler will execute goals G after consulting the current file.

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 ini-
tialization/1.

restore Do not execute Goal while loading the program, but only when
restoring a state (not implemented yet).

library_directory(+D)
Succeeds when D is a current library directory name. Library directories are
the places where files specified in the form library(File) are searched by the
predicates consult/1, reconsult/1, use_module/1 or ensure_loaded/1.

file_search_path(+NAME,-DIRECTORY)
Allows writing file names as compound terms. The NAME and DIRECTORY
must be atoms. The predicate may generate multiple solutions. The predicate
is originally defined as follows:

file_search_path(library,A) :-
    library_directory(A).

file_search_path(system,A) :-
    prolog_flag(host_type,A).

Thus, [library(A)] will search for a file using library_directory/1 to obtain
the prefix.

library_directory(+D)
Succeeds when D is a current library directory name. Library directories are
the places where files specified in the form library(File) are searched by the
predicates consult/1, reconsult/1, use_module/1 or ensure_loaded/1.

prolog_file_name(+Name,-FullPath)
Unify FullPath with the absolute path YAP would use to consult file Name.

public P [ISO extension]
Instructs the compiler that the source of a predicate of a list of predicates P
must be kept. This source is then accessible through the clause/2 procedure
and through the listing family of built-ins.
Note that all dynamic procedures are public. The source directive defines all
new or redefined predicates to be public.
Since YAP4.3.0 multifile procedures can be static or dynamic.
4.3 Conditional Compilation

Conditional compilation builds on the same principle as \texttt{term\_expansion/2}, \texttt{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.

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.

\texttt{if(+Goal)}

Compile subsequent code only if \texttt{Goal} succeeds. For enhanced portability, \texttt{Goal} is processed by \texttt{expand\_goal/2} before execution. If an error occurs, the error is printed and processing proceeds as if \texttt{Goal} has failed.

\texttt{else}

Start ‘else’ branch.

\texttt{endif}

End of conditional compilation.

\texttt{elif(+Goal)}

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

\begin{verbatim}
:- if(test1).
  section_1.
:- elif(test2).
  section_2.
:- elif(test3).
  section_3.
:- else.
  section_else.
:- endif.
\end{verbatim}

4.4 Saving and Loading Prolog States

\texttt{save(+F)}

Saves an image of the current state of YAP in file \texttt{F}. From \texttt{YAP4.1.3} onwards, YAP saved states are executable files in the Unix ports.

\texttt{save(+F,-OUT)}

Saves an image of the current state of YAP in file \texttt{F}. From \texttt{YAP4.1.3} onwards, YAP saved states are executable files in the Unix ports.

Unify \texttt{OUT} with 1 when saving the file and \texttt{OUT} with 0 when restoring the saved state.

\texttt{save\_program(+F)}

Saves an image of the current state of the YAP database in file \texttt{F}.
save_program(+F, :G)
Saves an image of the current state of the YAP database in file \( F \), and guarantee that execution of the restored code will start by trying goal \( G \).

restore(+F)
Restores a previously saved state of YAP from file \( F \).
YAP always tries to find saved states from the current directory first. If it cannot it will use the environment variable YAPLIBDIR, if defined, or search the default library directory.
5 The Module System

Module systems are quite important for the development of large applications. YAP implements a module system compatible with the Quintus Prolog module system.

The YAP module system is predicate-based. This means a module consists of a set of predicates (or procedures), such that some predicates are public and the others are local to a module. Atoms and terms in general are global to the system. Moreover, the module system is flat, meaning that we do not support a hierarchy of modules. Modules can automatically import other modules, though. For compatibility with other module systems the YAP module system is non-strict, meaning both that there is a way to access predicates private to a module and that it is possible to declare predicates for a module from some other module.

YAP allows one to ignore the module system if one does not want to use it. Last note that using the module system does not introduce any significant overheads.

5.1 Module Concepts

The YAP module system applies to predicates. All predicates belong to a module. System predicates belong to the module primitives, and by default new predicates belong to the module user. Predicates from the module primitives are automatically visible to every module.

Every predicate must belong to a module. This module is called its source module.

By default, the source module for a clause occurring in a source file with a module declaration is the declared module. For goals typed in a source file without module declarations, their module is the module the file is being loaded into. If no module declarations exist, this is the current type-in module. The default type-in module is user, but one can set the current module by using the built-in module/1.

Note that in this module system one can explicitly specify the source module for a clause by prefixing a clause with its module, say:

\[ \text{user:}(a :- b). \]

In fact, to specify the source module for a clause it is sufficient to specify the source mode for the clause’s head:

\[ \text{user:a :- b}. \]

The rules for goals are similar. If a goal appears in a text file with a module declaration, the goal’s source module is the declared module. Otherwise, it is the module the file is being loaded into or the type-in module.

One can override this rule by prefixing a goal with the module it is supposed to be executed in, say:

\[ \text{nasa:launch(apollo,13)}. \]

will execute the goal \text{launch(apollo,13)} as if the current source module was nasa.

Note that this rule breaks encapsulation and should be used with care.
5.2 Defining a New Module

A new module is defined by a `module` declaration:

```
module(+M,+L)
```

This directive defines the file where it appears as a module file; it must be the first declaration in the file. `M` must be an atom specifying the module name; `L` must be a list containing the module’s public predicates specification, in the form `[predicate_name/arity,...]`.

The public predicates of a module file can be made accessible by other files through the directives `use_module/1`, `use_module/2`, `ensure_loaded/1` and the predicates `consult/1` or `reconsult/1`. The non-public predicates of a module file are not visible by other files; they can, however, be accessed by prefixing the module name with the `:/2` operator.

The built-in `module/1` sets the current source module:

```
module(+M,+L, +Options)
```

Similar to `module/2`, this directive defines the file where it appears in as a module file; it must be the first declaration in the file. `M` must be an atom specifying the module name; `L` must be a list containing the module’s public predicates specification, in the form `[predicate_name/arity,...]`.

The last argument `Options` must be a list of options, which can be:

- `filename` the filename for a module to import into the current module.
- `library(file)` a library file to import into the current module.
- `hide(Opt)` if `Opt` is `false`, keep source code for current module, if `true`, disable.

```
module(+M)
```

Defines `M` to be the current working or type-in module. All files which are not bound to a module are assumed to belong to the working module (also referred to as type-in module). To compile a non-module file into a module which is not the working one, prefix the file name with the module name, in the form `Module:File`, when loading the file.

5.3 Using Modules

By default, all procedures to consult a file will load the modules defined therein. The two following declarations allow one to import a module explicitly. They differ on whether one imports all predicate declared in the module or not.

```
use_module(+F)
```

Loads the files specified by `F`, importing all their public predicates. Predicate name clashes are resolved by asking the user about importing or not the predicate. A warning is displayed when `F` is not a module file.
use_module(+F,+L)
    Loads the files specified by F, importing the predicates specified in the list L. 
    Predicate name clashes are resolved by asking the user about importing or not 
    the predicate. A warning is displayed when F is not a module file.

use_module(?M,?F,+L)
    If module M has been defined, import the procedures in L to the current module. 
    Otherwise, load the files specified by F, importing the predicates specified in 
    the list L.

5.4 Meta-Predicates in Modules

The module system must know whether predicates operate on goals or clauses. Otherwise, 
such predicates would call a goal in the module they were defined, instead of calling it in 
the module they are currently executing. So, for instance, consider a file example.pl:

```prolog
:- module(example,[a/1]).
a(G) :- call(G)
```

We import this module with use_module(example) into module user. The expected 
behavior for a goal a(p) is to execute goal p within the module user. However, a/1 will call p within module example.

The meta_predicate/1 declaration informs the system that some arguments of a pred-
icate are goals, clauses, clauses heads or other terms related to a module, and that these arguments must be prefixed with the current source module:

```prolog
meta_predicate G1,...,Gn
    Each Gi is a mode specification.
    If the argument is : or an integer, the argument is a call and must be ex-
    panded. Otherwise, the argument is not expanded. Note that the system 
    already includes declarations for all built-ins.
    For example, the declaration for call/1 and setof/3 are:
    :- meta_predicate call(,:), setof(?,?,?).
```

The previous example is expanded to the following code which explains, why the goal 
a(p) calls p in example and not in user. The goal call(G) is expanded because of the meta-predicate declaration for call/1.

```prolog
:- module(example,[a/1]).
a(G) :- call(example:G)
```

By adding a meta-predicate declaration for a/1, the goal a(p) in module user will be 
expanded to a(user:p) thereby preserving the module information.

```prolog
:- module(example,[a/1]).
:- meta_predicate a(:).
a(G) :- call(G)
```

An alternate mechanism is the directive module_transparent/1 offered for compatibility 
with SWI-Prolog.
module_transparent +Preds

Preds is a comma separated sequence of name/arity predicate indicators (like dynamic/1). Each goal associated with a transparent declared predicate will inherit the context module from its parent goal.

5.5 Re-Exporting Predicates From Other Modules

It is sometimes convenient to re-export predicates originally defined in a different module. This is often useful if you are adding to the functionality of a module, or if you are composing a large module with several small modules. The following declarations can be used for that purpose:

reexport(+F)

Export all predicates defined in file F as if they were defined in the current module.

reexport(+F,+Decls)

Export predicates defined in file F according to Decls. The declarations may be of the form:

- A list of predicate declarations to be exported. Each declaration may be a predicate indicator or of the form ”PI as NewName”, meaning that the predicate with indicator PI is to be exported under name NewName.
- except(List) In this case, all predicates not in List are exported. Moreover, if ”PI as NewName” is found, the predicate with indicator PI is to be exported under name NewName as before.

Re-exporting predicates must be used with some care. Please, take into account the following observations:

- The reexport declarations must be the first declarations to follow the module declaration.
- It is possible to use both reexport and use_module, but all predicates reexported are automatically available for use in the current module.
- In order to obtain efficient execution, YAP compiles dependencies between re-exported predicates. In practice, this means that changing a reexport declaration and then just recompiling the file may result in incorrect execution.
Chapter 6: Built-In Predicates

6 Built-In Predicates

6.1 Control Predicates

This chapter describes the predicates for controlling the execution of Prolog programs.

In the description of the arguments of functors the following notation will be used:

- a preceding plus sign will denote an argument as an "input argument" - it cannot be a free variable at the time of the call;
- a preceding minus sign will denote an "output argument";
- an argument with no preceding symbol can be used in both ways.

\(+P, +Q\) [ISO]

Conjunction of goals (and).

Example:

\[ p(X) \leftarrow q(X), r(X). \]

should be read as "\(p(X)\) if \(q(X)\) and \(r(X)\)".

\(+P \ ; \ +Q\) [ISO]

Disjunction of goals (or).

Example:

\[ p(X) \leftarrow q(X); r(X). \]

should be read as "\(p(X)\) if \(q(X)\) or \(r(X)\)".

\(\text{true}\) [ISO]

Succeeds once.

\(\text{fail}\) [ISO]

Fails always.

\(\text{false}\)

The same as fail

\(\text{!}\) [ISO]

Read as "cut". Cuts any choices taken in the current procedure. When first found "cut" succeeds as a goal, but if backtracking should later return to it, the parent goal (the one which matches the head of the clause containing the "cut", causing the clause activation) will fail. This is an extra-logical predicate and cannot be explained in terms of the declarative semantics of Prolog.

Example:

\[
\begin{align*}
\text{member}(X, [X|\_]). \\
\text{member}(X, [\_|L]) & \leftarrow \text{member}(X,L).
\end{align*}
\]

With the above definition

\[?- \text{member}(X, [1,2,3]).\]

will return each element of the list by backtracking. With the following definition:

\[
\begin{align*}
\text{member}(X, [X|\_]) & \leftarrow !. \\
\text{member}(X, [\_|L]) & \leftarrow \text{member}(X,L).
\end{align*}
\]

the same query would return only the first element of the list, since backtracking could not "pass through" the cut.
\+ +P [ISO]
Goal \(P\) is not provable. The execution of this predicate fails if and only if the goal \(P\) finitely succeeds. It is not a true logical negation, which is impossible in standard Prolog, but "negation-by-failure".
This predicate might be defined as:
\[
\begin{align*}
\ +(P) & : - P, !, \text{fail}. \\
\ +(_) & .
\end{align*}
\]
if \(P\) did not include "cuts".

not +P
Goal \(P\) is not provable. The same as \(\comb\ P\).
This predicate is kept for compatibility with C-Prolog and previous versions of YAP. Uses of \texttt{not/1} should be replaced by \(\texttt{(+/1)}\), as YAP does not implement true negation.

+P -> +Q [ISO]
Read as "if-then-else" or "commit". This operator is similar to the conditional operator of imperative languages and can be used alone or with an else part as follows:
\[
\begin{align*}
+P & -> +Q \hspace{1cm} \text{"if } P \text{ then } Q\text{"}. \\
+P & -> +Q; +R \hspace{1cm} \text{"if } P \text{ then } Q \text{ else } R\text{"}.
\end{align*}
\]
These two predicates could be defined respectively in Prolog as:
\[
(P -> Q) :- P, !, Q.
\]
and
\[
(P -> Q; R) :- P, !, Q. \\
(P -> Q; R) :- R.
\]
if there were no "cuts" in \(P, Q\) and \(R\).
Note that the commit operator works by "cutting" any alternative solutions of \(P\).
Note also that you can use chains of commit operators like:
\[
P -> Q ; R -> S ; T.
\]
Note that \(\texttt{(-)/2}\) does not affect the scope of cuts in its arguments.

+Condition *-+Action ; +Else
This construct implements the so-called soft-cut. The control is defined as follows: If \textit{Condition} succeeds at least once, the semantics is the same as \(\texttt{(Condition, Action)}\). If \textit{Condition} does not succeed, the semantics is that of \(\texttt{(\+Condition, Else)}\). In other words, If \textit{Condition} succeeds at least once, simply behave as the conjunction of \textit{Condition} and \textit{Action}, otherwise execute \textit{Else}.
The construct \(A *-+ B\), i.e. without an \textit{Else} branch, is translated as the normal conjunction \(A, B\).

repeat [ISO]
Succeeds repeatedly.
In the next example, \texttt{repeat} is used as an efficient way to implement a loop.
The next example reads all terms in a file:
a :- repeat, read(X), write(X), nl, X=end_of_file, !.

the loop is effectively terminated by the cut-goal, when the test-goal X=end
succeeds. While the test fails, the goals read(X), write(X), and nl are executed
repeatedly, because backtracking is caught by the repeat goal.

The built-in repeat/1 could be defined in Prolog by:

    repeat.
    repeat :- repeat.

call(+P) [ISO]
If P is instantiated to an atom or a compound term, the goal call(P) is
executed as if the value of P was found instead of the call to call/1, except
that any "cut" occurring in P only cuts alternatives in the execution of P.

incore(+P)
The same as call/1.

call(+Closure,...,?Ai,...)
Meta-call where Closure is a closure that is converted into a goal by appending
the Ai additional arguments. The number of arguments varies between 0 and
10.

call_with_args(+Name,...,?Ai,...)
Meta-call where Name is the name of the procedure to be called and the Ai are
the arguments. The number of arguments varies between 0 and 10.

If Name is a complex term, then call_with_args/n behaves as call/n:

    call(p(X1,...,Xm), Y1,...,Yn) :- p(X1,...,Xm,Y1,...,Yn).

+P
The same as call(P). This feature has been kept to provide compatibility
with C-Prolog. When compiling a goal, YAP generates a call(X) whenever a
variable X is found as a goal.

    a(X) :- X.

is converted to:

    a(X) :- call(X).

if(?G,?H,?I)
Call goal H once per each solution of goal H. If goal H has no solutions, call
goal I.

The built-in if/3 is similar to ->/3, with the difference that it will backtrack
over the test goal. Consider the following small data-base:

    a(1). b(a). c(x).
    a(2). b(b). c(y).

Execution of an if/3 query will proceed as follows:

    ?- if(a(X),b(Y),c(Z)).

    X = 1,
    Y = a ? ;

    X = 1,
\begin{verbatim}
Y = b ? ;
X = 2,
Y = a ? ;
X = 2,
Y = b ? ;
\end{verbatim}

no
The system will backtrack over the two solutions for \texttt{a/1} and the two solutions for \texttt{b/1}, generating four solutions.

Cuts are allowed inside the first goal \texttt{G}, but they will only prune over \texttt{G}.
If you want \texttt{G} to be deterministic you should use if-then-else, as it is both more efficient and more portable.

\texttt{once(:G) [ISO]}
Execute the goal \texttt{G} only once. The predicate is defined by:
\begin{verbatim}
\texttt{once(G) :- call(G), !.}
\end{verbatim}
Note that cuts inside \texttt{once/1} can only cut the other goals inside \texttt{once/1}.

\texttt{forall(:Cond,:Action)}
For all alternative bindings of \texttt{Cond Action} can be proven. The example verifies that all arithmetic statements in the list \texttt{L} are correct. It does not say which is wrong if one proves wrong.
\begin{verbatim}
?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 * 2]),
       Result =:= Formula).
\end{verbatim}

\texttt{ignore(:Goal)}
Calls \texttt{Goal} as \texttt{once/1}, but succeeds, regardless of whether \texttt{Goal} succeeded or not. Defined as:
\begin{verbatim}
\texttt{ignore(Goal) :-
    Goal, !.}
\texttt{ignore(_).}
\end{verbatim}

\texttt{abort}
Abandons the execution of the current goal and returns to top level. All break levels (see \texttt{break/0} below) are terminated. It is mainly used during debugging or after a serious execution error, to return to the top-level.

\texttt{break}
Suspends the execution of the current goal and creates a new execution level similar to the top level, displaying the following message:
\begin{verbatim}
[ Break \texttt{(level <number>)} ]
\end{verbatim}
telling the depth of the break level just entered. To return to the previous level just type the end-of-file character or call the end\_of\_file predicate. This predicate is especially useful during debugging.

\texttt{halt [ISO]}
Halts Prolog, and exits to the calling application. In YAP, \texttt{halt/0} returns the exit code 0.
halton(+I) [ISO]
    Halts Prolog, and exits to the calling application returning the code given by the integer I.

catch(+Goal,+Exception,+Action) [ISO]
    The goal catch(Goal,Exception,Action) tries to execute goal Goal. If during its execution, Goal throws an exception E’ and this exception unifies with Exception, the exception is considered to be caught and Action is executed. If the exception E’ does not unify with Exception, control again throws the exception.
    The top-level of YAP maintains a default exception handler that is responsible to capture uncaught exceptions.

throw(+Ball) [ISO]
    The goal throw(Ball) throws an exception. Execution is stopped, and the exception is sent to the ancestor goals until reaching a matching catch/3, or until reaching top-level.

garbage_collect
    The goal garbage_collect forces a garbage collection.

garbage_collect_atoms
    The goal garbage_collect forces a garbage collection of the atoms in the database. Currently, only atoms are recovered.

gc
    The goal gc enables garbage collection. The same as yap_flag(gc,on).

nogc
    The goal nogc disables garbage collection. The same as yap_flag(gc,off).

grow_heap(+Size)
    Increase heap size Size kilobytes.

grow_stack(+Size)
    Increase stack size Size kilobytes.

6.2 Handling Undefined Procedures

A predicate in a module is said to be undefined if there are no clauses defining the predicate, and if the predicate has not been declared to be dynamic. What YAP does when trying to execute undefined predicates can be specified in three different ways:

- By setting an YAP flag, through the yap_flag/2 or set_prolog_flag/2 built-ins. This solution generalizes the ISO standard.
- By using the unknown/2 built-in (this solution is compatible with previous releases of YAP).
- By defining clauses for the hook predicate user:unknown_predicate_handler/3. This solution is compatible with SICStus Prolog.

In more detail:

unknown(-O,+N)
    Specifies an handler to be called is a program tries to call an undefined static procedure P.
The arity of $N$ may be zero or one. If the arity is 0, the new action must be one of fail, warning, or error. If the arity is 1, $P$ is an user-defined handler and at run-time, the argument to the handler $P$ will be unified with the undefined goal. Note that $N$ must be defined prior to calling unknown/2, and that the single argument to $N$ must be unbound.

In YAP, the default action is to fail (note that in the ISO Prolog standard the default action is error).

After defining undefined/1 by:

```prolog
undefined(A) :- format('Undefined predicate: ~w~n', [A]), fail.
```

and executing the goal:

```prolog
unknown(U,undefined(X)).
```
a call to a predicate for which no clauses were defined will result in the output of a message of the form:

```
Undefined predicate: user:xyz(A1,A2)
```

followed by the failure of that call.

```prolog
yap_flag(unknown,+SPEC)
```

Alternatively, one can use yap_flag/2, current_prolog_flag/2, or set_prolog_flag/2, to set this functionality. In this case, the first argument for the built-ins should be unknown, and the second argument should be either error, warning, fail, or a goal.

```prolog
user:unknown_predicate_handler(+G,+M,?NG)
```

The user may also define clauses for user:unknown_predicate_handler/3 hook predicate. This user-defined procedure is called before any system processing for the undefined procedure, with the first argument $G$ set to the current goal, and the second $M$ set to the current module. The predicate $G$ will be called from within the user module.

If user:unknown_predicate_handler/3 succeeds, the system will execute $NG$. If user:unknown_predicate_handler/3 fails, the system will execute default action as specified by unknown/2.

```prolog
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.

```prolog
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 SWI built-in autoloader, such as autoloading
code from a database. Do not use this hook to suppress existence errors on predicates. See also **unknown**.

**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.

### 6.3 Message Handling

The interaction between YAP and the user relies on YAP’s ability to portray messages. These messages range from prompts to error information. All message processing is performed through the builtin `print_message/2`, in two steps:

- The message is processed into a list of commands
- The commands in the list are sent to the `format/3` builtin in sequence.

The first argument to `print_message/2` specifies the importance of the message. The options are:

- **error** error handling
- **warning** compilation and run-time warnings,
- **informational**
  - generic informational messages
- **help**
  - help messages (not currently implemented in YAP)
- **query**
  - query used in query processing (not currently implemented in YAP)
- **silent**
  - messages that do not produce output but that can be intercepted by hooks.

The next table shows the main predicates and hooks associated to message handling in YAP:

<table>
<thead>
<tr>
<th><code>print_message(+Kind, Term)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>The predicate <code>print_message/2</code> is used to print messages, notably from exceptions in a human-readable format. <em>Kind</em> is one of informational, banner, warning, error, help or silent. A human-readable message is printed to the stream <code>user_error</code>.</td>
</tr>
</tbody>
</table>

If the Prolog flag `verbose` is `silent`, messages with *Kind* informational, or banner are treated as silent.

This predicate first translates the *Term* into a list of ‘message lines’ (see `print_message_lines/3` for details). Next it will call the hook `message_hook/3` to allow the user intercepting the message. If `message_hook/3` fails it will print the message unless *Kind* is silent.

If you need to report errors from your own predicates, we advise you to stick to the existing error terms if you can; but should you need to invent new ones, you can define corresponding error messages by asserting clauses for `prolog:message/2`. You will need to declare the predicate as multifile.
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.

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’).

<Format> Handed to format/3 as format(Stream, Format, []).
nl A new line is started and if the message is not complete the Prefix
is printed too.

user:message_hook(+Term, +Kind, +Lines)
Hook predicate that may be define 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.
This predicate should be defined dynamic and multifile to allow other modules
defining clauses for it too.

message_to_string(+Term, -String)
Translates a message-term into a string object. Primarily intended for SWI-Prolog emulation.

6.4 Predicates on terms

var(T) [ISO]
Succeeds if T is currently a free variable, otherwise fails.

atom(T) [ISO]
Succeeds if and only if T is currently instantiated to an atom.

atomic(T) [ISO]
Checks whether T is an atomic symbol (atom or number).

compound(T) [ISO]
Checks whether T is a compound term.

db_reference(T)
Checks whether T is a database reference.

float(T) [ISO]
Checks whether T is a floating point number.

rational(T) [ISO]
Checks whether T is a rational number.
integer(T) [ISO]
    Succeeds if and only if T is currently instantiated to an integer.

nonvar(T) [ISO]
    The opposite of var(T).

number(T) [ISO]
    Checks whether T is an integer, rational or a float.

primitive(T)
    Checks whether T is an atomic term or a database reference.

simple(T)
    Checks whether T is unbound, an atom, or a number.

callable(T)
    Checks whether T is a callable term, that is, an atom or a compound term.

numbervars(T, +N1, -Nn)
    Instantiates each variable in term T to a term of the form: '\$VAR'(I), with I increasing from N1 to Nn.

ground(T)
    Succeeds if there are no free variables in the term T.

arg(+N, +T, A) [ISO]
    Succeeds if the argument N of the term T unifies with A. The arguments are numbered from 1 to the arity of the term.
    The current version will generate an error if T or N are unbound, if T is not a compound term, or if N is not a positive integer. Note that previous versions of YAP would fail silently under these errors.

functor(T, F, N) [ISO]
    The top functor of term T is named F and has arity N.
    When T is not instantiated, F and N must be. If N is 0, F must be an atomic symbol, which will be unified with T. If N is not 0, then F must be an atom and T becomes instantiated to the most general term having functor F and arity N. If T is instantiated to a term then F and N are respectively unified with its top functor name and arity.
    In the current version of YAP the arity N must be an integer. Previous versions allowed evaluable expressions, as long as the expression would evaluate to an integer. This feature is not available in the ISO Prolog standard.

T =.. L [ISO]
    The list L is built with the functor and arguments of the term T. If T is instantiated to a variable, then L must be instantiated either to a list whose head is an atom, or to a list consisting of just a number.

X = Y [ISO]
    Tries to unify terms X and Y.

X \= Y [ISO]
    Succeeds if terms X and Y are not unifiable.
unify_with_occurs_check(?T1,?T2) [ISO]
Obtain the most general unifier of terms \( T1 \) and \( T2 \), if there is one.
This predicate implements the full unification algorithm. An example:

\[
\text{unify_with_occurs_check}(a(X,b,Z),a(X,A,f(B))).
\]
will succeed with the bindings \( A = b \) and \( Z = f(B) \). On the other hand:

\[
\text{unify_with_occurs_check}(a(X,b,Z),a(X,A,f(Z))).
\]
would fail, because \( Z \) is not unifiable with \( f(Z) \). Note that \((=)/2\) would succeed
for the previous examples, giving the following bindings \( A = b \) and \( Z = f(Z) \).

\[
\text{copy_term}(?TI,-TF) \ [\text{ISO}]
\]
Term \( TF \) is a variant of the original term \( TI \), such that for each variable \( V \) in
the term \( TI \) there is a new variable \( V' \) in term \( TF \). Notice that:
- suspended goals and attributes for attributed variables in \( TI \) are also duplicated;
- ground terms are shared between the new and the old term.
If you do not want any sharing to occur please use \text{duplicate_term}/2.

duplicate_term(?TI,-TF)
Term \( TF \) is a variant of the original term \( TI \), such that for each variable \( V \) in
the term \( TI \) there is a new variable \( V' \) in term \( TF \), and the two terms do not
share any structure. All suspended goals and attributes for attributed variables
in \( TI \) are also duplicated.
Also refer to \text{copy_term}/2.

\[
is_list(+List)
\]
True when \( List \) is a proper list. That is, \( List \) is bound to the empty list (nil)
or a term with functor '.\' and arity 2.

\[
?\text{Term1} = ?\text{Term2}
\]
Same as \text{variant}/2, succeeds if \( \text{Term1} \) and \( \text{Term2} \) are variant terms.

6.5 Predicates on Atoms
The following predicates are used to manipulate atoms:

\[
\text{name}(A,L)
\]
The predicate holds when at least one of the arguments is ground (otherwise,
an error message will be displayed). The argument \( A \) will be unified with an
atomic symbol and \( L \) with the list of the ASCII codes for the characters of the
external representation of \( A \).

\[
\text{name}(yap,L).
\]
will return:

\[
L = [121,97,112].
\]
and

\[
\text{name}(3,L).
\]
will return:
L = [51].

atom_chars(?A, ?L) [ISO]
The predicate holds when at least one of the arguments is ground (otherwise, an error message will be displayed). The argument A must be unifiable with an atom, and the argument L with the list of the ASCII codes for the characters of the external representation of A.

The ISO-Prolog standard dictates that atom_chars/2 should unify the second argument with a list of one-char atoms, and not the character codes. For compatibility with previous versions of YAP, and with other Prolog implementations, YAP unifies the second argument with the character codes, as in atom_codes/2. Use the set_prolog_flag(to_chars_mode, iso) to obtain ISO standard compatibility.

atom_codes(?A, ?L) [ISO]
The predicate holds when at least one of the arguments is ground (otherwise, an error message will be displayed). The argument A will be unified with an atom and L with the list of the ASCII codes for the characters of the external representation of A.

atom_concat(+As, ?A)
The predicate holds when the first argument is a list of atoms, and the second unifies with the atom obtained by concatenating all the atoms in the first list.

atomic_concat(+As, ?A)
The predicate holds when the first argument is a list of atomic terms, and the second unifies with the atom obtained by concatenating all the atomic terms in the first list. The first argument thus may contain atoms or numbers.

atomic_list_concat(+As, ?A)
The predicate holds when the first argument is a list of atomic terms, and the second unifies with the atom obtained by concatenating all the atomic terms in the first list. The first argument thus may contain atoms or numbers.

atomic_list_concat(?As, +Separator, ?A)
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'

YAP emulates the SWI-Prolog version of this predicate that can also be used to split atoms by instantiating Separator and Atom as shown below.

?- atomic_list_concat(L, '-', 'gnu-gnat').

L = [gnu, gnat]

atom_length(+A, ?I) [ISO]
The predicate holds when the first argument is an atom, and the second unifies with the number of characters forming that atom.
atom_concat(?A1,?A2,?A12) [ISO]
The predicate holds when the third argument unifies with an atom, and the first and second unify with atoms such that their representations concatenated are the representation for A12.
If A1 and A2 are unbound, the built-in will find all the atoms that concatenated give A12.

number_chars(?I,?L)
The predicate holds when at least one of the arguments is ground (otherwise, an error message will be displayed). The argument I must be unifiable with a number, and the argument L with the list of the ASCII codes for the characters of the external representation of I.
The ISO-Prolog standard dictates that number_chars/2 should unify the second argument with a list of one-char atoms, and not the character codes. For compatibility with previous versions of YAP, and with other Prolog implementations, YAP unifies the second argument with the character codes, as in number_codes/2. Use the set_prolog_flag(to_chars_mode,iso) to obtain ISO standard compatibility.

number_codes(?A,?L) [ISO]
The predicate holds when at least one of the arguments is ground (otherwise, an error message will be displayed). The argument A will be unified with a number and L with the list of the ASCII codes for the characters of the external representation of A.

atom_number(?Atom,?Number)
The predicate holds when at least one of the arguments is ground (otherwise, an error message will be displayed). If the argument Atom is an atom, Number must be the number corresponding to the characters in Atom, otherwise the characters in Atom must encode a number Number.

number_atom(?I,?L)
The predicate holds when at least one of the arguments is ground (otherwise, an error message will be displayed). The argument I must be unifiable with a number, and the argument L must be unifiable with an atom representing the number.

sub_atom(+A,?Bef, ?Size, ?After, ?At_out) [ISO]
True when A and At_out are atoms such that the name of At_out has size Size and is a sub-string of the name of A, such that Bef is the number of characters before and After the number of characters afterwards.
Note that A must always be known, but At_out can be unbound when calling this built-in. If all the arguments for sub_atom/5 but A are unbound, the built-in will backtrack through all possible sub-strings of A.

6.6 Predicates on Characters
The following predicates are used to manipulate characters:
**char_code(?A, ?I)** [ISO]

The built-in succeeds with A bound to character represented as an atom, and I bound to the character code represented as an integer. At least, one of either A or I must be bound before the call.

**char_type(?Char, ?Type)**

Tests or generates alternative Types or Chars. 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.

double_quotes

end_of_line  Char ends a line (ASCII: 10..13).

newline  Char is a 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 and the built-in predicates atom_chars/2 and atom_codes/2.

6.7 Comparing Terms

The following predicates are used to compare and order terms, using the standard ordering:

- variables come before numbers, numbers come before atoms which in turn come before compound terms, i.e.: variables @< numbers @< atoms @< compound terms.
- Variables are roughly ordered by "age" (the "oldest" variable is put first);
- Floating point numbers are sorted in increasing order;
- Rational numbers are sorted in increasing order;
- Integers are sorted in increasing order;
- Atoms are sorted in lexicographic order;
- Compound terms are ordered first by arity of the main functor, then by the name of the main functor, and finally by their arguments in left-to-right order.

cancel(C,X,Y)

As a result of comparing X and Y, C may take one of the following values:

- = if X and Y are identical;
- < if X precedes Y in the defined order;
- > if Y precedes X in the defined order;
Chapter 6: Built-In Predicates

\[ X == Y \ [\text{ISO}] \]
Succeeds if terms \( X \) and \( Y \) are strictly identical. The difference between this predicate and \( \neq/2 \) is that, if one of the arguments is a free variable, it only succeeds when they have already been unified.

\[- X == Y.\]
fails, but,
\{- X = Y, X == Y.\}
succeeds.
\{- X == 2.\}
fails, but,
\{- X = 2, X == 2.\}
succeeds.

\[ X \ \backslash== \ Y \ [\text{ISO}] \]
Terms \( X \) and \( Y \) are not strictly identical.

\[ X @< Y \ [\text{ISO}] \]
Term \( X \) precedes term \( Y \) in the standard order.

\[ X @<= Y \ [\text{ISO}] \]
Term \( X \) does not follow term \( Y \) in the standard order.

\[ X @> Y \ [\text{ISO}] \]
Term \( X \) follows term \( Y \) in the standard order.

\[ X @>> Y \ [\text{ISO}] \]
Term \( X \) does not precede term \( Y \) in the standard order.

\[ \text{sort}(+L,-S) \]
Unifies \( S \) with the list obtained by sorting \( L \) and merging identical (in the sense of \( == \)) elements.

\[ \text{keysort}(+L,S) \]
Assuming \( L \) is a list of the form \( \text{Key}-\text{Value} \), \( \text{keysort}(+L,S) \) unifies \( S \) with the list obtained from \( L \), by sorting its elements according to the value of \( \text{Key} \).

\[- \text{keysort}([3-a,1-b,2-c,1-a,1-b],S).\]
would return:
\[ S = [1-b,1-a,1-b,2-c,3-a] \]

\[ \text{predsort}(+Pred, +List, -Sorted) \]
Sorts similar to \( \text{sort}/2 \), but determines the order of two terms by calling \( \text{Pred}(-\Delta, +E1, +E2) \). This call must unify \( \Delta \) with one of \( <, > \) or \( = \). If built-in predicate \( \text{compare}/3 \) is used, the result is the same as \( \text{sort}/2 \).

\[ \text{length}(?L,?S) \]
Unify the well-defined list \( L \) with its length. The procedure can be used to find the length of a pre-defined list, or to build a list of length \( S \).
6.8 Arithmetic

YAP now supposes several different numeric types:

**integers**  When YAP 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 have the same size as an address. 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 cell values on the global stack. Integers that do not fit in 64-bit are represented as serialised GNU MPZ structures on the global stack.

**number**  Rational numbers (Q) are quotients of two integers. Rational arithmetic is only provided if GMP is used (see above). 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.

**float**  Floating point numbers are represented using the C-type double. On most today platforms these are 64-bit IEEE floating point numbers.

Arithmetic functions that require integer arguments accept, in addition to integers, rational numbers with denominator ‘1’ and floating point numbers that can be accurately converted to integers. If the required argument is a float the argument is converted to float. Note that conversion of integers to floating point numbers may raise an overflow exception. In all other cases, arguments are converted to the same type using the order integer to rational number to floating point number.

Arithmetic expressions in YAP may use the following operators or *evaluable predicates*:

\[ +X \]

The value of \( X \) itself.

\[ -X \]  [ISO]

Symmetric value.

\[ X+Y \]  [ISO]

Sum.

\[ X-Y \]  [ISO]

Difference.

\[ X*Y \]  [ISO]

Product.

\[ X/Y \]  [ISO]

Quotient.

\[ X//Y \]  [ISO]

Integer quotient.

\[ X \mod Y \]  [ISO]

Integer remainder.
$X \text{ rem } Y \text{ [ISO]}$
Integer remainder, the same as $\text{mod}$.

$\text{exp}(X) \text{ [ISO]}$
Natural exponential.

$\text{log}(X) \text{ [ISO]}$
Natural logarithm.

$\text{log10}(X)$  Decimal logarithm.

$\text{sqrt}(X) \text{ [ISO]}$
Square root.

$\text{sin}(X) \text{ [ISO]}$
Sine.

$\text{cos}(X) \text{ [ISO]}$
Cosine.

$\text{tan}(X)$  Tangent.

$\text{asin}(X)$  Arc sine.

$\text{acos}(X)$  Arc cosine.

$\text{atan}(X) \text{ [ISO]}$
Arc tangent.

$\text{atan}(X,Y)$  
Four-quadrant arc tangent.

$\text{atan2}(X,Y)$  
Four-quadrant arc tangent, same as $\text{atan}/2$.

$\text{sinh}(X)$  Hyperbolic sine.

$\text{cosh}(X)$  Hyperbolic cosine.

$\text{tanh}(X)$  Hyperbolic tangent.

$\text{asinh}(X)$  Hyperbolic arc sine.

$\text{acosh}(X)$  Hyperbolic arc cosine.

$\text{atanh}(X)$  Hyperbolic arc tangent.

$\text{lgamma}(X)$  
Logarithm of gamma function.

$\text{erf}(X)$  Gaussian error function.

$\text{erfc}(X)$  Complementary gaussian error function.

$\text{random}(X) \text{ [ISO]}$
An integer random number between 0 and $X$.

In \text{iso} language mode the argument must be a floating point-number, the result is an integer and if the float is equidistant it is rounded up, that is, to the least integer greater than $X$. 

integer(X)
    If X evaluates to a float, the integer between the value of X and 0 closest to the value of X, else if X evaluates to an integer, the value of X.

float(X) [ISO]
    If X evaluates to an integer, the corresponding float, else the float itself.

float_fractional_part(X) [ISO]
    The fractional part of the floating point number X, or 0.0 if X is an integer.
    In the iso language mode, X must be an integer.

float_integer_part(X) [ISO]
    The float giving the integer part of the floating point number X, or X if X is an integer. In the iso language mode, X must be an integer.

abs(X) [ISO]
    The absolute value of X.

ceiling(X) [ISO]
    The integer that is the smallest integral value not smaller than X.
    In iso language mode the argument must be a floating point-number and the result is an integer.

floor(X) [ISO]
    The integer that is the greatest integral value not greater than X.
    In iso language mode the argument must be a floating point-number and the result is an integer.

round(X) [ISO]
    The nearest integral value to X. If X is equidistant to two integers, it will be rounded to the closest even integral value.
    In iso language mode the argument must be a floating point-number, the result is an integer and it the float is equidistant it is rounded up, that is, to the least integer greater than X.

sign(X) [ISO]
    Return 1 if the X evaluates to a positive integer, 0 it if evaluates to 0, and -1 if it evaluates to a negative integer. If X evaluates to a floating-point number return 1.0 for a positive X, 0.0 for 0.0, and -1.0 otherwise.

truncate(X) [ISO]
    The integral value between X and 0 closest to X.

rational(X)
    Convert the expression X 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 gives a more intuitive result.
Chapter 6: Built-In Predicates

?- A is rational(0.25).
A is 1 rdiv 4
?- A is rational(0.1).
A = 3602879701896397 rdiv 36028797018963968

rationalize(X)
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.

?- A is rationalize(0.25).
A = 1 rdiv 4
?- A is rationalize(0.1).
A = 1 rdiv 10

max(X,Y) The greater value of X and Y.
min(X,Y) The lesser value of X and Y.
X ^ Y X raised to the power of Y, (from the C-Prolog syntax).
exp(X,Y) X raised to the power of Y, (from the Quintus Prolog syntax).
X ** Y [ISO] X raised to the power of Y (from ISO).
X /\ Y [ISO] Integer bitwise conjunction.
X \ Y [ISO] Integer bitwise disjunction.
X # Y Integer bitwise exclusive disjunction.
X << Y Integer bitwise left logical shift of X by Y places.
X >> Y [ISO] Integer bitwise right logical shift of X by Y places.
\ X [ISO] Integer bitwise negation.
gcd(X,Y) The greatest common divisor of the two integers X and Y.
msb(X) The most significant bit of the non-negative integer X.
lsb(X) The least significant bit of the non-negative integer X.
popcount(X) The number of bits set to 1 in the binary representation of the non-negative integer X.

[X] Evaluates to X for expression X. Useful because character strings in Prolog are lists of character codes.
\[ X \text{ is } Y\times10+C-"0" \]
is the same as
\[ X \text{ is } Y\times10+C-[48]. \]
which would be evaluated as:
\[ X \text{ is } Y\times10+C-48. \]

Besides numbers and the arithmetic operators described above, certain atoms have a special meaning when present in arithmetic expressions:

- **pi** The value of *\pi*, the ratio of a circle's circumference to its diameter.
- **e** The base of the natural logarithms.
- **epsilon** The difference between the float 1.0 and the first larger floating point number.
- **inf** Infinity according to the IEEE Floating-Point standard. Note that evaluating this term will generate a domain error in the iso language mode.
- **nan** Not-a-number according to the IEEE Floating-Point standard. Note that evaluating this term will generate a domain error in the iso language mode.
- **cputime** CPU time in seconds, since YAP was invoked.
- **heapused** Heap space used, in bytes.
- **local** Local stack in use, in bytes.
- **global** Global stack in use, in bytes.
- **random** A "random" floating point number between 0 and 1.

The primitive YAP predicates involving arithmetic expressions are:

- **X is +Y** [2]
  This predicate succeeds if the result of evaluating the expression *Y* unifies with *X*. This is the predicate normally used to perform evaluation of arithmetic expressions:
  \[ X \text{ is } 2+3\times4 \]
  succeeds with *X* = 14.
- **+X < +Y** [ISO]
  The value of the expression *X* is less than the value of expression *Y*.
- **+X <= +Y** [ISO]
  The value of the expression *X* is less than or equal to the value of expression *Y*.
- **+X > +Y** [ISO]
  The value of the expression *X* is greater than the value of expression *Y*.
- **+X >= +Y** [ISO]
  The value of the expression *X* is greater than or equal to the value of expression *Y*.
- **+X =:= +Y** [ISO]
  The value of the expression *X* is equal to the value of expression *Y*. 
$X \neq Y$ [ISO]

The value of the expression $X$ is different from the value of expression $Y$.

srandom(+X)

Use the argument $X$ as a new seed for YAP’s random number generator. The argument should be an integer, but floats are acceptable.

Notes:
- Since YAP4, YAP does not convert automatically between integers and floats.
- arguments to trigonometric functions are expressed in radians.
- if a (non-instantiated) variable occurs in an arithmetic expression YAP will generate an exception. If no error handler is available, execution will be thrown back to the top-level.

The following predicates provide counting:

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 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. 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.

plus(?Int1, ?Int2, ?Int3)

True if $Int3 = Int1 + Int2$. At least two of the three arguments must be instantiated to integers.

6.9 I/O Predicates

Some of the I/O predicates described below will in certain conditions provide error messages and abort only if the file_errors flag is set. If this flag is cleared the same predicates will just fail. Details on setting and clearing this flag are given under 7.7.

6.9.1 Handling Streams and Files

open(+F, +M, -S) [ISO]

Opens the file with name $F$ in mode $M$ (‘read’, ‘write’ or ‘append’), returning $S$ unified with the stream name.

At most, there are 17 streams opened at the same time. Each stream is either an input or an output stream but not both. There are always 3 open streams: user_input for reading, user_output for writing and user_error for writing. If there is no ambiguity, the atoms user_input and user_output may be referred to as user.
The file_errors flag controls whether errors are reported when in mode 'read' or 'append' the file \( F \) does not exist or is not readable, and whether in mode 'write' or 'append' the file is not writable.

\[
\text{open}(+F,+M,-S,+\text{Opts}) \quad \text{[ISO]}
\]

Opens the file with name \( F \) in mode \( M \) ('read', 'write' or 'append'), returning \( S \) unified with the stream name, and following these options:

- **type(+\( T \))** Specify whether the stream is a text stream (default), or a binary stream.
- **reposition(+\( \text{Bool} \))** Specify whether it is possible to reposition the stream (true), or not (false). By default, YAP enables repositioning for all files, except terminal files and sockets.
- **eof_action(+\( \text{Action} \))** Specify the action to take if attempting to input characters from a stream where we have previously found an end_of_file. The possible actions are error, that raises an error, reset, that tries to reset the stream and is used for tty type files, and eof_code, which generates a new end_of_file (default for non-tty files).
- **alias(+\( \text{Name} \))** Specify an alias to the stream. The alias \( \text{Name} \) must be an atom. The alias can be used instead of the stream descriptor for every operation concerning the stream. The operation will fail and give an error if the alias name is already in use. YAP allows several aliases for the same file, but only one is returned by stream_property/2.
- **bom(+\( \text{Bool} \))** 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 3.3.2 [BOM], page 19 for details.
- **encoding(+\( \text{Encoding} \))** Set the encoding used for text. See Section 3.3 [Encoding], page 17 for an overview of wide character and encoding issues.
- **representation_errors(+\( \text{Mode} \))** Change the behaviour when writing characters to the stream that cannot be represented by the encoding. The behaviour is one of error (throw and I/O error exception), prolog (write \u...\ 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 3.3 [Encoding], page 17.
- **expand_filename(+\( \text{Mode} \))** If \( \text{Mode} \) is true then do filename expansion, then ask Prolog to do file name expansion before actually trying to opening the file: this includes processing ~ characters and processing $ environment
variables at the beginning of the file. Otherwise, just try to open the file using the given name.
The default behavior is given by the Prolog flag \texttt{open_expands_filename}.

\begin{verbatim}
\textbf{close(+S) [ISO]}
\end{verbatim}
Closes the stream \texttt{S}. If \texttt{S} does not stand for a stream currently opened an error is reported. The streams \texttt{user_input}, \texttt{user_output}, and \texttt{user_error} can never be closed.

\begin{verbatim}
\textbf{close(+S,+O) [ISO]}
\end{verbatim}
Closes the stream \texttt{S}, following options \texttt{O}.
The only valid options are \texttt{force(true)} and \texttt{force(false)}. YAP currently ignores these options.

\begin{verbatim}
\textbf{time_file(+File,-Time)}
\end{verbatim}
Unify the last modification time of \texttt{File} with \texttt{Time}. \texttt{Time} is a floating point number expressing the seconds elapsed since Jan 1, 1970.

\begin{verbatim}
\textbf{absolute_file_name(+Name,+Options, -FullPath)}
\textbf{absolute_file_name(+Name, -FullPath,+Options)}
\end{verbatim}
Converts the given file specification into an absolute path. \texttt{Option} is a list of options to guide the conversion:

\begin{verbatim}
\textbf{extensions(+ListOfExtensions)}
\end{verbatim}
List of file-extensions to try. Default is ‘’’. For each extension, \texttt{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 \texttt{.ext} or plain \texttt{ext}.

\begin{verbatim}
\textbf{relative_to(+FileOrDir)}
\end{verbatim}
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 \texttt{working_directory/2}) or, if \texttt{Spec} is atomic and \texttt{absolute_file_name/[2,3]} is executed in a directive, it uses the current source-file as reference.

\begin{verbatim}
\textbf{access(+Mode)}
\end{verbatim}
Imposes the condition \texttt{access_file(File, Mode)}. \texttt{Mode} is on of \texttt{read}, \texttt{write}, \texttt{append}, \texttt{exist} or \texttt{none} (default). See also \texttt{access_file/2}.

\begin{verbatim}
\textbf{file_type(+Type)}
\end{verbatim}
Defines extensions. Current mapping: \texttt{txt} implies \texttt{[’’]}, \texttt{prolog} implies \texttt{[’’.pl’, ’’]}, \texttt{executable} implies \texttt{[’’.so’, ’’]}, \texttt{qlf} implies \texttt{[’’.qlf’, ’’]} and \texttt{directory} implies \texttt{[’’]}. The file-type \texttt{source} is an alias for \texttt{prolog} for compatibility to SICStus Prolog. See also \texttt{prolog_file_type/2}.

\begin{verbatim}
\textbf{file_errors(fail/error)}
\end{verbatim}
If \texttt{error} (default), throw and \texttt{existence_error} exception if the file cannot be found. If \texttt{fail}, stay silent.
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.

Compatibility considerations to common argument-order in ISO as well as SIC-Stus absolute_file_name/3 forced us to be flexible here. If the last argument is a list and the 2nd not, the arguments are swapped, making the call absolute_file_name(+Spec, -Path, +Options) valid as well.

absolute_file_name(+Name,-FullPath)
Give the path a full path FullPath YAP would use to consult a file named Name. Unify FullPath with user if the file name is user.

file_base_name(+Name,-FileName)
Give the path a full path FullPath extract the FileName.

file_name_extension(?Base,?Extension, ?Name)
This predicate is used to add, remove or test filename extensions. The main reason for its introduction is to deal with different filename properties in a portable manner. If the file system is case-insensitive, testing for an extension will be done case-insensitive too. Extension may be specified with or without a leading dot (.). If an Extension is generated, it will not have a leading dot.

current_stream(F,M,S)
Defines the relation: The stream S is opened on the file F in mode M. It might be used to obtain all open streams (by backtracking) or to access the stream for a file F in mode M, or to find properties for a stream S.

is_stream(S)
Succeeds if S is a currently open stream.

flush_output [ISO]
Send out all data in the output buffer of the current output stream.

flush_output(+S) [ISO]
Send all data in the output buffer for stream S.

set_input(+S)
Set stream S as the current input stream. Predicates like read/1 and get/1 will start using stream S.

set_output(+S)
Set stream S as the current output stream. Predicates like write/1 and put/1 will start using stream S.

stream_select(+STREAMS,+TIMEOUT,-READSTREAMS)
Given a list of open STREAMS opened in read mode and a TIMEOUT return a list of streams who are now available for reading.
If the TIMEOUT is instantiated to off, stream_select/3 will wait indefinitely for a stream to become open. Otherwise the timeout must be of the form SECS:USECS where SECS is an integer gives the number of seconds to wait for a timeout and USECS adds the number of micro-seconds.
This built-in is only defined if the system call `select` is available in the system.

`current_input(-S) [ISO]`
Unify S with the current input stream.

`current_output(-S) [ISO]`
Unify S with the current output stream.

`at_end_of_stream [ISO]`
Succeed if the current stream has stream position end-of-stream or past-end-of-stream.

`at_end_of_stream(+S) [ISO]`
Succeed if the stream S has stream position end-of-stream or past-end-of-stream. Note that S must be a readable stream.

`set_stream_position(+S,+POS) [ISO]`
Given a stream position POS for a stream S, set the current stream position for S to be POS.

`stream_property(?Stream,?Prop) [ISO]`
Obtain the properties for the open streams. If the first argument is unbound, the procedure will backtrack through all open streams. Otherwise, the first argument must be a stream term (you may use `current_stream` to obtain a current stream given a file name).

The following properties are recognized:

`file_name(P)`
An atom giving the file name for the current stream. The file names are `user_input`, `user_output`, and `user_error` for the standard streams.

`mode(P)`
The mode used to open the file. It may be one of `append`, `read`, or `write`.

`input`  
The stream is readable.

`output`  
The stream is writable.

`alias(A)`  
ISO-Prolog primitive for stream aliases. YAP returns one of the existing aliases for the stream.

`position(P)`  
A term describing the position in the stream.

`end_of_stream(E)`  
Whether the stream is at the end of stream, or it has found the end of stream and is past, or whether it has not yet reached the end of stream.

`eof_action(A)`  
The action to take when trying to read after reaching the end of stream. The action may be one of `error`, generate an error, `eof_code`, return character code -1, or `reset` the stream.
reposition(B)
Whether the stream can be repositioned or not, that is, whether it is seekable.

type(T)
Whether the stream is a text stream or a binary stream.

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 3.3.2 [BOM], page 19 for details.

encoding(+Encoding)
Query the encoding used for text. See Section 3.3 [Encoding], page 17 for an overview of wide character and encoding issues in YAP.

representation_errors(+Mode)
Behaviour when writing characters to the stream that cannot be represented by the encoding. The behaviour is one of error (throw and I/O error exception), prolog (write \u...\ 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 3.3 [Encoding], page 17 and open/4.

current_line_number(-LineNumber)
Unify LineNumber with the line number for the current stream.

current_line_number(+Stream,-LineNumber)
Unify LineNumber with the line number for the Stream.

line_count(+Stream,-LineNumber)
Unify LineNumber with the line number for the Stream.

character_count(+Stream,-CharacterCount)
Unify CharacterCount with the number of characters written to or read from Stream.

line_position(+Stream,-LinePosition)
Unify LinePosition with the position on current text stream Stream.

stream_position(+Stream,-StreamPosition)
Unify StreamPosition with the packaged information of position on current stream Stream. Use stream_position_data/3 to retrieve information on character or line count.

stream_position_data(+Field,+StreamPosition,-Info)
Given the packaged stream position term StreamPosition, unify Info with Field line_count, byte_count, or char_count.

6.9.2 Handling Streams and Files

tell(+S)
If S is a currently opened stream for output, it becomes the current output stream. If S is an atom it is taken to be a filename. If there is no output stream currently associated with it, then it is opened for output, and the new output
stream created becomes the current output stream. If it is not possible to open
the file, an error occurs. If there is a single opened output stream currently
associated with the file, then it becomes the current output stream; if there are
more than one in that condition, one of them is chosen.

Whenever $S$ is a stream not currently opened for output, an error may be
reported, depending on the state of the file_errors flag. The predicate just fails,
if $S$ is neither a stream nor an atom.

telling(-$S$)
The current output stream is unified with $S$.

told Closes the current output stream, and the user’s terminal becomes again the
current output stream. It is important to remember to close streams after
having finished using them, as the maximum number of simultaneously opened
streams is 17.

see(+$S$) If $S$ is a currently opened input stream then it is assumed to be the current input
stream. If $S$ is an atom it is taken as a filename. If there is no input stream
currently associated with it, then it is opened for input, and the new input
stream thus created becomes the current input stream. If it is not possible to
open the file, an error occurs. If there is a single opened input stream currently
associated with the file, it becomes the current input stream; if there are more
than one in that condition, then one of them is chosen.

When $S$ is a stream not currently opened for input, an error may be reported,
depending on the state of the file_errors flag. If $S$ is neither a stream nor
an atom the predicates just fails.

seeing(-$S$)
The current input stream is unified with $S$.

seen Closes the current input stream (see 6.7.).

6.9.3 Handling Input/Output of Terms

read(-$T$) [ISO]
Reads the next term from the current input stream, and unifies it with $T$. The
term must be followed by a dot (‘.’) and any blank-character as previously
defined. The syntax of the term must match the current declarations for op-
erators (see op). If the end-of-stream is reached, $T$ is unified with the atom
end_of_file. Further reads from of the same stream may cause an error failure
(see open/3).

read_term(-$T$,+Options) [ISO]
Reads term $T$ from the current input stream with execution controlled by the
following options:

term_position(-Position)
Unify Position with a term describing the position of the stream at
the start of parse. Use stream_position_data/3 to obtain extra
information.
singletons(-Names)
   Unify Names with a list of the form Name=Var, where Name is the
   name of a non-anonymous singleton variable in the original term,
   and Var is the variable’s representation in YAP.

syntax_errors(+Val)
   Control action to be taken after syntax errors. See yap_flag/2 for
detailed information.

variable_names(-Names)
   Unify Names with a list of the form Name=Var, where Name is the
   name of a non-anonymous variable in the original term, and Var is
   the variable’s representation in YAP.

variables(-Names)
   Unify Names with a list of the variables in term T.

char_conversion(+IN,+OUT) [ISO]
   While reading terms convert unquoted occurrences of the character IN to the
   character OUT. Both IN and OUT must be bound to single characters atoms.
   Character conversion only works if the flag char_conversion is on. This is
   default in the iso and sicstus language modes. As an example, character
   conversion can be used for instance to convert characters from the ISO-LATIN-
   1 character set to ASCII.
   If IN is the same character as OUT, char_conversion/2 will remove this con-
   version from the table.

current_char_conversion(?IN,?OUT) [ISO]
   If IN is unbound give all current character translations. Otherwise, give the
   translation for IN, if one exists.

write(T) [ISO]
   The term T is written to the current output stream according to the operator
   declarations in force.

writeln(T) [ISO]
   Same as write/1 followed by nl/0.

display(+T)
   Displays term T on the current output stream. All Prolog terms are written in
   standard parenthesized prefix notation.

write_canonical(+T) [ISO]
   Displays term T on the current output stream. Atoms are quoted when neces-
   sary, and operators are ignored, that is, the term is written in standard paren-
   thesized prefix notation.

write_term(+T, +Opts) [ISO]
   Displays term T on the current output stream, according to the following op-
   tions:
quoted(+Bool)
   If true, quote atoms if this would be necessary for the atom to
   be recognized as an atom by YAP’s parser. The default value is
   false.

ignore_ops(+Bool)
   If true, ignore operator declarations when writing the term. The
   default value is false.

numbervars(+Bool)
   If true, output terms of the form ’$VAR’(N), where N is an integer,
   as a sequence of capital letters. The default value is false.

portrayed(+Bool)
   If true, use portray/1 to portray bound terms. The default value
   is false.

portray(+Bool)
   If true, use portray/1 to portray bound terms. The default value
   is false.

max_depth(+Depth)
   If Depth is a positive integer, use Depth as the maximum depth to
   portray a term. The default is 0, that is, unlimited depth.

priority(+Priority)
   If Priority is a positive integer smaller than 1200, give the context
   priority. The default is 1200.

cycles(+Bool)
   Do not loop in rational trees (default).

writeq(T) [ISO]
   Writes the term T, quoting names to make the result acceptable to the predicate
   ‘read’ whenever necessary.

print(T)
   Prints the term T to the current output stream using write/1 unless T is
   bound and a call to the user-defined predicate portray/1 succeeds. To do
   pretty printing of terms the user should define suitable clauses for portray/1
   and use print/1.

format(+T,+L)
   Print formatted output to the current output stream. The arguments in list L
   are output according to the string or atom T.

   A control sequence is introduced by a w. The following control sequences are
   available in YAP:
   ’~~’       Print a single tilde.
   ’~a’       The next argument must be an atom, that will be printed as if by
               write.
   ’~Nc’      The next argument must be an integer, that will be printed as a
               character code. The number N is the number of times to print the
               character (default 1).
The next argument must be a floating point number. The float $F$, the number $N$ and the control code $c$ will be passed to `printf` as:

```fortran
printf("%s.Nc", F)
```

As an example:

```prolog
?- format("~8e, ~8E, ~8f, ~8g, ~8G~w", 
3.14,3.14,3.14,3.14,3.14)).
3.140000e+00, 3.140000E+00, 3.140000, 3.14, 3.143.14
```

The next argument must be an integer, and $N$ is the number of digits after the decimal point. If $N$ is 0 no decimal points will be printed. The default is $N = 0$.

```prolog
?- format("~2d, ~d",[15000, 15000]).
150.00, 15000
```

Identical to `~Nd`, except that commas are used to separate groups of three digits.

```prolog
?- format("~2D, ~D",[150000, 150000]).
1,500.00, 150,000
```

Ignore the next argument in the list of arguments:

```prolog
?- format('The ~i met the boregrove',[mimsy]).
The met the boregrove
```

Print the next argument with `write_canonical`:

```prolog
?- format("Good night ~k",a+[1,2]).
Good night +(a,[1,2])
```

Print $N$ newlines (where $N$ defaults to 1).

Print $N$ newlines if at the beginning of the line (where $N$ defaults to 1).

The next argument must be an integer, and $N$ is interpreted as a radix, such that $2 \leq N \leq 36$ (the default is 8).

```prolog
?- format("~2r, 0x~16r, ~r", 
[150000, 150000, 150000]).
10010010011110000, 0x249f0, 444760
```

Note that the letters a-z denote digits larger than 9.

Similar to `~NR`. The next argument must be an integer, and $N$ is interpreted as a radix, such that $2 \leq N \leq 36$ (the default is 8).

```prolog
?- format("~2r, 0x~16r, ~r", 
[150000, 150000, 150000]).
10010010011110000, 0x249F0, 444760
```

The only difference is that letters A-Z denote digits larger than 9.
''p' Print the next argument with print/1:

?- format("Good night "p",a+[1,2]).
   Good night a+[1,2]

''q' Print the next argument with writeq/1:

?- format("Good night "q",'Hello'+[1,2]).
   Good night 'Hello'+[1,2]

''Ns' The next argument must be a list of character codes. The system then outputs their representation as a string, where N is the maximum number of characters for the string (N defaults to the length of the string).

?- format("The ~s are ~4s",["woods","lovely"]).
   The woods are love

''w' Print the next argument with write/1:

?- format("Good night "w",'Hello'+[1,2]).
   Good night Hello+[1,2]

The number of arguments, N, may be given as an integer, or it may be given as an extra argument. The next example shows a small procedure to write a variable number of a characters:

write_many_as(N) :-
   format("~*c",[N,0'a]).

The format/2 built-in also allows for formatted output. One can specify column boundaries and fill the intermediate space by a padding character:

''N|' Set a column boundary at position N, where N defaults to the current position.

''N+ ' Set a column boundary at N characters past the current position, where N defaults to 8.

''Nt' Set padding for a column, where N is the fill code (default is SPC).

The next example shows how to align columns and padding. We first show left-alignment:

?- format("~n*Hello~16++~n",[]).
   *Hello *

Note that we reserve 16 characters for the column.

The following example shows how to do right-alignment:

?- format("*~tHello~16++~n",[]).
   *Hello*
The `~t` escape sequence forces filling before Hello.
We next show how to do centering:

```prolog
?- format("*~tHello~t~16*~n",[]).
   * Hello *
```

The two `~t` escape sequence force filling both before and after Hello. Space is then evenly divided between the right and the left sides.

`format(+T)`
Print formatted output to the current output stream.

`format(+S,+T,+L)`
Print formatted output to stream `S`.

`with_output_to(+Output,:Goal)`
Run `Goal` as `once/1`, while characters written to the current output are sent to `Output`. The predicate is SWI-Prolog specific.

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/3` to insert a term in the output:

```prolog
term(Term, In, Tail) :-
   with_output_to(codes(In, Tail), write(Term)).
```

```prolog
?- phrase(term(hello), X).
```

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 (not supported in YAP).

`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.
Chapter 6: Built-In Predicates

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.

6.9.4 Handling Input/Output of Characters

put(+N) Outputs to the current output stream the character whose ASCII code is N. The character N must be a legal ASCII character code, an expression yielding such a code, or a list in which case only the first element is used.

put_byte(+N) [ISO]
Outputs to the current output stream the character whose code is N. The current output stream must be a binary stream.

put_char(+N) [ISO]
Outputs to the current output stream the character who is used to build the representation of atom A. The current output stream must be a text stream.

put_code(+N) [ISO]
Outputs to the current output stream the character whose ASCII code is N. The current output stream must be a text stream. The character N must be a legal ASCII character code, an expression yielding such a code, or a list in which case only the first element is used.

get(-C)
The next non-blank character from the current input stream is unified with C. Blank characters are the ones whose ASCII codes are not greater than 32. If there are no more non-blank characters in the stream, C is unified with -1. If end_of_stream has already been reached in the previous reading, this call will give an error message.

get0(-C)
The next character from the current input stream is consumed, and then unified with C. There are no restrictions on the possible values of the ASCII code for the character, but the character will be internally converted by YAP.

get_byte(-C) [ISO]
If C is unbound, or is a character code, and the current stream is a binary stream, read the next byte from the current stream and unify its code with C.

get_char(-C) [ISO]
If C is unbound, or is an atom representation of a character, and the current stream is a text stream, read the next character from the current stream and unify its atom representation with C.

get_code(-C) [ISO]
If C is unbound, or is the code for a character, and the current stream is a text stream, read the next character from the current stream and unify its code with C.
peek_byte(-C) [ISO]
If C is unbound, or is a character code, and the current stream is a binary stream, read the next byte from the current stream and unify its code with C, while leaving the current stream position unaltered.

peek_char(-C) [ISO]
If C is unbound, or is an atom representation of a character, and the current stream is a text stream, read the next character from the current stream and unify its atom representation with C, while leaving the current stream position unaltered.

peek_code(-C) [ISO]
If C is unbound, or is the code for a character, and the current stream is a text stream, read the next character from the current stream and unify its code with C, while leaving the current stream position unaltered.

skip(+N) Skips input characters until the next occurrence of the character with ASCII code N. The argument to this predicate can take the same forms as those for put (see 6.11).

tab(+N) Outputs N spaces to the current output stream.

nl [ISO] Outputs a new line to the current output stream.

6.9.5 Input/Output Predicates applied to Streams

read(+S,-T) [ISO]
Reads term T from the stream S instead of from the current input stream.

read_term(+S,-T,+Options) [ISO]
Reads term T from stream S with execution controlled by the same options as read_term/2.

write(+S,T) [ISO]
Writes term T to stream S instead of to the current output stream.

write_canonical(+S,+T) [ISO]
Displays term T on the stream S. Atoms are quoted when necessary, and operators are ignored.

write_canonical(+T) [ISO]
Displays term T. Atoms are quoted when necessary, and operators are ignored.

write_term(+S, +T, +Opts) [ISO]
Displays term T on the current output stream, according to the same options used by write_term/3.

writeq(+S,T) [ISO]
As writeq/1, but the output is sent to the stream S.

display(+S,T)
Like display/1, but using stream S to display the term.

print(+S,T)
Prints term T to the stream S instead of to the current output stream.
put(+S,+N)
   As put(N), but to stream S.

put_byte(+S,+N) [ISO]
   As put_byte(N), but to binary stream S.

put_char(+S,+A) [ISO]
   As put_char(A), but to text stream S.

put_code(+S,+N) [ISO]
   As put_code(N), but to text stream S.

get(+S,-C)
   The same as get(C), but from stream S.

get0(+S,-C)
   The same as get0(C), but from stream S.

get_byte(+S,-C) [ISO]
   If C is unbound, or is a character code, and the stream S is a binary stream,
   read the next byte from that stream and unify its code with C.

get_char(+S,-C) [ISO]
   If C is unbound, or is an atom representation of a character, and the stream
   S is a text stream, read the next character from that stream and unify its
   representation as an atom with C.

get_code(+S,-C) [ISO]
   If C is unbound, or is a character code, and the stream S is a text stream, read
   the next character from that stream and unify its code with C.

peek_byte(+S,-C) [ISO]
   If C is unbound, or is a character code, and S is a binary stream, read the
   next byte from the current stream and unify its code with C, while leaving
   the current stream position unaltered.

peek_char(+S,-C) [ISO]
   If C is unbound, or is an atom representation of a character, and the stream
   S is a text stream, read the next character from that stream and unify its
   representation as an atom with C, while leaving the current stream position
   unaltered.

peek_code(+S,-C) [ISO]
   If C is unbound, or is an atom representation of a character, and the stream
   S is a text stream, read the next character from that stream and unify its
   representation as an atom with C, while leaving the current stream position
   unaltered.

skip(+S,-C)
   Like skip/1, but using stream S instead of the current input stream.

tab(+S,+N)
   The same as tab/1, but using stream S.

nl(+S)  Outputs a new line to stream S.
6.9.6 Compatible C-Prolog predicates for Terminal I/O

ttyput(+N)
As put(N) but always to user_output.

ttyget(-C)
The same as get(C), but from stream user_input.

ttyget0(-C)
The same as get0(C), but from stream user_input.

ttyskip(-C)
Like skip/1, but always using stream user_input. stream.

ttytab(+N)
The same as tab/1, but using stream user_output.

ttynl
Outputs a new line to stream user_output.

6.9.7 Controlling Input/Output

exists(+F)
Checks if file F exists in the current directory.

nofileerrors
Switches off the file_errors flag, so that the predicates see/1, tell/1, open/3
and close/1 just fail, instead of producing an error message and aborting
whenever the specified file cannot be opened or closed.

fileerrors
Switches on the file_errors flag so that in certain error conditions I/O predicates
will produce an appropriated message and abort.

write_depth(T,L,A)
Unifies T with the value of the maximum depth of a term to be written, L with
the maximum length of a list to write, and A with the maximum number of
arguments of a compound term to write. The setting will be used by write/1
or write/2. The default value for all arguments is 0, meaning unlimited depth
and length.

?- write_depth(3,5,5).
yes
?- write(a(b(c(d(e(f(g))))))).
a(b(c(....)))
yes
?- write([1,2,3,4,5,6,7,8]).
[1,2,3,4,5,...]
yes
?- write(a(1,2,3,4,5,6,7,8)).
a(1,2,3,4,5,...)
yes

write_depth(T,L)
Same as write_depth(T,L,_).. Unifies T with the value of the maximum depth
depth of a term to be written, and L with the maximum length of a list to write. The
setting will be used by write/1 or write/2. The default value for all arguments is 0, meaning unlimited depth and length.

?- write_depth(3,5,5).
yes
?- write(a(b(c(d(e(f(g))))))).
a(b(c(...)))
yes
?- write([1,2,3,4,5,6,7,8]).
[1,2,3,4,5,...]
yes

always_prompt_user
Force the system to prompt the user even if the user_input stream is not a terminal. This command is useful if you want to obtain interactive control from a pipe or a socket.

6.9.8 Using Sockets From YAP
YAP includes a SICStus Prolog compatible socket interface. This is a low level interface that provides direct access to the major socket system calls. These calls can be used both to open a new connection in the network or connect to a networked server. Socket connections are described as read/write streams, and standard I/O built-ins can be used to write on or read from sockets. The following calls are available:

socket(+DOMAIN,+TYPE,+PROTOCOL,-SOCKET)
Corresponds to the BSD system call socket. Create a socket for domain DOMAIN of type TYPE and protocol PROTOCOL. Both DOMAIN and TYPE should be atoms, whereas PROTOCOL must be an integer. The new socket object is accessible through a descriptor bound to the variable SOCKET.
The current implementation of YAP only accepts two socket domains: ‘AF_INET’ and ‘AF_UNIX’. Socket types depend on the underlying operating system, but at least the following types are supported: ‘SOCK_STREAM’ and ‘SOCK_DGRAM’.

socket(+DOMAIN,-SOCKET)
Call socket/4 with TYPE bound to ‘SOCK_STREAM’ and PROTOCOL bound to 0.

socket_close(+SOCKET)
Close socket SOCKET. Note that sockets used in socket_connect (that is, client sockets) should not be closed with socket_close, as they will be automatically closed when the corresponding stream is closed with close/1 or close/2.

socket_bind(+SOCKET, ?PORT)
Interface to system call bind, as used for servers: bind socket to a port. Port information depends on the domain:

‘AF_UNIX’(+FILENAME)
‘AF_FILE’(+FILENAME)
use file name FILENAME for UNIX or local sockets.
'AF_INET'(HOST, PORT)
If HOST is bound to an atom, bind to host HOST, otherwise if unbound bind to local host (HOST remains unbound). If port PORT is bound to an integer, try to bind to the corresponding port. If variable PORT is unbound allow operating systems to choose a port number, which is unified with PORT.

socket_connect(+SOCKET, +PORT, -STREAM)
Interface to system call connect, used for clients: connect socket SOCKET to PORT. The connection results in the read/write stream STREAM.
Port information depends on the domain:
'AF_UNIX'(FILENAME)
'AF_FILE'(FILENAME)
connect to socket at file FILENAME.
'AF_INET'(HOST, PORT)
Connect to socket at host HOST and port PORT.

socket_listen(+SOCKET, +LENGTH)
Interface to system call listen, used for servers to indicate willingness to wait for connections at socket SOCKET. The integer LENGTH gives the queue limit for incoming connections, and should be limited to 5 for portable applications. The socket must be of type SOCK_STREAM or SOCK_SEQPACKET.

socket_accept(+SOCKET, -STREAM)
socket_accept(+SOCKET, -CLIENT, -STREAM)
Interface to system call accept, used for servers to wait for connections at socket SOCKET. The stream descriptor STREAM represents the resulting connection. If the socket belongs to the domain 'AF_INET', CLIENT unifies with an atom containing the IP address for the client in numbers and dots notation.

socket_accept(+SOCKET, -STREAM)
Accept a connection but do not return client information.

socket_buffering(+SOCKET, -MODE, -OLD, +NEW)
Set buffering for SOCKET in read or write MODE. OLD is unified with the previous status, and NEW receives the new status which may be one of unbuf or fullbuf.

socket_select(+SOCKETS, +NEWSTREAMS, +TIMEOUT, +STREAMS, -READSTREAMS)
Interface to system call select, used for servers to wait for connection requests or for data at sockets. The variable SOCKETS is a list of form KEY-SOCKET, where KEY is an user-defined identifier and SOCKET is a socket descriptor. The variable TIMEOUT is either off, indicating execution will wait until something is available, or of the form SEC-USEC, where SEC and USEC give the seconds and microseconds before socket_select/5 returns. The variable SOCKETS is a list of form KEY-STREAM, where KEY is an user-defined identifier and STREAM is a stream descriptor.
Execution of socket_select/5 unifies READSTREAMS from STREAMS with readable data, and NEWSTREAMS with a list of the form KEY-STREAM,
where $KEY$ was the key for a socket with pending data, and $STREAM$ the stream descriptor resulting from accepting the connection.

```prolog
current_host(\(?HOSTNAME\))

Unify $HOSTNAME$ with an atom representing the fully qualified hostname for the current host. Also succeeds if $HOSTNAME$ is bound to the unqualified hostname.
```

```prolog
hostname_address(\(?HOSTNAME,\(?IP_ADDRESS\))

$HOSTNAME$ is an host name and $IP_ADDRESS$ its IP address in number and dots notation.
```

## 6.10 Using the Clausal Data Base

Predicates in YAP may be dynamic or static. By default, when consulting or reconsulting, predicates are assumed to be static: execution is faster and the code will probably use less space. Static predicates impose some restrictions: in general there can be no addition or removal of clauses for a procedure if it is being used in the current execution.

Dynamic predicates allow programmers to change the Clausal Data Base with the same flexibility as in C-Prolog. With dynamic predicates it is always possible to add or remove clauses during execution and the semantics will be the same as for C-Prolog. But the programmer should be aware of the fact that asserting or retracting are still expensive operations, and therefore he should try to avoid them whenever possible.

```prolog
dynamic +P

Declares predicate $P$ or list of predicates $[P1,...,Pn]$ as a dynamic predicate. $P$ must be written in form: \textit{name/arity}.

```prolog
:- dynamic god/1.
```

a more convenient form can be used:

```prolog
:- dynamic son/3, father/2, mother/2.
```

or, equivalently,

```prolog
:- dynamic [son/3, father/2, mother/2].
```

Note:

a predicate is assumed to be dynamic when asserted before being defined.

```prolog
dynamic_predicate(+P,+Semantics)

Declares predicate $P$ or list of predicates $[P1,...,Pn]$ as a dynamic predicate following either logical or immediate semantics.
```

## 6.10.1 Modification of the Data Base

These predicates can be used either for static or for dynamic predicates:

```prolog
assert(+C)

Adds clause $C$ to the program. If the predicate is undefined, declare it as dynamic.
```

Most Prolog systems only allow asserting clauses for dynamic predicates. This is also as specified in the ISO standard. YAP allows asserting clauses for static predicates, as long as the predicate is not in use and the language flag is
cprolog. Note that this feature is deprecated, if you want to assert clauses for static procedures you should use **assert_static/1**.

**asserta(+C)** [ISO]

Adds clause \( C \) to the beginning of the program. If the predicate is undefined, declare it as dynamic.

**assertz(+C)** [ISO]

Adds clause \( C \) to the end of the program. If the predicate is undefined, declare it as dynamic.

Most Prolog systems only allow asserting clauses for dynamic predicates. This is also as specified in the ISO standard. YAP allows asserting clauses for static predicates. The current version of YAP supports this feature, but this feature is deprecated and support may go away in future versions.

**abolish(+PredSpec)** [ISO]

Deletes the predicate given by \( PredSpec \) from the database. If \( PredSpec \) is an unbound variable, delete all predicates for the current module. The specification must include the name and arity, and it may include module information. Under iso language mode this built-in will only abolish dynamic procedures. Under other modes it will abolish any procedures.

**abolish(+P,+N)**

Deletes the predicate with name \( P \) and arity \( N \). It will remove both static and dynamic predicates.

**assert_static(:C)**

Adds clause \( C \) to a static procedure. Asserting a static clause for a predicate while choice-points for the predicate are available has undefined results.

**asserta_static(:C)**

Adds clause \( C \) to the beginning of a static procedure.

**assertz_static(:C)**

Adds clause \( C \) to the end of a static procedure. Asserting a static clause for a predicate while choice-points for the predicate are available has undefined results.

The following predicates can be used for dynamic predicates and for static predicates, if source mode was on when they were compiled:

**clause(+H,B)** [ISO]

A clause whose head matches \( H \) is searched for in the program. Its head and body are respectively unified with \( H \) and \( B \). If the clause is a unit clause, \( B \) is unified with true.

This predicate is applicable to static procedures compiled with source active, and to all dynamic procedures.

**clause(+H,B,-R)**

The same as clause/2, plus \( R \) is unified with the reference to the clause in the database. You can use instance/2 to access the reference’s value. Note that you may not use erase/1 on the reference on static procedures.
nth_clause(+H,I,-R)
Find the Ith clause in the predicate defining H, and give a reference to the clause. Alternatively, if the reference R is given the head H is unified with a description of the predicate and I is bound to its position.

The following predicates can only be used for dynamic predicates:

retract(+C) [ISO]
Erases the first clause in the program that matches C. This predicate may also be used for the static predicates that have been compiled when the source mode was on. For more information on source/0 (see Section 4.2 [Setting the Compiler], page 23).

retractall(+G)
Retract all the clauses whose head matches the goal G. Goal G must be a call to a dynamic predicate.

6.10.2 Looking at the Data Base

listing
Lists in the current output stream all the clauses for which source code is available (these include all clauses for dynamic predicates and clauses for static predicates compiled when source mode was on).

listing(+P)
Lists predicate P if its source code is available.

portray_clause(+C)
Write clause C as if written by listing/0.

portray_clause(+S,+C)
Write clause C on stream S as if written by listing/0.

current_atom(A)
Checks whether A is a currently defined atom. It is used to find all currently defined atoms by backtracking.

current_predicate(F) [ISO]
F is the predicate indicator for a currently defined user or library predicate. F is of the form Na/Ar, where the atom Na is the name of the predicate, and Ar its arity.

current_predicate(A,P)
Defines the relation: P is a currently defined predicate whose name is the atom A.

system_predicate(A,P)
Defines the relation: P is a built-in predicate whose name is the atom A.

predicate_property(P,Prop)
For the predicates obeying the specification P unify Prop with a property of P. These properties may be:

  built_in  true for built-in predicates,
  dynamic  true if the predicate is dynamic
static        true if the predicate is static

meta_predicate(M)
    true if the predicate has a meta_predicate declaration M.

multifile    true if the predicate was declared to be multifile

imported_from(Mod)
    true if the predicate was imported from module Mod.

exported     true if the predicate is exported in the current module.

public       true if the predicate is public; note that all dynamic predicates are public.

tabled       true if the predicate is tabled; note that only static predicates can be tabled in YAP.

source       true if source for the predicate is available.

number_of_clauses(ClauseCount)
    Number of clauses in the predicate definition. Always one if external or built-in.

6.10.3 Using Data Base References

Data Base references are a fast way of accessing terms. The predicates erase/1 and instance/1 also apply to these references and may sometimes be used instead of retract/1 and clause/2.

assert(+C,-R)
The same as assert(C) (see Section 6.10.1 [Modifying the Database], page 71) but unifies R with the database reference that identifies the new clause, in a one-to-one way. Note that asserta/2 only works for dynamic predicates. If the predicate is undefined, it will automatically be declared dynamic.

asserta(+C,-R)
The same as asserta(C) but unifying R with the database reference that identifies the new clause, in a one-to-one way. Note that asserta/2 only works for dynamic predicates. If the predicate is undefined, it will automatically be declared dynamic.

assertz(+C,-R)
The same as assertz(C) but unifying R with the database reference that identifies the new clause, in a one-to-one way. Note that asserta/2 only works for dynamic predicates. If the predicate is undefined, it will automatically be declared dynamic.

retract(+C,-R)
Erases from the program the clause C whose database reference is R. The predicate must be dynamic.
6.11 Internal Data Base

Some programs need global information for, e.g. counting or collecting data obtained by backtracking. As a rule, to keep this information, the internal data base should be used instead of asserting and retracting clauses (as most novice programmers do). In YAP (as in some other Prolog systems) the internal data base (i.d.b. for short) is faster, needs less space and provides a better insulation of program and data than using asserted/retracted clauses. The i.d.b. is implemented as a set of terms, accessed by keys that unlikely what happens in (non-Prolog) data bases are not part of the term. Under each key a list of terms is kept. References are provided so that terms can be identified: each term in the i.d.b. has a unique reference (references are also available for clauses of dynamic predicates).

\begin{verbatim}
recorda(+K,T,-R)
  Makes term T the first record under key K and unifies R with its reference.

recordz(+K,T,-R)
  Makes term T the last record under key K and unifies R with its reference.

recorda_at(+R0,T,-R)
  Makes term T the record preceding record with reference R0, and unifies R with its reference.

recordz_at(+R0,T,-R)
  Makes term T the record following record with reference R0, and unifies R with its reference.

recordaifnot(+K,T,-R)
  If a term equal to T up to variable renaming is stored under key K fail. Otherwise, make term T the first record under key K and unify R with its reference.

recordzifnot(+K,T,-R)
  If a term equal to T up to variable renaming is stored under key K fail. Otherwise, make term T the first record under key K and unify R with its reference.

recorded(+K,T,R)
  Searches in the internal database under the key K, a term that unifies with T and whose reference matches R. This built-in may be used in one of two ways:
  \begin{itemize}
    \item K may be given, in this case the built-in will return all elements of the internal data-base that match the key.
    \item R may be given, if so returning the key and element that match the reference.
  \end{itemize}

erase(+R)
  The term referred to by R is erased from the internal database. If reference R does not exist in the database, erase just fails.

erased(+R)
  Succeeds if the object whose database reference is R has been erased.

instance(+R,-T)
  If R refers to a clause or a recorded term, T is unified with its most general instance. If R refers to an unit clause C, then T is unified with C :- true.
\end{verbatim}
When $R$ is not a reference to an existing clause or to a recorded term, this goal fails.

**eraseall(+K)**

All terms belonging to the key $K$ are erased from the internal database. The predicate always succeeds.

**current_key(?A,?K)**

Defines the relation: $K$ is a currently defined database key whose name is the atom $A$. It can be used to generate all the keys for the internal data-base.

**nth_instance(?Key,?Index,?R)**

Fetches the $Index$th entry in the internal database under the key $Key$. Entries are numbered from one. If the key $Key$ or the $Index$ are bound, a reference is unified with $R$. Otherwise, the reference $R$ must be given, and YAP will find the matching key and index.

**nth_instance(?Key,?Index,T,?R)**

Fetches the $Index$th entry in the internal database under the key $Key$. Entries are numbered from one. If the key $Key$ or the $Index$ are bound, a reference is unified with $R$. Otherwise, the reference $R$ must be given, and YAP will find the matching key and index.

**key_statistics(+K,-Entries,-Size,-IndexSize)**

Returns several statistics for a key $K$. Currently, it says how many entries we have for that key, $Entries$, what is the total size spent on entries, $Size$, and what is the amount of space spent in indices.

**key_statistics(+K,-Entries,-TotalSize)**

Returns several statistics for a key $K$. Currently, it says how many entries we have for that key, $Entries$, what is the total size spent on this key.

**get_value(+A,-V)**

In YAP, atoms can be associated with constants. If one such association exists for atom $A$, unify the second argument with the constant. Otherwise, unify $V$ with $[]$.

This predicate is YAP specific.

**set_value(+A,+C)**

Associate atom $A$ with constant $C$.

The **set_value** and **get_value** built-ins give a fast alternative to the internal data-base. This is a simple form of implementing a global counter.

```prolog
read_and_increment_counter(Value) :-
    get_value(counter, Value),
    Value1 is Value+1,
    set_value(counter, Value1).
```

This predicate is YAP specific.

**recordzifnot(+K,T,-R)**

If a variant of $T$ is stored under key $K$ fail. Otherwise, make term $T$ the last record under key $K$ and unify $R$ with its reference.

This predicate is YAP specific.
recordaifnot(+K,T,-R)
If a variant of T is stored under key K fail. Otherwise, make term T the first
record under key K and unify R with its reference.
This predicate is YAP specific.

There is a strong analogy between the i.d.b. and the way dynamic predicates are stored.
In fact, the main i.d.b. predicates might be implemented using dynamic predicates:
recorda(X,T,R) :- asserta(idb(X,T),R).
recordz(X,T,R) :- assertz(idb(X,T),R).
recorded(X,T,R) :- clause(idb(X,T),R).

We can take advantage of this, the other way around, as it is quite easy to write a simple
Prolog interpreter, using the i.d.b.:
asserta(G) :- recorda(interpreter,G,_,).  
assertz(G) :- recordz(interpreter,G,_,).  
retract(G) :- recorded(interpreter,G,R), !, erase(R).  
call(V) :- var(V), !, fail.  
call((H :- B)) :- !, recorded(interpreter,(H :- B),_), call(B).  
call(G) :- recorded(interpreter,G,_,).

In YAP, much attention has been given to the implementation of the i.d.b., especially to the
problem of accelerating the access to terms kept in a large list under the same key. Besides
using the key, YAP uses an internal lookup function, transparent to the user, to find only
the terms that might unify. For instance, in a data base containing the terms
b
b(a)
c(d)
e(g)
b(X)
e(h)
stored under the key k/1, when executing the query
:- recorded(k(_),c(_),R).
recorded would proceed directly to the third term, spending almost the time as if a(X) or
b(X) was being searched. The lookup function uses the functor of the term, and its first
three arguments (when they exist). So, recorded(k(_,e(h),_),_) would go directly to the
last term, while recorded(k(_,e(_),_),_) would find first the fourth term, and then, after
backtracking, the last one.

This mechanism may be useful to implement a sort of hierarchy, where the functors of
the terms (and eventually the first arguments) work as secondary keys.

In the YAP’s i.d.b. an optimized representation is used for terms without free variables.
This results in a faster retrieval of terms and better space usage. Whenever possible, avoid
variables in terms in terms stored in the i.d.b.

6.12 The Blackboard

YAP implements a blackboard in the style of the SICStus Prolog blackboard. The black-
board uses the same underlying mechanism as the internal data-base but has several im-
portant differences:
• It is module aware, in contrast to the internal data-base.
• Keys can only be atoms or integers, and not compound terms.
• A single term can be stored per key.
• An atomic update operation is provided; this is useful for parallelism.

\[\text{bb\_put}(\text{+}\text{Key}, \text{?}\text{Term})\]
Store term table \text{Term} in the blackboard under key \text{Key}. If a previous term was stored under key \text{Key} it is simply forgotten.

\[\text{bb\_get}(\text{+}\text{Key}, \text{?}\text{Term})\]
Unify \text{Term} with a term stored in the blackboard under key \text{Key}, or fail silently if no such term exists.

\[\text{bb\_delete}(\text{+}\text{Key}, \text{?}\text{Term})\]
Delete any term stored in the blackboard under key \text{Key} and unify it with \text{Term}. Fail silently if no such term exists.

\[\text{bb\_update}(\text{+}\text{Key}, \text{?}\text{Term}, \text{?}\text{New})\]
Atomically unify a term stored in the blackboard under key \text{Key} with \text{Term}, and if the unification succeeds replace it by \text{New}. Fail silently if no such term exists or if unification fails.

### 6.13 Collecting Solutions to a Goal

When there are several solutions to a goal, if the user wants to collect all the solutions he may be led to use the data base, because backtracking will forget previous solutions.

YAP allows the programmer to choose from several system predicates instead of writing his own routines. \texttt{findall/3} gives you the fastest, but crudest solution. The other built-in predicates post-process the result of the query in several different ways:

\[\text{findall}(\text{T}, \text{+}\text{G}, \text{-}\text{L}) \text{ [ISO]}\]
Unifies \text{L} with a list that contains all the instantiations of the term \text{T} satisfying the goal \text{G}.

With the following program:
\[
\begin{align*}
\text{a}(2,1). \\
\text{a}(1,1). \\
\text{a}(2,2).
\end{align*}
\]
the answer to the query
\[
\text{findall}(X, \text{a}(X,Y), L).
\]
would be:
\[
\begin{align*}
X &= _{32} \\
Y &= _{33} \\
L &= [2,1,2]; \\
\text{no}
\end{align*}
\]

\[\text{findall}(\text{T}, \text{+}\text{G}, \text{+}\text{L}, \text{-}\text{L0})\]
Similar to \texttt{findall/3}, but appends all answers to list \text{L0}.
all(\(T, +G, -L\))

Similar to `findall(T, G, L)` but eliminate repeated elements. Thus, assuming the same clauses as in the above example, the reply to the query

\[\text{all}(X, a(X,Y), L).\]

would be:

\[X = _32\]
\[Y = _33\]
\[L = [2,1];\]
\[\text{no}\]

Note that `all/3` will fail if no answers are found.

`bagof(T, +G, -L)` [ISO]

For each set of possible instances of the free variables occurring in \(G\) but not in \(T\), generates the list \(L\) of the instances of \(T\) satisfying \(G\). Again, assuming the same clauses as in the examples above, the reply to the query

\[\text{bagof}(X, a(X,Y), L).\]

would be:

\[X = _32\]
\[Y = 1\]
\[L = [2,1];\]
\[X = _32\]
\[Y = 2\]
\[L = [2];\]
\[\text{no}\]

`setof(X, +P, -B)` [ISO]

Similar to `bagof(T, G, L)` but sorting list \(L\) and keeping only one copy of each element. Again, assuming the same clauses as in the examples above, the reply to the query

\[\text{setof}(X, a(X,Y), L).\]

would be:

\[X = _32\]
\[Y = 1\]
\[L = [1,2];\]
\[X = _32\]
\[Y = 2\]
\[L = [2];\]
\[\text{no}\]

### 6.14 Grammar Rules

Grammar rules in Prolog are both a convenient way to express definite clause grammars and an extension of the well known context-free grammars.

A grammar rule is of the form:

\[\text{head} \rightarrow \text{body}\]
where both head and body are sequences of one or more items linked by the standard conjunction operator ",".

Items can be:

- a non-terminal symbol may be either a complex term or an atom.
- a terminal symbol may be any Prolog symbol. Terminals are written as Prolog lists.
- an empty body is written as the empty list '[]'.
- extra conditions may be inserted as Prolog procedure calls, by being written inside curly brackets '{' and '}'.
- the left side of a rule consists of a nonterminal and an optional list of terminals.
- alternatives may be stated in the right-hand side of the rule by using the disjunction operator ';'.
- the cut and conditional symbol ('->') may be inserted in the right hand side of a grammar rule.

Grammar related built-in predicates:

**CurrentModule:expand_term(T,-X)**

user:expand_term(T,-X)

This predicate is used by YAP for preprocessing each top level term read when consulting a file and before asserting or executing it. It rewrites a term T to a term X according to the following rules: first try term_expansion/2 in the current module, and then try to use the user defined predicate user:term_expansion/2. If this call fails then the translating process for DCG rules is applied, together with the arithmetic optimizer whenever the compilation of arithmetic expressions is in progress.

**CurrentModule:goal_expansion(+G,+M,-NG)**

user:goal_expansion(+G,+M,-NG)

YAP now supports goal_expansion/3. This is an user-defined procedure that is called after term expansion when compiling or asserting goals for each subgoal in a clause. The first argument is bound to the goal and the second to the module under which the goal G will execute. If goal_expansion/3 succeeds the new sub-goal NG will replace G and will be processed in the same way. If goal_expansion/3 fails the system will use the default rules.

**phrase(+P,L,R)**

This predicate succeeds when the difference list \(L-R\) is a phrase of type \(P\).

**phrase(+P,L)**

This predicate succeeds when \(L\) is a phrase of type \(P\). The same as phrase(P,L,[[]]).

Both this predicate and the previous are used as a convenient way to start execution of grammar rules.

**'C'(S1,T,S2)**

This predicate is used by the grammar rules compiler and is defined as 

\[ 'C'(\{H|T\},H,T) \]
6.15 Access to Operating System Functionality

The following built-in predicates allow access to underlying Operating System functionality:

\texttt{cd(+D)} \hspace{1em} \text{Changes the current directory (on UNIX environments).}

\texttt{environ(+E,-S)}
\hspace{1em} \text{Given an environment variable } E \text{ this predicate unifies the second argument } S \text{ with its value.}

\texttt{getcwd(-D)}
\hspace{1em} \text{Unify the current directory, represented as an atom, with the argument } D.

\texttt{putenv(+E,+S)}
\hspace{1em} \text{Set environment variable } E \text{ to the value } S. \text{ If the environment variable } E \text{ does not exist, create a new one. Both the environment variable and the value must be atoms.}

\texttt{rename(+F,+G)}
\hspace{1em} \text{Renames file } F \text{ to } G.

\texttt{sh}
\hspace{1em} \text{Creates a new shell interaction.}

\texttt{system(+S)}
\hspace{1em} \text{Passes command } S \text{ to the Bourne shell (on UNIX environments) or the current command interpreter in WIN32 environments.}

\texttt{unix(+S)} \hspace{1em} \text{Access to Unix-like functionality:}

\texttt{argv/1} \hspace{1em} \text{Return a list of arguments to the program. These are the arguments that follow a \texttt{-}, as in the usual Unix convention.}

\texttt{cd/0} \hspace{1em} \text{Change to home directory.}

\texttt{cd/1} \hspace{1em} \text{Change to given directory. Acceptable directory names are strings or atoms.}

\texttt{environ/2} \hspace{1em} \text{If the first argument is an atom, unify the second argument with the value of the corresponding environment variable.}

\texttt{getcwd/1} \hspace{1em} \text{Unify the first argument with an atom representing the current directory.}

\texttt{putenv/2} \hspace{1em} \text{Set environment variable } E \text{ to the value } S. \text{ If the environment variable } E \text{ does not exist, create a new one. Both the environment variable and the value must be atoms.}

\texttt{shell/1} \hspace{1em} \text{Execute command under current shell. Acceptable commands are strings or atoms.}

\texttt{system/1} \hspace{1em} \text{Execute command with } /bin/sh. \text{ Acceptable commands are strings or atoms.}

\texttt{shell/0} \hspace{1em} \text{Execute a new shell.}
alarm(+Seconds,+Callable,+OldAlarm)

Arranges for YAP to be interrupted in Seconds seconds, or in [Seconds|MicroSeconds]. When interrupted, YAP will execute Callable and then return to the previous execution. If Seconds is 0, no new alarm is scheduled. In any event, any previously set alarm is canceled.

The variable OldAlarm unifies with the number of seconds remaining until any previously scheduled alarm was due to be delivered, or with 0 if there was no previously scheduled alarm.

Note that execution of Callable will wait if YAP is executing built-in predicates, such as Input/Output operations.

The next example shows how alarm/3 can be used to implement a simple clock:

```prolog
loop :- loop.

ticker :- write('.'), flush_output,
        get_value(tick, yes),
        alarm(1,ticker,_).

:- set_value(tick, yes), alarm(1,ticker,_), loop.
```

The clock, ticker, writes a dot and then checks the flag tick to see whether it can continue ticking. If so, it calls itself again. Note that there is no guarantee that the each dot corresponds a second: for instance, if the YAP is waiting for user input, ticker will wait until the user types the entry in.

The next example shows how alarm/3 can be used to guarantee that a certain procedure does not take longer than a certain amount of time:

```prolog
loop :- loop.

:- catch((alarm(10, throw(ball), _),loop),
       ball,
       format('Quota exhausted.~n',[])).
```

In this case after 10 seconds our loop is interrupted, ball is thrown, and the handler writes Quota exhausted. Execution then continues from the handler.

Note that in this case loop/0 always executes until the alarm is sent. Often, the code you are executing succeeds or fails before the alarm is actually delivered. In this case, you probably want to disable the alarm when you leave the procedure. The next procedure does exactly so:

```prolog
once_with_alarm(Time,Goal,DoOnAlarm) :-
    catch(execute_once_with_alarm(Time, Goal), alarm, DoOnAlarm).
```

```prolog
execute_once_with_alarm(Time, Goal) :-
    alarm(Time, alarm, _),
    ( call(Goal) -> alarm(0, alarm, _) ; alarm(0, alarm, _), fail).
```

The procedure once_with_alarm/3 has three arguments: the Time to wait before the alarm is sent; the Goal to execute; and the goal DoOnAlarm to execute if the alarm is sent. It uses catch/3 to handle the case the alarm is
sent. Then it starts the alarm, calls the goal Goal, and disables the alarm on success or failure.

\[
on\text{signal}(+\text{Signal}, ?\text{OldAction}, +\text{Callable})
\]
Set the interrupt handler for soft interrupt \text{Signal} to be \text{Callable}. \text{OldAction} is unified with the previous handler.

Only a subset of the software interrupts (signals) can have their handlers manipulated through \text{on_signal}/3. Their POSIX names, YAP names and default behavior is given below. The "YAP name" of the signal is the atom that is associated with each signal, and should be used as the first argument to \text{on_signal}/3. It is chosen so that it matches the signal’s POSIX name.

\text{on_signal}/3 succeeds, unless when called with an invalid signal name or one that is not supported on this platform. No checks are made on the handler provided by the user.

\[
sig\text{_up}(\text{Hangup})
\]
\text{SIGHUP} in Unix/Linux; Reconsult the initialization files 
\~/.yaprc, \~/.prologrc and \~/.prolog.ini.

\[
sig\text{_usr1 and sig}\_\text{usr2}(\text{User signals})
\]
\text{SIGUSR1} and \text{SIGUSR2} in Unix/Linux; Print a message and halt.

A special case is made, where if \text{Callable} is bound to \text{default}, then the default handler is restored for that signal.

A call in the form \text{on_signal}(S,H,H) can be used to retrieve a signal’s current handler without changing it.

It must be noted that although a signal can be received at all times, the handler is not executed while YAP is waiting for a query at the prompt. The signal will be, however, registered and dealt with as soon as the user makes a query.

Please also note, that neither POSIX Operating Systems nor YAP guarantee that the order of delivery and handling is going to correspond with the order of dispatch.

### 6.16 Term Modification

It is sometimes useful to change the value of instantiated variables. Although, this is against the spirit of logic programming, it is sometimes useful. As in other Prolog systems, YAP has several primitives that allow updating Prolog terms. Note that these primitives are also backtrackable.

The \text{setarg}/3 primitive allows updating any argument of a Prolog compound terms. The \text{mutable} family of predicates provides \text{mutable variables}. They should be used instead of \text{setarg}/3, as they allow the encapsulation of accesses to updatable variables. Their implementation can also be more efficient for long deterministic computations.

\[
\text{setarg}(+I,+S, ?T)
\]
Set the value of the \text{I}th argument of term \text{S} to term \text{T}.

\[
\text{create\_mutable}(+D, -M)
\]
Create new mutable variable \text{M} with initial value \text{D}. 
get Mutable (?D, +M)
Unify the current value of mutable term M with term D.

is Mutable (?D)
Holds if D is a mutable term.

get Mutable (?D, +M)
Unify the current value of mutable term M with term D.

update Mutable (+D, +M)
Set the current value of mutable term M to term D.

6.17 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 from 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 to 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 called exception/3. The latter technique is used by CHR.

b_setval(+Name, +Value)
Associate the term Value with the atom Name or replaces 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.

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.
Notice that for compatibility with other systems Name must be already associated with a term: otherwise the system will generate an error.

\[ \text{nb_setval(+Name, +Value)} \]
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.

\[ \text{nb_getval(+Name, -Value)} \]
The \text{nb_getval/2} predicate is a synonym for \text{b_getval/2}, introduced for compatibility and symmetry. As most scenarios will use a particular global variable either using non-backtrackable or backtrackable assignment, using \text{nb_getval/2} can be used to document that the variable is used non-backtrackable.

\[ \text{nb_linkval(+Name, +Value)} \]
Associates the term Value with the atom Name without copying it. This is a fast special-purpose variation of \text{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. Please consider the following example:

\[
\text{demo_nb_linkval : -}
\text{T = nice(N),}
( \text{ N = world,}
\text{ nb_linkval(myvar, T),}
\text{ fail}
\text{ ; nb_getval(myvar, V),}
\text{ writeln(V)}
\).
\]

\[ \text{nb_set_shared_val(+Name, +Value)} \]
Associates the term Value with the atom Name, but sharing non-backtrackable terms. This may be useful if you want to rewrite a global variable so that the new copy will survive backtracking, but you want to share structure with the previous term.

The next example shows the differences between the three built-ins:

\[
?- \text{nb_setval(a,a(_)),nb_getval(a,A),nb_setval(b,t(C,A)),nb_getval(b,B).} \]
\text{A = a(_A),}
\text{B = t(_B,a(_C)) ?}
\]

\[
?- \text{nb_setval(a,a(_)),nb_getval(a,A),nb_set_shared_val(b,t(C,A)),nb_getval(b,B).} \]
\]

\[
?- \text{nb_setval(a,a(_)),nb_getval(a,A),nb_linkval(b,t(C,A)),nb_getval(b,B).} \]
\text{A = a(_A),}
\text{B = t(C,a(_A)) ?}
\]
**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 Term 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 to 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.

```prolog
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_set_shared_arg(+Arg, +Term, +Value)**

As nb_setarg/3, but like nb_linkval/2 it does not duplicate the global sub-terms in Value. Use with extreme care and consult the documentation of nb_linkval/2 before use.

**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.

**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.

Global variables have been introduced by various Prolog implementations recently. We follow the implementation of them in SWI-Prolog, itself based on hProlog by Bart Demoen.

GNU-Prolog provides a rich set of global variables, including arrays. Arrays can be implemented easily in YAP and SWI-Prolog using functor/3 and setarg/3 due to the unrestricted arity of compound terms.

### 6.18 Profiling Prolog Programs

YAP includes two profilers. The count profiler keeps information on the number of times a predicate was called. This information can be used to detect what are the most commonly called predicates in the program. The count profiler can be compiled by setting YAP’s flag profiling to on. The time-profiler is a gprof profiler, and counts how many ticks are
being spent on specific predicates, or on other system functions such as internal data-base accesses or garbage collects.

The YAP profiling sub-system is currently under development. Functionality for this sub-system will increase with newer implementation.

### 6.18.1 The Count Profiler

**Notes:**

The count profiler works by incrementing counters at procedure entry or backtracking. It provides exact information:

- Profiling works for both static and dynamic predicates.
- Currently only information on entries and retries to a predicate are maintained. This may change in the future.
- As an example, the following user-level program gives a list of the most often called procedures in a program. The procedure `list_profile` shows all procedures, irrespective of module, and the procedure `list_profile/1` shows the procedures being used in a specific module.

```prolog
list_profile :-
  % get number of calls for each profiled procedure
  setof(D-[M:P|D1],(current_module(M),profile_data(M:P,calls,D),profile_data(M:P,retries,D1)),LP),
  % output so that the most often called
  % predicates will come last:
  write_profile_data(LP).

list_profile(Module) :-
  % get number of calls for each profiled procedure
  setof(D-[Module:P|D1],(profile_data(Module:P,calls,D),profile_data(Module:P,retries,D1)),LP),
  % output so that the most often called
  % predicates will come last:
  write_profile_data(LP).

write_profile_data([]).
write_profile_data([D-[M:P|R]|SLP]) :-
  % swap the two calls if you want the most often
  % called predicates first.
  format('~a:~w: ~32+~t~d~12+~t~d~12+~n', [M,P,D,R]),
  write_profile_data(SLP).
```

These are the current predicates to access and clear profiling data:

**profile_data(\(Na/Ar\), \(Parameter\), -\(Data\))**

Give current profile data on \(Parameter\) for a predicate described by the predicate indicator \(Na/Ar\). If any of \(Na/Ar\) or \(Parameter\) are unbound, backtrack through all profiled predicates or stored parameters. Current parameters are:

- **calls** Number of times a procedure was called.
- **retries** Number of times a call to the procedure was backtracked to and retried.
profile_reset
    Reset all profiling information.

6.18.2 Tick Profiler
The tick profiler works by interrupting the Prolog code every so often and checking at each point the code was. The profiler must be able to retrace the state of the abstract machine at every moment. The major advantage of this approach is that it gives the actual amount of time being spent per procedure, or whether garbage collection dominates execution time. The major drawback is that tracking down the state of the abstract machine may take significant time, and in the worst case may slow down the whole execution.

The following procedures are available:
profinit   Initialise the data-structures for the profiler. Unnecessary for dynamic profiler.
profon     Start profiling.
proffoff   Stop profiling.
showprofres
    Show profiling info.
showprofres(N)
    Show profiling info for the top-most N predicates.

6.19 Counting Calls
Predicates compiled with YAP’s flag call_counting set to on update counters on the numbers of calls and of retries. Counters are actually decreasing counters, so that they can be used as timers. Three counters are available:
- calls: number of predicate calls since execution started or since system was reset;
- retries: number of retries for predicates called since execution started or since counters were reset;
- calls_and_retries: count both on predicate calls and retries.

These counters can be used to find out how many calls a certain goal takes to execute. They can also be used as timers.

The code for the call counters piggybacks on the profiling code. Therefore, activating the call counters also activates the profiling counters.

These are the predicates that access and manipulate the call counters:
call_count_data(-Calls, -Retries, -CallsAndRetries)
    Give current call count data. The first argument gives the current value for the Calls counter, next theRetries counter, and last the CallsAndRetries counter.
call_count_data_reset
    Reset call count counters. All timers are also reset.
call_count(?CallsMax, ?RetriesMax, ?CallsAndRetriesMax)
    Set call count counter as timers. YAP will generate an exception if one of the instantiated call counters decreases to 0. YAP will ignore unbound arguments:
• **CallsMax**: throw the exception `call_counter` when the counter `calls` reaches 0;
• **RetriesMax**: throw the exception `retry_counter` when the counter `retries` reaches 0;
• **CallsAndRetriesMax**: throw the exception `call_and_retry_counter` when the counter `calls_and_retries` reaches 0.

Next, we show a simple example of how to use call counters:

```prolog
?- yap_flag(call_counting, on), [\-user]. l :- l. end_of_file. yap_flag(call_counting, off).
yes

?- catch((call_count(10000, _, _), l), call_counter, format("limit_exceeded.\n", [])).
limit_exceeded.

yes
```

Notice that we first compile the looping predicate `l/0` with `call_counting on`. Next, we `catch/3` to handle an exception when `l/0` performs more than 10000 reductions.

### 6.20 Arrays

The YAP system includes experimental support for arrays. The support is enabled with the option `YAP_ARRAYS`.

There are two very distinct forms of arrays in YAP. The **dynamic arrays** are a different way to access compound terms created during the execution. Like any other terms, any bindings to these terms and eventually the terms themselves will be destroyed during backtracking. Our goal in supporting dynamic arrays is twofold. First, they provide an alternative to the standard `arg/3` built-in. Second, because dynamic arrays may have name that are globally visible, a dynamic array can be visible from any point in the program. In more detail, the clause

```
g(X) :- array_element(a, 2, X).
```

will succeed as long as the programmer has used the built-in `array/2` to create an array term with at least 3 elements in the current environment, and the array was associated with the name `a`. The element `X` is a Prolog term, so one can bind it and any such bindings will be undone when backtracking. Note that dynamic arrays do not have a type: each element may be any Prolog term.

The **static arrays** are an extension of the database. They provide a compact way for manipulating data-structures formed by characters, integers, or floats imperatively. They can also be used to provide two-way communication between YAP and external programs through shared memory.

In order to efficiently manage space elements in a static array must have a type. Currently, elements of static arrays in YAP should have one of the following predefined types:

- **byte**: an 8-bit signed character.
• **unsigned_byte**: an 8-bit unsigned character.
• **int**: Prolog integers. Size would be the natural size for the machine’s architecture.
• **float**: Prolog floating point number. Size would be equivalent to a double in C.
• **atom**: a Prolog atom.
• **dref**: an internal database reference.
• **term**: a generic Prolog term. Note that this will term will not be stored in the array itself, but instead will be stored in the Prolog internal database.

Arrays may be **named** or **anonymous**. Most arrays will be named, that is associated with an atom that will be used to find the array. Anonymous arrays do not have a name, and they are only of interest if the `TERM_EXTENSIONS` compilation flag is enabled. In this case, the unification and parser are extended to replace occurrences of Prolog terms of the form `X[I]` by run-time calls to `array_element/3`, so that one can use array references instead of extra calls to `arg/3`. As an example:

\[
g(X,Y,Z,I,J) :- X[I] \text{ is } Y[J]+Z[I].
\]

should give the same results as:

\[
G(X,Y,Z,I,J) :-
\begin{align*}
    & \text{array_element}(X,I,E1), \\
    & \text{array_element}(Y,J,E2), \\
    & \text{array_element}(Z,I,E3), \\
    & E1 \text{ is } E2+E3.
\end{align*}
\]

Note that the only limitation on array size are the stack size for dynamic arrays; and, the heap size for static (not memory mapped) arrays. Memory mapped arrays are limited by available space in the file system and in the virtual memory space.

The following predicates manipulate arrays:

**array(+Name, +Size)**

Creates a new dynamic array. The Size must evaluate to an integer. The Name may be either an atom (named array) or an unbound variable (anonymous array).

Dynamic arrays work as standard compound terms, hence space for the array is recovered automatically on backtracking.

**static_array(+Name, +Size, +Type)**

Create a new static array with name Name. Note that the Name must be an atom (named array). The Size must evaluate to an integer. The Type must be bound to one of types mentioned previously.

**reset_static_array(+Name)**

Reset static array with name Name to its initial value.

**static_array_location(+Name, -Ptr)**

Give the location for a static array with name Name.

**static_array_properties(?Name, ?Size, ?Type)**

Show the properties size and type of a static array with name Name. Can also be used to enumerate all current static arrays.

This built-in will silently fail if the there is no static array with that name.
static_array_to_term(?Name, ?Term)
Convert a static array with name Name to a compound term of name Name.
This built-in will silently fail if there is no static array with that name.

mmapped_array(+Name, +Size, +Type, +File)
Similar to static_array/3, but the array is memory mapped to file File. This
means that the array is initialized from the file, and that any changes to the
array will also be stored in the file.
This built-in is only available in operating systems that support the system call
mmap. Moreover, mmapped arrays do not store generic terms (type term).

close_static_array(+Name)
Close an existing static array of name Name. The Name must be an atom
(named array). Space for the array will be recovered and further accesses to
the array will return an error.

resize_static_array(+Name, -OldSize, +NewSize)
Expand or reduce a static array. The Size must evaluate to an integer. The
Name must be an atom (named array). The Type must be bound to one of
int, ddbref, float or atom.
Note that if the array is a mmapped array the size of the mmapped file will be
actually adjusted to correspond to the size of the array.

array_element(+Name, +Index, ?Element)
Unify Element with Name[Index]. It works for both static and dynamic arrays,
but it is read-only for static arrays, while it can be used to unify with an element
of a dynamic array.

update_array(+Name, +Index, ?Value)
Attribute value Value to Name[Index]. Type restrictions must be respected
for static arrays. This operation is available for dynamic arrays if MULTI-
ASSIGNMENT_VARIABLES is enabled (true by default). Backtracking undoes update_array/3 for dynamic arrays, but not for static arrays.
Note that update_array/3 actually uses setarg/3 to update elements of dyn-
amic arrays, and setarg/3 spends an extra cell for every update. For intensive
operations we suggest it may be less expensive to unify each element of the array
with a mutable terms and to use the operations on mutable terms.

add_to_array_element(+Name, +Index, , +Number, ?NewValue)
Add Number Name[Index] and unify NewValue with the incremented value.
Observe that Name[Index] must be an number. If Name is a static array the
type of the array must be int or float. If the type of the array is int you
only may add integers, if it is float you may add integers or floats. If Name
corresponds to a dynamic array the array element must have been previously
bound to a number and Number can be any kind of number.
The add_to_array_element/3 built-in actually uses setarg/3 to update ele-
ments of dynamic arrays. For intensive operations we suggest it may be less
expensive to unify each element of the array with a mutable terms and to use
the operations on mutable terms.
6.21 Predicate Information

Built-ins that return information on the current predicates and modules:

\( \text{current_module}(M) \)

Succeeds if \( M \) are defined modules. A module is defined as soon as some predicate defined in the module is loaded, as soon as a goal in the module is called, or as soon as it becomes the current type-in module.

\( \text{current_module}(M,F) \)

Succeeds if \( M \) are current modules associated to the file \( F \).

6.22 Miscellaneous

\( \text{statistics}/0 \)

Send to the current user error stream general information on space used and time spent by the system.

?- statistics.

<table>
<thead>
<tr>
<th>Location</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory (total)</td>
<td>4784124 bytes</td>
</tr>
<tr>
<td>program space</td>
<td>3055616 bytes: 1392224 in use, 1663392 free</td>
</tr>
<tr>
<td>stack space</td>
<td>1531904 bytes: 464 in use, 1531440 free</td>
</tr>
<tr>
<td>global stack:</td>
<td>96 in use, 616684 max</td>
</tr>
<tr>
<td>local stack:</td>
<td>368 in use, 546208 max</td>
</tr>
<tr>
<td>trail stack</td>
<td>196604 bytes: 8 in use, 196596 free</td>
</tr>
</tbody>
</table>

0.010 sec. for 5 code, 2 stack, and 1 trail space overflows
0.130 sec. for 3 garbage collections which collected 421000 bytes
0.000 sec. for 0 atom garbage collections which collected 0 bytes
0.880 sec. runtime
1.020 sec. cputime
25.055 sec. elapsed time

The example shows how much memory the system spends. Memory is divided into Program Space, Stack Space and Trail. In the example we have 3MB allocated for program spaces, with less than half being actually used. YAP also shows the maximum amount of heap space having been used which was over 2MB.

The stack space is divided into two stacks which grow against each other. We are in the top level so very little stack is being used. On the other hand, the system did use a lot of global and local stack during the previous execution (we refer the reader to a WAM tutorial in order to understand what are the global and local stacks).

YAP also shows information on how many memory overflows and garbage collections the system executed, and statistics on total execution time. Cputime includes all running time, runtime excludes garbage collection and stack overflow time.
statistics(\(\text{Param},-\text{Info}\))

Gives statistical information on the system parameter given by first argument:

- **atoms** \([\text{NumberOfAtoms}, \text{SpaceUsedBy Atoms}]\)
  - This gives the total number of atoms \(\text{NumberOfAtoms}\) and how much space they require in bytes, \(\text{SpaceUsedBy Atoms}\).

- **cputime** \([\text{Time since Boot}, \text{Time From Last Call to Cputime}]\)
  - This gives the total cputime in milliseconds spent executing Prolog code, garbage collection and stack shifts time included.

- **dynamic_code**
  - \([\text{Clause Size}, \text{Index Size}, \text{Tree Index Size}, \text{Choice Point Instructions Size}, \text{Expansion Nodes Size}, \text{Index Switch Size}]\)
  - Size of static code in YAP in bytes: \(\text{Clause Size}\), the number of bytes allocated for clauses, plus \(\text{Index Size}\), the number of bytes spent in the indexing code. The indexing code is divided into main tree, \(\text{Tree Index Size}\), tables that implement choice-point manipulation, \(\text{Choice Point Instructions Size}\), tables that cache clauses for future expansion of the index tree, \(\text{Expansion Nodes Size}\), and tables such as hash tables that select according to value, \(\text{Index Switch Size}\).

- **garbage_collection**
  - \([\text{Number of GCs}, \text{Total Global Recovered}, \text{Total Time Spent}]\)
  - Number of garbage collections, amount of space recovered in kbytes, and total time spent doing garbage collection in milliseconds. More detailed information is available using \text{yap_flag(gc_trace,verbose)}.

- **global_stack**
  - \([\text{Global Stack Used}, \text{Execution Stack Free}]\)
  - Space in kbytes currently used in the global stack, and space available for expansion by the local and global stacks.

- **local_stack**
  - \([\text{Local Stack Used}, \text{Execution Stack Free}]\)
  - Space in kbytes currently used in the local stack, and space available for expansion by the local and global stacks.

- **heap** \([\text{Heap Used}, \text{Heap Free}]\)
  - Total space in kbytes not recoverable in backtracking. It includes the program code, internal data base, and, atom symbol table.

- **program** \([\text{Program Space Used}, \text{Program Space Free}]\)
  - Equivalent to heap.

- **runtime** \([\text{Time since Boot}, \text{Time From Last Call to Runtime}]\)
  - This gives the total cputime in milliseconds spent executing Prolog code, not including garbage collections and stack shifts. Note that until YAP4.1.2 the **runtime** statistics would return time spent on garbage collection and stack shifting.
stack_shifts

[Number of Heap Shifts, Number of Stack Shifts, Number of Trail Shifts]
Number of times YAP had to expand the heap, the stacks, or the trail. More detailed information is available using yap_flag(gc_trace, verbose).

static_code

[Clause Size, Index Size, Tree Index Size, Expansion Nodes Size, Index Switch Size]
Size of static code in YAP in bytes: Clause Size, the number of bytes allocated for clauses, plus Index Size, the number of bytes spent in the indexing code. The indexing code is divided into a main tree, Tree Index Size, table that cache clauses for future expansion of the index tree, Expansion Nodes Size, and and tables such as hash tables that select according to value, Index Switch Size.

trail

[Trail Used, Trail Free]
Space in kbytes currently being used and still available for the trail.

walltime

[Time since Boot, Time From Last Call to Walltime]
This gives the clock time in milliseconds since starting Prolog.

time(:Goal)

Prints the CPU time and the wall time for the execution of Goal. Possible choice-points of Goal are removed. Based on the SWI-Prolog definition (minus reporting the number of inferences, which YAP currently does not support).

yap_flag(?Param, ?Value)

Set or read system properties for Param:

argv
Read-only flag. It unifies with a list of atoms that gives the arguments to YAP after --.

agc_margin
An integer: 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.

bounded [ISO]
Read-only flag telling whether integers are bounded. The value depends on whether YAP uses the GMP library or not.

profiling
If off (default) do not compile call counting information for procedures. If on compile predicates so that they calls and retries to the predicate may be counted. Profiling data can be read through the call_count_data/3 built-in.
char_conversion [ISO]

Writable flag telling whether a character conversion table is used when reading terms. The default value for this flag is off except in sicstus and iso language modes, where it is on.

character_escapes [ISO]

Writable flag telling whether a character escapes are enabled, on, or disabled, off. The default value for this flag is on.

debug [ISO]

If Value is unbound, tell whether debugging is on or off. If Value is bound to on enable debugging, and if it is bound to off disable debugging.

+debugger_print_options

If bound, set the argument to the write_term/3 options the debugger uses to write terms. If unbound, show the current options.

dialect

Read-only flag that always returns yap.

discontiguous_warnings

If Value is unbound, tell whether warnings for discontiguous predicates are on or off. If Value is bound to on enable these warnings, and if it is bound to off disable them. The default for YAP is off, unless we are in sicstus or iso mode.

dollar_as_lower_case

If off (default) consider the character '§' a control character, if on consider '§' a lower case character.

double_quotes [ISO]

If Value is unbound, tell whether a double quoted list of characters token is converted to a list of atoms, chars, to a list of integers, codes, or to a single atom, atom. If Value is bound, set to the corresponding behavior. The default value is codes.

executable

Read-only flag. It unifies with an atom that gives the original program path.

fast

If on allow fast machine code, if off (default) disable it. Only available in experimental implementations.
fileerrors

If on fileerrors is on, if off (default) fileerrors is disabled.

float_format

C-library printf() format specification used by write/1 and friends to determine how floating point numbers are printed. The default is %.15g. The specified value is passed to printf() without further checking. For example, if you want less digits printed, %g will print all floats using 6 digits instead of the default 15.

gc

If on allow garbage collection (default), if off disable it.

gc_margin

Set or show the minimum free stack before starting garbage collection. The default depends on total stack size.

gc_trace

If off (default) do not show information on garbage collection and stack shifts, if on inform when a garbage collection or stack shift happened, if verbose give detailed information on garbage collection and stack shifts. Last, if very_verbose give detailed information on data-structures found during the garbage collection process, namely, on choice-points.

generate_debugging_info

If true (default) generate debugging information for procedures, including source mode. If false predicates no information is generated, although debugging is still possible, and source mode is disabled.

host_type

Return configure system information, including the machine-id for which YAP was compiled and Operating System information.

index

If on allow indexing (default), if off disable it.

informational_messages

If on allow printing of informational messages, such as the ones that are printed when consulting. If off disable printing these messages. It is on by default except if YAP is booted with the -L flag.
integer_rounding_function [ISO]

Read-only flag telling the rounding function used for integers. Takes the value `down` for the current version of YAP.

language

Choose whether YAP is closer to C-Prolog, `cprolog`, iso-prolog, `iso` or SICStus Prolog, `sicstus`. The current default is `cprolog`. This flag affects update semantics, leashing mode, style checking, handling calls to undefined procedures, how directives are interpreted, when to use dynamic, character escapes, and how files are consulted.

max_arity [ISO]

Read-only flag telling the maximum arity of a functor. Takes the value `unbounded` for the current version of YAP.

max_integer [ISO]

Read-only flag telling the maximum integer in the implementation. Depends on machine and Operating System architecture, and on whether YAP uses the GMP multi-precision library. If `bounded` is false, requests for `max_integer` will fail.

max_tagged_integer

Read-only flag telling the maximum integer we can store as a single word. Depends on machine and Operating System architecture. It can be used to find the word size of the current machine.

min_integer [ISO]

Read-only flag telling the minimum integer in the implementation. Depends on machine and Operating System architecture, and on whether YAP uses the GMP multi-precision library. If `bounded` is false, requests for `min_integer` will fail.

min_tagged_integer

Read-only flag telling the minimum integer we can store as a single word. Depends on machine and Operating System architecture.

n_of_integer_keys_in_bb

Read or set the size of the hash table that is used for looking up the blackboard when the key is an integer.

n_of_integer_keys_in_db

Read or set the size of the hash table that is used for looking up the internal data-base when the key is an integer.
open_expands_filename

If true the open/3 builtin performs filename-expansion before opening a file (SICStus Prolog like). If false it does not (SWI-Prolog like).

open_shared_object

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

profiling

If off (default) do not compile profiling information for procedures. If on compile predicates so that they will output profiling information. Profiling data can be read through the profile_data/3 built-in.

prompt_alternatives_on(atom, changeable)

SWI-Compatible option, determines prompting for alternatives in the Prolog toplevel. Default is groundness, YAP prompts for alternatives if and only if the query contains variables. The alternative, default in SWI-Prolog is determinism which implies the system prompts for alternatives if the goal succeeded while leaving choice-points.

redefine_warnings

If Value is unbound, tell whether warnings for procedures defined in several different files are on or off. If Value is bound to on enable these warnings, and if it is bound to off disable them. The default for YAP is off, unless we are in sicstus or iso mode.

shared_object_search_path

Name of the environment variable used by the system to search for shared objects.

single_var_warnings

If Value is unbound, tell whether warnings for singleton variables are on or off. If Value is bound to on enable these warnings, and if it is bound to off disable them. The default for YAP is off, unless we are in sicstus or iso mode.

strict_iso

If Value is unbound, tell whether strict ISO compatibility mode is on or off. If Value is bound to on set language mode to iso and enable strict mode. If Value is bound to off disable strict mode, and keep the current language mode. The default for YAP is off.
Under strict ISO Prolog mode all calls to non-ISO built-ins generate an error. Compilation of clauses that would call non-ISO built-ins will also generate errors. Pre-processing for grammar rules is also disabled. Module expansion is still performed.

Arguably, ISO Prolog does not provide all the functionality required from a modern Prolog system. Moreover, because most Prolog implementations do not fully implement the standard and because the standard itself gives the implementor latitude in a few important questions, such as the unification algorithm and maximum size for numbers there is no guarantee that programs compliant with this mode will work the same way in every Prolog and in every platform. We thus believe this mode is mostly useful when investigating how a program depends on a Prolog’s platform specific features.

`stack_dump_on_error`

If on show a stack dump when YAP finds an error. The default is off.

`syntax_errors`

Control action to be taken after syntax errors while executing `read/1`, `read/2`, or `read_term/3`:

- `dec10` Report the syntax error and retry reading the term.
- `fail` Report the syntax error and fail (default).
- `error` Report the syntax error and generate an error.
- `quiet` Just fail

`system_options`

This read only flag tells which options were used to compile YAP. Currently it informs whether the system supports `big_numbers`, `coroutining`, `depth_limit`, `low_level_tracer`, `or-parallelism`, `rational_trees`, `readline`, `tabling`, `threads`, or the `wam_profiler`.

`tabling_mode`

Sets or reads the tabling mode for all tabled predicates. Please see Chapter 17 [Tabling], page 193 for the list of options.

`to_chars_mode`

Define whether YAP should follow `quintus`-like semantics for the
atom_chars/1 or number_chars/1 built-in, or whether it should follow the ISO standard (iso option).

+ toplevel_hook

+If bound, set the argument to a goal to be executed before entering the top-level. If unbound show the current goal or true if none is presented. Only the first solution is considered and the goal is not backtracked into.

+ toplevel_print_options

+If bound, set the argument to the write_term/3 options used to write terms from the top-level. If unbound, show the current options.

typein_module

If bound, set the current working or type-in module to the argument, which must be an atom. If unbound, unify the argument with the current working module.

unix

Read-only Boolean flag that unifies with true if YAP is running on an Unix system. Defined if the C-compiler used to compile this version of YAP either defines __unix__ or unix.

unknown [ISO]

Corresponds to calling the unknown/2 built-in.

update_semantics

Define whether YAP should follow immediate update semantics, as in C-Prolog (default), logical update semantics, as in Quintus Prolog, SICStus Prolog, or in the ISO standard. There is also an intermediate mode, logical_assert, where dynamic procedures follow logical semantics but the internal data base still follows immediate semantics.

user_error

If the second argument is bound to a stream, set user_error to this stream. If the second argument is unbound, unify the argument with the current user_error stream.

By default, the user_error stream is set to a stream corresponding to the Unix stderr stream.

The next example shows how to use this flag:
?- open( '/dev/null', append, Error, [alias(mauri_tripa)])).

Error = '$stream'(3) ? ;
no
?- set_prolog_flag(user_error, mauri_tripa).
close(mauri_tripa).
yes
?- 

We execute three commands. First, we open a stream in write mode and give it an alias, in this case mauri_tripa. Next, we set user_error to the stream via the alias. Note that after we did so prompts from the system were redirected to the stream mauri_tripa. Last, we close the stream. At this point, YAP automatically redirects the user_error alias to the original stderr.

user_flags

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, with warning printing 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 error, and developers are encouraged to use create_prolog_flag/3 to create flags for their library.

user_input

If the second argument is bound to a stream, set user_input to this stream. If the second argument is unbound, unify the argument with the current user_input stream.

By default, the user_input stream is set to a stream corresponding to the Unix stdin stream.

user_output

If the second argument is bound to a stream, set user_output to this stream. If the second argument is unbound, unify the argument with the current user_output stream.

By default, the user_output stream is set to a stream corresponding to the Unix stdout stream.

verbose

If normal allow printing of informational and banner messages, such as the ones that are printed when consulting. If silent disable
printing these messages. It is normal by default except if YAP is booted with the -q or -L flag.

verbose_load

If true allow printing of informational messages when consulting files. If false disable printing these messages. It is normal by default except if YAP is booted with the -L flag.

version

Read-only flag that returns an atom with the current version of YAP.

version_data

Read-only flag that reads a term of the form yap(Major,Minor,Patch,Undefined), where Major is the major version, Minor is the minor version, and Patch is the patch number.

windows

Read-only boolean flag that unifies with true if YAP is running on an Windows machine.

write_strings

Writable flag telling whether the system should write lists of integers that are writable character codes using the list notation. It is on if enables or off if disabled. The default value for this flag is off.

max_workers

Read-only flag telling the maximum number of parallel processes.

max_threads

Read-only flag telling the maximum number of Prolog threads that can be created.

current_prolog_flag(?Flag,-Value) [ISO]

Obtain the value for a YAP Prolog flag. Equivalent to calling yap_flag/2 with the second argument unbound, and unifying the returned second argument with Value.

prolog_flag(?Flag,-OldValue,+NewValue)

Obtain the value for a YAP Prolog flag and then set it to a new value. Equivalent to first calling current_prolog_flag/2 with the second argument OldValue unbound and then calling set_prolog_flag/2 with the third argument NewValue.

set_prolog_flag(+Flag,+Value) [ISO]

Set the value for YAP Prolog flag Flag. Equivalent to calling yap_flag/2 with both arguments bound.
create_prolog_flag(+Flag,+Value,+Options)
Create a new YAP Prolog flag. Options include type(+Type) and access(+Access) with Access one of read_only or read_write and Type one of boolean, integer, float, atom and term (that is, no type).

op(+P,+T,+A) [ISO]
Defines the operator A or the list of operators A with type T (which must be one of xfx, xfy,yfx, xf, yf, fx or fy) and precedence P (see appendix iv for a list of predefined operators).
Note that if there is a preexisting operator with the same name and type, this operator will be discarded. Also, ',' may not be defined as an operator, and it is not allowed to have the same for an infix and a postfix operator.

current_op(P,T,F) [ISO]
Defines the relation: P is a currently defined operator of type T and precedence P.

prompt(-A,+B)
Changes YAP input prompt from A to B.

initialization
Execute the goals defined by initialization/1. Only the first answer is considered.

prolog_initialization(G)
Add a goal to be executed on system initialization. This is compatible with SICStus Prolog’s initialization/1.

version
Write YAP’s boot message.

version(-Message)
Add a message to be written when yap boots or after aborting. It is not possible to remove messages.

prolog_load_context(?Key, ?Value)
Obtain information on what is going on in the compilation process. The following keys are available:
directory
Full name for the directory where YAP is currently consulting the file.
file
Full name for the file currently being consulted. Notice that included files are ignored.
module
Current source module.
source
Full name for the file currently being read in, which may be consulted, reconsulted, or included.
stream
Stream currently being read in.

term_position
Stream position at the stream currently being read in.

source_location(?FileName, ?Line)
SWI-compatible predicate. 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. Please use prolog_load_context/2.

source_file(?File)
SWI-compatible predicate. True if File is a loaded Prolog source file.

source_file(?ModuleNameAndPred,?File)
SWI-compatible predicate. True if the predicate specified by ModuleAndPred was loaded from file File, where File is an absolute path name (see absolute_file_name/2).
7 Library Predicates

Library files reside in the library directory path (set by the LIBDIR variable in the Makefile for YAP). Currently, most files in the library are from the Edinburgh Prolog library.

7.1 Aggregate

This is the SWI-Prolog library based on the Quintus and SICStus 4 library. Notice that forall/2 is a SWI-Prolog built-in and term_variables/3 is a SWI-Prolog with a different definition.

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:

```prolog
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:

```prolog
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.

\[ \text{max(Expr)} \]

Maximum of \( \text{Expr} \) for all solutions.

\[ \text{max(Expr, Witness)} \]

As \( \text{min(Expr, Witness)} \), but producing the maximum result.

\[ \text{set(X)} \]

An ordered set with all solutions for \( \text{X} \).

\[ \text{bag(X)} \]

A list of all solutions for \( \text{X} \).

The predicates are:

**[nondet]aggregate(+Template, :Goal, -Result)**

Aggregate bindings in \( \text{Goal} \) according to \( \text{Template} \). The aggregate/3 version performs bagof/3 on \( \text{Goal} \).

**[nondet]aggregate(+Template, +Discriminator, :Goal, -Result)**

Aggregate bindings in \( \text{Goal} \) according to \( \text{Template} \). The aggregate/3 version performs setof/3 on \( \text{Goal} \).

**[semidet]aggregate_all(+Template, :Goal, -Result)**

Aggregate bindings in \( \text{Goal} \) according to \( \text{Template} \). The aggregate_all/3 version performs findall/3 on \( \text{Goal} \).

**[semidet]aggregate_all(+Template, +Discriminator, :Goal, -Result)**

Aggregate bindings in \( \text{Goal} \) according to \( \text{Template} \). The aggregate_all/3 version performs findall/3 followed by sort/2 on \( \text{Goal} \).

**foreach(:Generator, :Goal)**

True if the conjunction of instances of \( \text{Goal} \) using the bindings from \( \text{Generator} \) is true. Unlike forall/2, which runs a failure-driven loop that proves \( \text{Goal} \) for each solution of \( \text{Generator} \), foreach creates a conjunction. Each member of the conjunction is a copy of \( \text{Goal} \), where the variables it shares with \( \text{Generator} \) are filled with the values from the corresponding solution.

The implementation executes forall/2 if \( \text{Goal} \) does not contain any variables that are not shared with \( \text{Generator} \).

Here is an example:

\[
\text{?- foreach(between(1,4,X), dif(X,Y)), Y = 5.}
\]

\[
\text{Y = 5}
\]

\[
\text{?- foreach(between(1,4,X), dif(X,Y)), Y = 3.}
\]

\[
\text{No}
\]

Notice that \( \text{Goal} \) is copied repeatedly, which may cause problems if attributed variables are involved.

**[det]free_variables(:Generator, +Template, +VarList0, -VarList)**

In order to handle variables properly, we have to find all the universally quantified variables in the \( \text{Generator} \). All variables as yet unbound are universally quantified, unless

1. they occur in the template
2. they are bound by $\text{X/P}$, setof, or bagof

\[
\text{free_variables(Generator, Template, OldList, NewList)}
\]
finds this set, using OldList as an accumulator.

The original author of this code was Richard O’Keefe. Jan Wielemaker made some SWI-Prolog enhancements, sponsored by SecuritEase, http://www.securitease.com. The code is public domain (from DEC10 library).

### 7.2 Apply Macros

This library provides a SWI-compatible set of utilities for applying a predicate to all elements of a list. The library just forwards definitions from the \texttt{maplist} library.

### 7.3 Association Lists

The following association list manipulation predicates are available once included with the \texttt{use_module(library(assoc))} command. The original library used Richard O’Keefe’s implementation, on top of unbalanced binary trees. The current code utilises code from the red-black trees library and emulates the SICStus Prolog interface.

\[
\text{assoc_to_list(}+\text{Assoc,} ?\text{List)}
\]

Given an association list \texttt{Assoc} unify \texttt{List} with a list of the form \texttt{Key-Val}, where the elements \texttt{Key} are in ascending order.

\[
\text{del_assoc(+Key, +Assoc, ?Val, ?NewAssoc)}
\]

Succeeds if \texttt{NewAssoc} is an association list, obtained by removing the element with \texttt{Key} and \texttt{Val} from the list \texttt{Assoc}.

\[
\text{del_max_assoc(+Assoc, ?Key, ?Val, ?NewAssoc)}
\]

Succeeds if \texttt{NewAssoc} is an association list, obtained by removing the largest element of the list, with \texttt{Key} and \texttt{Val} from the list \texttt{Assoc}.

\[
\text{del_min_assoc(+Assoc, ?Key, ?Val, ?NewAssoc)}
\]

Succeeds if \texttt{NewAssoc} is an association list, obtained by removing the smallest element of the list, with \texttt{Key} and \texttt{Val} from the list \texttt{Assoc}.

\[
\text{empty_assoc(+Assoc)}
\]

Succeeds if association list \texttt{Assoc} is empty.

\[
\text{gen_assoc(+Assoc,} ?\text{Key,} ?\text{Value)}
\]

Given the association list \texttt{Assoc}, unify \texttt{Key} and \texttt{Value} with two associated elements. It can be used to enumerate all elements in the association list.

\[
\text{get_assoc(+Key,+Assoc,} ?\text{Value)}
\]

If \texttt{Key} is one of the elements in the association list \texttt{Assoc}, return the associated value.

\[
\text{get_assoc(+Key,+Assoc,} ?\text{Value,+NAssoc,} ?\text{NValue)}
\]

If \texttt{Key} is one of the elements in the association list \texttt{Assoc}, return the associated value \texttt{Value} and a new association list \texttt{NAssoc} where \texttt{Key} is associated with \texttt{NValue}. 
get_prev_assoc(+Key,+Assoc,?Next,?Value)
If Key is one of the elements in the association list Assoc, return the previous
key, Next, and its value, Value.

get_next_assoc(+Key,+Assoc,?Next,?Value)
If Key is one of the elements in the association list Assoc, return the next key,
Next, and its value, Value.
is_assoc(+Assoc)
Succeeds if Assoc is an association list, that is, if it is a red-black tree.

list_to_assoc(+List,?Assoc)
Given a list List such that each element of List is of the form Key-Val, and all
the Keys are unique, Assoc is the corresponding association list.

map_assoc(+Pred,+Assoc)
Succeeds if the unary predicate name Pred(Val) holds for every element in the
association list.

map_assoc(+Pred,+Assoc,?New)
Given the binary predicate name Pred and the association list Assoc, New in
an association list with keys in Assoc, and such that if Key-Val is in Assoc, and
Key-Ans is in New, then Pred(Val,Ans) holds.

max_assoc(+Assoc,-Key,?Value)
Given the association list Assoc, Key in the largest key in the list, and Value
the associated value.

min_assoc(+Assoc,-Key,?Value)
Given the association list Assoc, Key in the smallest key in the list, and Value
the associated value.

ord_list_to_assoc(+List,?Assoc)
Given an ordered list List such that each element of List is of the form Key-Val,
and all the Keys are unique, Assoc is the corresponding association list.

put_assoc(+Key,+Assoc,+Val,+New)
The association list New includes and element of association key with Val, and
all elements of Assoc that did not have key Key.

7.4 AVL Trees
AVL trees are balanced search binary trees. They are named after their inventors, Adelson-
Velskii and Landis, and they were the first dynamically balanced trees to be proposed. The
YAP AVL tree manipulation predicates library uses code originally written by Martin van
Emdem and published in the Logic Programming Newsletter, Autumn 1981. A bug in this
code was fixed by Philip Vasey, in the Logic Programming Newsletter, Summer 1982. The
library currently only includes routines to insert and lookup elements in the tree. Please
try red-black trees if you need deletion.

avl_new(+T)
Create a new tree.
avl_insert(+Key, ?Value, +T0, -TF)
    Add an element with key Key and Value to the AVL tree T0 creating a new
    AVL tree TF. Duplicated elements are allowed.

avl_lookup(+Key, -Value, +T)
    Lookup an element with key Key in the AVL tree T, returning the value Value.

7.5 Heaps

A heap is a labelled binary tree where the key of each node is less than or equal to the keys
of its sons. The point of a heap is that we can keep on adding new elements to the heap
and we can keep on taking out the minimum element. If there are N elements total, the
total time is \(O(N\lg N)\). If you know all the elements in advance, you are better off doing a
merge-sort, but this file is for when you want to do say a best-first search, and have no idea
when you start how many elements there will be, let alone what they are.

The following heap manipulation routines are available once included with the use_
module(library(heaps)) command.

add_to_heap(+Heap, +key, +Datum, -NewHeap)
    Inserts the new Key-Datum pair into the heap. The insertion is not stable,
    that is, if you insert several pairs with the same Key it is not defined which of
    them will come out first, and it is possible for any of them to come out first
    depending on the history of the heap.

empty_heap(?Heap)
    Succeeds if Heap is an empty heap.

get_from_heap(+Heap, -key, -Datum, -Heap)
    Returns the Key-Datum pair in OldHeap with the smallest Key, and also a
    Heap which is the OldHeap with that pair deleted.

heap_size(+Heap, -Size)
    Reports the number of elements currently in the heap.

heap_to_list(+Heap, -List)
    Returns the current set of Key-Datum pairs in the Heap as a List, sorted into
    ascending order of Keys.

list_to_heap(+List, -Heap)
    Takes a list of Key-Datum pairs (such as keysort could be used to sort) and
    forms them into a heap.

min_of_heap(+Heap, -Key, -Datum)
    Returns the Key-Datum pair at the top of the heap (which is of course the pair
    with the smallest Key), but does not remove it from the heap.

min_of_heap(+Heap, -Key1, -Datum1,
           -Key2, -Datum2) Returns the smallest (Key1) and second smallest (Key2) pairs
    in the heap, without deleting them.
### 7.6 List Manipulation

The following list manipulation routines are available once included with the `use_module(library(lists))` command.

- **append(??Prefix, ??Suffix, ??Combined)**
  - True when all three arguments are lists, and the members of `Combined` are the members of `Prefix` followed by the members of `Suffix`. It may be used to form `Combined` from a given `Prefix`, `Suffix` or to take a given `Combined` apart.

- **append(??Lists, ??Combined)**
  - Holds if the lists of `Lists` can be concatenated as a `Combined` list.

- **delete(+List, ?Element, ?Residue)**
  - True when `List` is a list, in which `Element` may or may not occur, and `Residue` is a copy of `List` with all elements identical to `Element` deleted.

- **flatten(+List, ?FlattenedList)**
  - Flatten a list of lists `List` into a single list `FlattenedList`.
    
    ```prolog
    ?- flatten([[1],[2,3],[4,[5,6],7,8]],L).
    L = [1,2,3,4,5,6,7,8] ;
    no
    ```

- **last(+List, ?Last)**
  - True when `List` is a list and `Last` is identical to its last element.

- **list_concat(+Lists, ?List)**
  - True when `Lists` is a list of lists and `List` is the concatenation of `Lists`.

- **member(??Element, ??Set)**
  - True when `Set` is a list, and `Element` occurs in it. It may be used to test for an element or to enumerate all the elements by backtracking.

- **memberchk(+Element, +Set)**
  - As `member/2`, but may only be used to test whether a known `Element` occurs in a known `Set`. In return for this limited use, it is more efficient when it is applicable.

- **nth0(??N, ??List, ?Elem)**
  - True when `Elem` is the `N`th member of `List`, counting the first as element 0. (That is, throw away the first `N` elements and unify `Elem` with the next.) It can only be used to select a particular element given the list and index. For that task it is more efficient than `member/2`.

- **nth1(??N, ??List, ?Elem)**
  - The same as `nth0/3`, except that it counts from 1, that is `nth(1, [H|_], H)`.

- **nth(??N, ??List, ?Elem)**
  - The same as `nth1/3`.

- **nth0(??N, ??List, ?Elem, ?Rest)**
  - Unifies `Elem` with the `N`th element of `List`, counting from 0, and `Rest` with the other elements. It can be used to select the `N`th element of `List` (yielding
Elem and Rest), or to insert Elem before the Nth (counting from 1) element of Rest, when it yields List, e.g. nth0(2, List, c, [a,b,d,e]) unifies List with [a,b,c,d,e]. nth/4 is the same except that it counts from 1. nth0/4 can be used to insert Elem after the Nth element of Rest.

nth1(?N, ?List, ?Elem, ?Rest)
Unifies Elem with the Nth element of List, counting from 1, and Rest with the other elements. It can be used to select the Nth element of List (yielding Elem and Rest), or to insert Elem before the Nth (counting from 1) element of Rest, when it yields List, e.g. nth3(List, c, [a,b,d,e]) unifies List with [a,b,c,d,e]. nth/4 can be used to insert Elem after the Nth element of Rest.

Same as nth1/4.

permutation(+List,?Perm)
True when List and Perm are permutations of each other.

remove_duplicates(+List, ?Pruned)
Removes duplicated elements from List. Beware: if the List has non-ground elements, the result may surprise you.

reverse(+List, ?Reversed)
True when List and Reversed are lists with the same elements but in opposite orders.

same_length(?List1, ?List2)
True when List1 and List2 are both lists and have the same number of elements. No relation between the values of their elements is implied. Modes same_length(-,+), same_length(+,-) generate either list given the other; mode same_length(-,-) generates two lists of the same length, in which case the arguments will be bound to lists of length 0, 1, 2, ...

select(?Element, ?List, ?Residue)
True when Set is a list, Element occurs in List, and Residue is everything in List except Element (things stay in the same order).

selectchk(?Element, ?List, ?Residue)
Semi-deterministic selection from a list. Steadfast: defines as
   selectchk(Elem, List, Residue) :-
   select(Elem, List, Rest0), !,
   Rest = Rest0.

sublist(?Sublist, ?List)
True when both append(_,Sublist,S) and append(S,_,List) hold.

suffix(?Suffix, ?List)
Holds when append(_,Suffix,List) holds.

sum_list(?Numbers, ?Total)
True when Numbers is a list of numbers, and Total is their sum.

sum_list(?Numbers, +SoFar, ?Total)
True when Numbers is a list of numbers, and Total is the sum of their total plus SoFar.
sumlist(?Numbers, ?Total)
  True when Numbers is a list of integers, and Total is their sum. The same as sum_list/2, please do use sum_list/2 instead.

max_list(?Numbers, ?Max)
  True when Numbers is a list of numbers, and Max is the maximum.

min_list(?Numbers, ?Min)
  True when Numbers is a list of numbers, and Min is the minimum.

numlist(+Low, +High, +List)
  If Low and High are integers with Low =< High, unify List to a list [Low, Low+1, ...High]. See also between/3.

intersection(+Set1, +Set2, +Set3)
  Succeeds if Set3 unifies with the intersection of Set1 and Set2. Set1 and Set2 are lists without duplicates. They need not be ordered.

7.7 Line Manipulation Utilities

This package provides a set of useful predicates to manipulate sequences of characters codes, usually first read in as a line. It is available by loading the library library(lineutils).

search_for(+Char,+Line)
  Search for a character Char in the list of codes Line.

search_for(+Char,+Line)
  Search for a character Char in the list of codes Line.

search_for(+Char,+Line,-RestOfLine)
  Search for a character Char in the list of codes Line, RestOfLine has the line to the right.

scan_natural(?Nat,+Line,+RestOfLine)
  Scan the list of codes Line for a natural number Nat, zero or a positive integer, and unify RestOfLine with the remainder of the line.

scan_integer(?Int,+Line,+RestOfLine)
  Scan the list of codes Line for an integer Nat, either a positive, zero, or negative integer, and unify RestOfLine with the remainder of the line.

split(+Line,+Separators,-Split)
  Unify Words with a set of strings obtained from Line by using the character codes in Separators as separators. As an example, consider:

  ?- split("Hello * I am free"," *",S).

  S = ["Hello","I","am","free"] ?

  no

split(+Line,-Split)
  Unify Words with a set of strings obtained from Line by using the blank characters as separators.
fields(+Line,+Separators,-Split)
    Unify Words with a set of strings obtained from Line by using the character codes in Separators as separators for fields. If two separators occur in a row, the field is considered empty. As an example, consider:
    
    ?- fields("Hello I am free"," ",S).
    
    S = ["Hello","","I","","am","","free"] ?

fields(+Line,-Split)
    Unify Words with a set of strings obtained from Line by using the blank characters as field separators.

glue(+Words,+Separator,-Line)
    Unify Line with string obtained by glueing Words with the character code Separator.

copy_line(+StreamInput,+StreamOutput)
    Copy a line from StreamInput to StreamOutput.

copy_line(+StreamInput,+StreamOutput)
    Copy a line from StreamInput to StreamOutput.

process(+StreamInp, +Goal)
    For every line LineIn in stream StreamInp, call call(Goal,LineIn).

filter(+StreamInp, +StreamOut, +Goal)
    For every line LineIn in stream StreamInp, execute call(Goal,LineIn,LineOut) and output LineOut to stream StreamOut.

file_filter(+FileIn, +FileOut, +Goal)
    For every line LineIn in file FileIn, execute call(Goal,LineIn,LineOut), and output LineOut to file FileOut.

file_filter(+FileIn, +FileOut, +Goal,
            +FormatCommand, +Arguments)
    Same as file_filter/3, but before starting the filter execute format/3 on the output stream, using FormatCommand and Arguments.

7.8 Maplist

This library provides a set of utilities for applying a predicate to all elements of a list or to all sub-terms of a term. They allow to easily perform the most common do-loop constructs in Prolog. To avoid performance degradation due to apply/2, each call creates an equivalent Prolog program, without meta-calls, which is executed by the Prolog engine instead. Note that if the equivalent Prolog program already exists, it will be simply used. The library is based on code by Joachim Schimpf and on code from SWI-Prolog.

The following routines are available once included with the use_module(library(apply_macros)) command.

maplist(+Pred, ?ListIn, ?ListOut)
    Creates ListOut by applying the predicate Pred to all elements of ListIn.
maplist(+Pred, ?ListIn)
    Creates ListOut by applying the predicate Pred to all elements of ListIn.

maplist(+Pred, ?L1, ?L2, ?L3)
    L1, L2, and L3 are such that call(Pred, A1, A2, A3) holds for every corresponding element in lists L1, L2, and L3.

maplist(+Pred, ?L1, ?L2, ?L3, ?L4)
    L1, L2, L3, and L4 are such that call(Pred, A1, A2, A3, A4) holds for every corresponding element in lists L1, L2, L3, and L4.

checklist(+Pred, +List)
    Succeeds if the predicate Pred succeeds on all elements of List.

selectlist(+Pred, +ListIn, ?ListOut)
    Creates ListOut of all list elements of ListIn that pass a given test.

convlist(+Pred, +ListIn, ?ListOut)
    A combination of maplist and selectlist: creates ListOut by applying the predicate Pred to all list elements on which Pred succeeds.

sumlist(+Pred, +List, ?AccIn, ?AccOut)
    Calls Pred on all elements of List and collects a result in Accumulator.

mapargs(+Pred, ?TermIn, ?TermOut)
    Creates TermOut by applying the predicate Pred to all arguments of TermIn.

sumargs(+Pred, +Term, ?AccIn, ?AccOut)
    Calls the predicate Pred on all arguments of Term and collects a result in Accumulator.

mapnodes(+Pred, +TermIn, ?TermOut)
    Creates TermOut by applying the predicate Pred to all sub-terms of TermIn (depth-first and left-to-right order).

checknodes(+Pred, +Term)
    Succeeds if the predicate Pred succeeds on all sub-terms of Term (depth-first and left-to-right order).

sumnodes(+Pred, +Term, ?AccIn, ?AccOut)
    Calls the predicate Pred on all sub-terms of Term and collects a result in Accumulator (depth-first and left-to-right order).

include(+Pred, +ListIn, ?ListOut)
    Same as selectlist/3.

exclude(+Goal, +List1, ?List2)
    Filter elements for which Goal fails. True if List2 contains those elements Xi of List1 for which call(Goal, Xi) fails.

partition(+Pred, +List1, ?Included, ?Excluded)
    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, +List1, ?Lesser, ?Equal, ?Greater)
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.

Examples:
%given
plus(X,Y,Z) :- Z is X + Y.
plus_if_pos(X,Y,Z) :- Y > 0, Z is X + Y.
vars(X, Y, [X|Y]) :- var(X), !.
vars(_, Y, Y).
trans(TermIn, TermOut) :-
    (compound(TermIn) ; atom(TermIn)),
    TermIn =.. [p|Args],
    TermOut =..[q|Args],
    !.
trans(X,X).

%success
maplist(plus(1), [1,2,3,4], [2,3,4,5]).
checklist(var, [X,Y,Z]).
selectlist(<(0), [-1,0,1], [1]).
convlist(plus_if_pos(1), [-1,0,1], [2]).
sumlist(plus, [1,2,3,4], 1, 11).
mapargs(number_atom,s(1,2,3), s('1','2','3')).
sumargs(vars, s(1,X,2,Y), [], [Y,X]).
mapnodes(trans, p(a,p(b,a),c), q(a,q(b,a),c)).
checknodes(\==(T), p(X,p(Y,X),Z)).
sumnodes(vars, [c(X), p(X,Y), q(Y)], [], [Y,Y,X,X]).
% another one
maplist(mapargs(number_atom),[c(1),s(1,2,3)],[c('1'),s('1','2','3')]).

7.9 Matrix Library
This package provides a fast implementation of multi-dimensional matrices of integers and float numbers. In contrast to dynamic arrays, these matrices are multi-dimensional and compact. In contrast to static arrays, these arrays are allocated in the stack. Matrices are available by loading the library library(matrix).

Accessing the matlab dynamic libraries can be complicated. In Linux machines, to use this interface, you may have to set the environment variable LD_LIBRARY_PATH. Next, follows an example using bash in a 64-bit Linux PC:

```bash
export LD_LIBRARY_PATH='$MATLAB_HOME'/sys/os/glnxa64:'$MATLAB_HOME'/bin/glnxa64:'$LD_LIBRARY_PATH'
```

where MATLAB_HOME is the directory where matlab is installed at. Please replace ax64 for x86 on a 32-bit PC.

Notice that the functionality in this library is only partial. Please contact the YAP maintainers if you require extra functionality.
matrix_new(+Type,+Dims,-Matrix)
Create a new matrix Matrix of type Type, which may be one of ints or floats, and with a list of dimensions Dims. The matrix will be initialised to zeros.

?- matrix_new(ints,[2,3],Matrix).
Matrix = 0
Notice that currently YAP will always write a matrix as 0.

matrix_new(+Type,+Dims,+List,-Matrix)
Create a new matrix Matrix of type Type, which may be one of ints or floats, with dimensions Dims, and initialised from list List.

matrix_new_set(?Dims,+OldMatrix,+Value,-NewMatrix)
Create a new matrix NewMatrix of type Type, with dimensions Dims. The elements of NewMatrix are set to Value.

matrix_dims(+Matrix,-Dims)
Unify Dims with a list of dimensions for Matrix.

matrix_ndims(+Matrix,-Dims)
Unify NDims with the number of dimensions for Matrix.

matrix_size(+Matrix,-NElems)
Unify NElems with the number of elements for Matrix.

matrix_type(+Matrix,-Type)
Unify NElems with the type of the elements in Matrix.

matrix_to_list(+Matrix,-Elems)
Unify Elems with the list including all the elements in Matrix.

matrix_get(+Matrix,+Position,-Elem)
Unify Elem with the element of Matrix at position Position.

matrix_set(+Matrix,+Position,+Elem)
Set the element of Matrix at position Position to Elem.

matrix_set_all(+Matrix,+Elem)
Set all element of Matrix to Elem.

matrix_add(+Matrix,+Position,+Operand)
Add Operand to the element of Matrix at position Position.

matrix_inc(+Matrix,+Position)
Increment the element of Matrix at position Position.

matrix_inc(+Matrix,+Position,-Element)
Increment the element of Matrix at position Position and unify with Element.

matrix_dec(+Matrix,+Position)
Decrement the element of Matrix at position Position.

matrix_dec(+Matrix,+Position,-Element)
Decrement the element of Matrix at position Position and unify with Element.
matrix_arg_to_offset(+Matrix,+Position,-Offset)
Given matrix Matrix return what is the numerical Offset of the element at Position.

matrix_offset_to_arg(+Matrix,-Offset,+Position)
Given a position Position for matrix Matrix return the corresponding numerical Offset from the beginning of the matrix.

matrix_max(+Matrix,+Max)
Unify Max with the maximum in matrix Matrix.

matrix_maxarg(+Matrix,+Maxarg)
Unify Max with the position of the maximum in matrix Matrix.

matrix_min(+Matrix,+Min)
Unify Min with the minimum in matrix Matrix.

matrix_minarg(+Matrix,+Minarg)
Unify Min with the position of the minimum in matrix Matrix.

matrix_sum(+Matrix,+Sum)
Unify Sum with the sum of all elements in matrix Matrix.

matrix_agg_lines(+Matrix,+Aggregate)
If Matrix is a n-dimensional matrix, unify Aggregate with the n-1 dimensional matrix where each element is obtained by adding all Matrix elements with same last n-1 index.

matrix_agg_cols(+Matrix,+Aggregate)
If Matrix is a n-dimensional matrix, unify Aggregate with the one dimensional matrix where each element is obtained by adding all Matrix elements with same first index.

matrix_op(+Matrix1,+Matrix2,+Op,-Result)
Result is the result of applying Op to matrix Matrix1 and Matrix2. Currently, only addition (+) is supported.

matrix_op_to_all(+Matrix1,+Op,+Operand,-Result)
Result is the result of applying Op to all elements of Matrix1, with Operand as the second argument. Currently, only addition (+), multiplication (*), and division (/) are supported.

matrix_op_to_lines(+Matrix1,+Lines,+Op,-Result)
Result is the result of applying Op to all elements of Matrix1, with the corresponding element in Lines as the second argument. Currently, only division (/) is supported.

matrix_op_to_cols(+Matrix1,+Cols,+Op,-Result)
Result is the result of applying Op to all elements of Matrix1, with the corresponding element in Cols as the second argument. Currently, only addition (+) is supported. Notice that Cols will have n-1 dimensions.

matrix_shuffle(+Matrix,+NewOrder,-Shuffle)
Shuffle the dimensions of matrix Matrix according to NewOrder. The list NewOrder must have all the dimensions of Matrix, starting from 0.
\texttt{matrix\_transpose(+Matrix,-Transpose)}

Transpose matrix \texttt{Matrix} to \texttt{Transpose}. Equivalent to:

\begin{verbatim}
\texttt{matrix\_transpose(Matrix,Transpose) :-}
\texttt{matrix\_shuffle(Matrix,[1,0],Transpose).}
\end{verbatim}

\texttt{matrix\_expand(+Matrix,+NewDimensions,-New)}

Expand \texttt{Matrix} to occupy new dimensions. The elements in \texttt{NewDimensions} are either 0, for an existing dimension, or a positive integer with the size of the new dimension.

\texttt{matrix\_select(+Matrix,+Dimension,+Index,-New)}

Select from \texttt{Matrix} the elements who have \texttt{Index} at \texttt{Dimension}.

\texttt{matrix\_row(+Matrix,+Column,-NewMatrix)}

Select from \texttt{Matrix} the row matching \texttt{Column} as new matrix \texttt{NewMatrix}. \texttt{Column} must have one less dimension than the original matrix. \texttt{Dimension}.

### 7.10 MATLAB Package Interface

The MathWorks MATLAB is a widely used package for array processing. YAP now includes a straightforward interface to MATLAB. To actually use it, you need to install YAP calling \texttt{configure} with the \texttt{--with-matlab=DIR} option, and you need to call \texttt{use\_module(library(lists))} command.

\texttt{start\_matlab(+Options)}

Start a matlab session. The argument \texttt{Options} may either be the empty string/atom or the command to call \texttt{matlab}. The command may fail.

\texttt{close\_matlab}

Stop the current matlab session.

\texttt{matlab\_on}

Holds if a matlab session is on.

\texttt{matlab\_eval\_string(+Command)}

Holds if matlab evaluated successfully the command \texttt{Command}.

\texttt{matlab\_eval\_string(+Command, -Answer)}

MATLAB will evaluate the command \texttt{Command} and unify \texttt{Answer} with a string reporting the result.

\texttt{matlab\_cells(+Size, ?Array)}

MATLAB will create an empty vector of cells of size \texttt{Size}, and if \texttt{Array} is bound to an atom, store the array in the matlab variable with name \texttt{Array}. Corresponds to the MATLAB command \texttt{cells}.

\texttt{matlab\_cells(+SizeX, +SizeY, ?Array)}

MATLAB will create an empty array of cells of size \texttt{SizeX} and \texttt{SizeY}, and if \texttt{Array} is bound to an atom, store the array in the matlab variable with name \texttt{Array}. Corresponds to the MATLAB command \texttt{cells}.

\texttt{matlab\_initialized\_cells(+SizeX, +SizeY, +List, ?Array)}

MATLAB will create an array of cells of size \texttt{SizeX} and \texttt{SizeY}, initialized from the list \texttt{List}, and if \texttt{Array} is bound to an atom, store the array in the matlab variable with name \texttt{Array}.
matlab_matrix(+SizeX, +SizeY, +List, ?Array)
MATLAB will create an array of floats of size SizeX and SizeY, initialized from the list List, and if Array is bound to an atom, store the array in the matlab variable with name Array.

matlab_set(+MatVar, +X, +Y, +Value)
Call MATLAB to set element MatVar(X, Y) to Value. Notice that this command uses the MATLAB array access convention.

matlab_get_variable(+MatVar, -List)
Unify MATLAB variable MatVar with the List List.

matlab_item(+MatVar, +X, ?Val)
Read or set MATLAB MatVar(X) from/to Val. Use C notation for matrix access (i.e., starting from 0).

matlab_item(+MatVar, +X, +Y, ?Val)
Read or set MATLAB MatVar(X,Y) from/to Val. Use C notation for matrix access (i.e., starting from 0).

matlab_item1(+MatVar, +X, ?Val)
Read or set MATLAB MatVar(X) from/to Val. Use MATLAB notation for matrix access (i.e., starting from 1).

matlab_item1(+MatVar, +X, +Y, ?Val)
Read or set MATLAB MatVar(X,Y) from/to Val. Use MATLAB notation for matrix access (i.e., starting from 1).

matlab_sequence(+Min, +Max, ?Array)
MATLAB will create a sequence going from Min to Max, and if Array is bound to an atom, store the sequence in the matlab variable with name Array.

matlab_vector(+Size, +List, ?Array)
MATLAB will create a vector of floats of size Size, initialized from the list List, and if Array is bound to an atom, store the array in the matlab variable with name Array.

matlab_zeros(+Size, ?Array)
MATLAB will create a vector of zeros of size Size, and if Array is bound to an atom, store the array in the matlab variable with name Array. Corresponds to the MATLAB command zeros.

matlab_zeros(+SizeX, +SizeY, ?Array)
MATLAB will create an array of zeros of size SizeX and SizeY, and if Array is bound to an atom, store the array in the matlab variable with name Array. Corresponds to the MATLAB command zeros.

matlab_zeros(+SizeX, +SizeY, +SizeZ, ?Array)
MATLAB will create an array of zeros of size SizeX, SizeY, and SizeZ. If Array is bound to an atom, store the array in the matlab variable with name Array. Corresponds to the MATLAB command zeros.
matlab_zeros(+SizeX, +SizeY, +SizeZ, ?Array)
MATLAB will create an array of zeros of size SizeX, SizeY, and SizeZ. If Array
is bound to an atom, store the array in the matlab variable with name Array.
Corresponds to the MATLAB command zeros.

7.11 Non-Backtrackable Data Structures
The following routines implement well-known data-structures using global non-
backtrackable variables (implemented on the Prolog stack). The data-structures currently
supported are Queues, Heaps, and Beam for Beam search. They are allowed through
library(nb).

nb_queue(-Queue)
   Create a Queue.

nb_queue_close(+Queue, -Head, ?Tail)
   Unify the queue Queue with a difference list Head-Tail. The queue will now be
   empty and no further elements can be added.

nb_queue_enqueue(+Queue, +Element)
   Add Element to the front of the queue Queue.

nb_queue_dequeue(+Queue, -Element)
   Remove Element from the front of the queue Queue. Fail if the queue is empty.

nb_queue_peek(+Queue, -Element)
   Element is the front of the queue Queue. Fail if the queue is empty.

nb_queue_size(+Queue, -Size)
   Unify Size with the number of elements in the queue Queue.

nb_queue_empty(+Queue)
   Succeeds if Queue is empty.

nb_heap(+DefaultSize, -Heap)
   Create a Heap with default size DefaultSize. Note that size will expand as
   needed.

nb_heap_close(+Heap)
   Close the heap Heap: no further elements can be added.

nb_heap_add(+Heap, +Key, +Value)
   Add Key-Value to the heap Heap. The key is sorted on Key only.

nb_heap_del(+Heap, -Key, -Value)
   Remove element Key-Value with smallest Value in heap Heap. Fail if the heap
   is empty.

nb_heap.peek(+Heap, -Key, -Value)
   Key-Value is the element with smallest Key in the heap Heap. Fail if the heap
   is empty.

nb_heap_size(+Heap, -Size)
   Unify Size with the number of elements in the heap Heap.
Chapter 7: Library Predicates

nb_heap_empty(+Heap)
Succeeds if Heap is empty.

nb_beam(+DefaultSize,-Beam)
Create a Beam with default size DefaultSize. Note that size is fixed throughout.

nb_beam_close(+Beam)
Close the beam Beam: no further elements can be added.

nb_beam_add(+Beam, +Key, +Value)
Add Key-Value to the beam Beam. The key is sorted on Key only.

nb_beam_del(+Beam, -Key, -Value)
Remove element Key-Value with smallest Value in beam Beam. Fail if the beam is empty.

nb_beam_peek(+Beam, -Key, -Value))
Key-Value is the element with smallest Key in the beam Beam. Fail if the beam is empty.

nb_beam_size(+Beam, -Size)
Unify Size with the number of elements in the beam Beam.

nb_beam_empty(+Beam)
Succeeds if Beam is empty.

7.12 Ordered Sets

The following ordered set manipulation routines are available once included with the use_module(library(ordsets)) command. An ordered set is represented by a list having unique and ordered elements. Output arguments are guaranteed to be ordered sets, if the relevant inputs are. This is a slightly patched version of Richard O'Keefe's original library.

list_to_ord_set(+List, ?Set)
Holds when Set is the ordered representation of the set represented by the unordered representation List.

merge(+List1, +List2, -Merged)
Holds when Merged is the stable merge of the two given lists.
Notice that merge/3 will not remove duplicates, so merging ordered sets will not necessarily result in an ordered set. Use ord_union/3 instead.

ord_add_element(+Set1, +Element, ?Set2)
Inserting Element in Set1 returns Set2. It should give exactly the same result as merge(Set1, [Element], Set2), but a bit faster, and certainly more clearly. The same as ord_insert/3.

ord_del_element(+Set1, +Element, ?Set2)
Removing Element from Set1 returns Set2.

ord_disjoint(+Set1, +Set2)
Holds when the two ordered sets have no element in common.

ord_member(+Element, +Set)
Holds when Element is a member of Set.
ord_insert(+Set1, +Element, ?Set2)
Inserting Element in Set1 returns Set2. It should give exactly the same result as
merge(Set1, [Element], Set2), but a bit faster, and certainly more clearly.
The same as ord_add_element/3.

ord_intersect(+Set1, +Set2)
Holds when the two ordered sets have at least one element in common.

ord_intersection(+Set1, +Set2, ?Intersection)
Holds when Intersection is the ordered representation of Set1 and Set2.

ord_intersection(+Set1, +Set2, ?Intersection, ?Diff)
Holds when Intersection is the ordered representation of Set1 and Set2. Diff is
the difference between Set2 and Set1.

ord_seteq(+Set1, +Set2)
Holds when the two arguments represent the same set.

ord_setproduct(+Set1, +Set2, -Set)
If Set1 and Set2 are ordered sets, Product will be an ordered set of x1-x2 pairs.

ord_subset(+Set1, +Set2)
Holds when every element of the ordered set Set1 appears in the ordered set
Set2.

ord_subtract(+Set1, +Set2, ?Difference)
Holds when Difference contains all and only the elements of Set1 which are not
also in Set2.

ord_symdiff(+Set1, +Set2, ?Difference)
Holds when Difference is the symmetric difference of Set1 and Set2.

ord_union(+Sets, ?Union)
Holds when Union is the union of the lists Sets.

ord_union(+Set1, +Set2, ?Union)
Holds when Union is the union of Set1 and Set2.

ord_union(+Set1, +Set2, ?Union, ?Diff)
Holds when Union is the union of Set1 and Set2 and Diff is the difference.

7.13 Pseudo Random Number Integer Generator

The following routines produce random non-negative integers in the range 0 .. 2^w -1, where w is the word size available for integers, e.g. 32 for Intel machines and 64 for Alpha machines. Note that the numbers generated by this random number generator are repeatable. This generator was originally written by Allen Van Gelder and is based on Knuth Vol 2.

rannum(-I)
Produces a random non-negative integer I whose low bits are not all that ran-
dom, so it should be scaled to a smaller range in general. The integer I is in
the range 0 .. 2^(w-1) - 1. You can use:
rannum(X) :- yap_flag(max_integer,MI), rannum(R), X is R/MI.

to obtain a floating point number uniformly distributed between 0 and 1.

ranstart Initialize the random number generator using a built-in seed. The ranstart/0 built-in is always called by the system when loading the package.

ranstart(+Seed)
Initialize the random number generator with user-defined Seed. The same Seed always produces the same sequence of numbers.

ranunif(+Range,-I)
ranunif/2 produces a uniformly distributed non-negative random integer I over a caller-specified range R. If range is R, the result is in 0 .. R-1.

7.14 Queues
The following queue manipulation routines are available once included with the use_module(library(queues)) command. Queues are implemented with difference lists.

make_queue(+Queue)
Creates a new empty queue. It should only be used to create a new queue.

join_queue(+Element, +OldQueue, -NewQueue)
Adds the new element at the end of the queue.

list_join_queue(+List, +OldQueue, -NewQueue)
Adds the new elements at the end of the queue.

jump_queue(+Element, +OldQueue, -NewQueue)
Adds the new element at the front of the list.

list_jump_queue(+List, +OldQueue, +NewQueue)
Adds all the elements of List at the front of the queue.

head_queue(+Queue, ?Head)
Unifies Head with the first element of the queue.

serve_queue(+OldQueue, +Head, -NewQueue)
Removes the first element of the queue for service.

empty_queue(+Queue)
Tests whether the queue is empty.

length_queue(+Queue, -Length)
Counts the number of elements currently in the queue.

list_to_queue(+List, -Queue)
Creates a new queue with the same elements as List.

queue_to_list(+Queue, -List)
Creates a new list with the same elements as Queue.
7.15 Random Number Generator

The following random number operations are included with the use_module(library(random)) command. Since YAP-4.3.19 YAP uses the O'Keefe public-domain algorithm, based on the "Applied Statistics" algorithm AS183.

getrand(-Key)
  Unify Key with a term of the form rand(X,Y,Z) describing the current state of the random number generator.

random(-Number)
  Unify Number with a floating-point number in the range [0...1).

random(+LOW, +HIGH, -NUMBER)
  Unify Number with a number in the range [LOW...HIGH). If both LOW and HIGH are integers then NUMBER will also be an integer, otherwise NUMBER will be a floating-point number.

randseq(+LENGTH, +MAX, -Numbers)
  Unify Numbers with a list of LENGTH unique random integers in the range [1...MAX).

randset(+LENGTH, +MAX, -Numbers)
  Unify Numbers with an ordered list of LENGTH unique random integers in the range [1...MAX).

setrand(+Key)
  Use a term of the form rand(X,Y,Z) to set a new state for the random number generator. The integer X must be in the range [1...30269], the integer Y must be in the range [1...30307], and the integer Z must be in the range [1...30323).

7.16 Read Utilities

The readutil library contains primitives to read lines, files, multiple terms, etc.

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 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 upto 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. Currently ignores Options.

read_file_to_terms(+Spec, -Terms, +Options)
Read a file to a list of Prolog terms (see read/1).

7.17 Red-Black Trees

Red-Black trees are balanced search binary trees. They are named because nodes can be classified as either red or black. The code we include is based on "Introduction to Algorithms", second edition, by Cormen, Leiserson, Rivest and Stein. The library includes routines to insert, lookup and delete elements in the tree.

rb_new(?)
Create a new tree.

rb_empty(?)
Succeeds if tree T is empty.

is_rbtree(+T)
Check whether T is a valid red-black tree.

rb_insert(+T0,+Key,?Value,+TF)
Add an element with key Key and Value to the tree T0 creating a new red-black tree TF. Duplicated elements are not allowed.
Add a new element with key Key and Value to the tree T0 creating a new red-black tree TF. Fails is an element with Key exists in the tree.

rb_lookup(+Key,-Value,+T)
Backtrack through all elements with key Key in the red-black tree T, returning for each the value Value.

rb_lookupall(+Key,-Value,+T)
Lookup all elements with key Key in the red-black tree T, returning the value Value.

rb_delete(+T,+Key,-TN)
Delete element with key Key from the tree T, returning a new tree TN.
rb_delete(+T,+Key,-Val,-TN)
   Delete element with key Key from the tree T, returning the value Val associated with the key and a new tree TN.

rb_del_min(+T,-Key,-Val,-TN)
   Delete the least element from the tree T, returning the key Key, the value Val associated with the key and a new tree TN.

rb_del_max(+T,-Key,-Val,-TN)
   Delete the largest element from the tree T, returning the key Key, the value Val associated with the key and a new tree TN.

rb_update(+T,+Key,+NewVal,-TN)
   Tree TN is tree T, but with value for Key associated with NewVal. Fails if it cannot find Key in T.

rb_apply(+T,+Key,+G,-TN)
   If the value associated with key Key is Val0 in T, and if call(G,Val0,ValF) holds, then TN differs from T only in that Key is associated with value ValF in tree TN. Fails if it cannot find Key in T, or if call(G,Val0,ValF) is not satisfiable.

rb_visit(+T,-Pairs)
   Pairs is an infix visit of tree T, where each element of Pairs is of the form K-Val.

rb_size(+T,-Size)
   Size is the number of elements in T.

rb_keys(+T,+Keys)
   Keys is an infix visit with all keys in tree T. Keys will be sorted, but may be duplicate.

rb_map(+T,+G,-TN)
   For all nodes Key in the tree T, if the value associated with key Key is Val0 in tree T, and if call(G,Val0,ValF) holds, then the value associated with Key in TN is ValF. Fails if or if call(G,Val0,ValF) is not satisfiable for all Var0.

rb_partial_map(+T,+Keys,+G,-TN)
   For all nodes Key in Keys, if the value associated with key Key is Val0 in tree T, and if call(G,Val0,ValF) holds, then the value associated with Key in TN is ValF. Fails if or if call(G,Val0,ValF) is not satisfiable for all Var0. Assumes keys are not repeated.

rb_clone(+T,+NT,+Nodes)
   “Clone” the red-back tree into a new tree with the same keys as the original but with all values set to unbound values. Nodes is a list containing all new nodes as pairs K-V.

rb_min(+T,-Key,-Value)
   Key is the minimum key in T, and is associated with Val.

rb_max(+T,-Key,-Value)
   Key is the maximal key in T, and is associated with Val.
Chapter 7: Library Predicates

rb_next(+T, +Key, -Next, -Value)
Next is the next element after Key in T, and is associated with Val.

rb_previous(+T, +Key, -Previous, -Value)
Previous is the previous element after Key in T, and is associated with Val.

list_to_rbtree(+L, -T)
T is the red-black tree corresponding to the mapping in list L.

ord_list_to_rbtree(+L, -T)
T is the red-black tree corresponding to the mapping in ordered list L.

7.18 Regular Expressions
This library includes routines to determine whether a regular expression matches part or all of a string. The routines can also return which parts of the string matched the expression or subexpressions of it. This library relies on Henry Spencer’s C-package and is only available in operating systems that support dynamic loading. The C-code has been obtained from the sources of FreeBSD-4.0 and is protected by copyright from Henry Spencer and from the Regents of the University of California (see the file library/regex/COPYRIGHT for further details).

Much of the description of regular expressions below is copied verbatim from Henry Spencer’s manual page.

A regular expression is zero or more branches, separated by “|”. It matches anything that matches one of the branches.

A branch is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by “*”, “+”, or “?”. An atom followed by “*” matches a sequence of 0 or more matches of the atom. An atom followed by “+” matches a sequence of 1 or more matches of the atom. An atom followed by “?” matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a range (see below), “.” (matching any single character), “^” (matching the null string at the beginning of the input string), “$” (matching the null string at the end of the input string), a “\” followed by a single character (matching that character), or a single character with no other significance (matching that character).

A range is a sequence of characters enclosed in “[]”. It normally matches any single character from the sequence. If the sequence begins with “\[”, it matches any single character not from the rest of the sequence. If two characters in the sequence are separated by “\,”, this is shorthand for the full list of ASCII characters between them (e.g. “[0-9]” matches any decimal digit). To include a literal “[” in the sequence, make it the first character (following a possible “\”) and include the single character “\]”, make it the first or last character.

regexp(+RegExp, +String, +Opts)
Match regular expression RegExp to input string String according to options Opts. The options may be:

- nocase: Causes upper-case characters in String to be treated as lower case during the matching process.
regexp(+RegExp,+String,+Opts,?SubMatchVars)

Match regular expression RegExp to input string String according to options Opts. The variable SubMatchVars should be originally unbound or a list of unbound variables all will contain a sequence of matches, that is, the head of SubMatchVars will contain the characters in String that matched the leftmost parenthesized subexpression within RegExp, the next head of list will contain the characters that matched the next parenthesized subexpression to the right in RegExp, and so on.

The options may be:

- **nocase**: Causes upper-case characters in String to be treated as lower case during the matching process.
- **indices**: Changes what is stored in SubMatchVars. Instead of storing the matching characters from String, each variable will contain a term of the form \text{IO-IF} giving the indices in String of the first and last characters in the matching range of characters.

In general there may be more than one way to match a regular expression to an input string. For example, consider the command

\[
\text{regexp}("(a*)b*", "aabaaabb", [], [X, Y])
\]

Considering only the rules given so far, X and Y could end up with the values "aab" and "a", "aab" and "aa", "ab" and "a", or any of several other combinations. To resolve this potential ambiguity regexp chooses among alternatives using the rule “first then longest”. In other words, it considers the possible matches in order working from left to right across the input string and the pattern, and it attempts to match longer pieces of the input string before shorter ones. More specifically, the following rules apply in decreasing order of priority:

1. If a regular expression could match two different parts of an input string then it will match the one that begins earliest.
2. If a regular expression contains "|" operators then the leftmost matching sub-expression is chosen.
3. In *, +, and ? constructs, longer matches are chosen in preference to shorter ones.
4. In sequences of expression components the components are considered from left to right.

In the example from above, "(a*)b*" matches "aab": the "(a*)" portion of the pattern is matched first and it consumes the leading "aa"; then the "b*" portion of the pattern consumes the next "b". Or, consider the following example:

\[
\text{regexp} \left( \text{(ab|a)(b*)c"}, \text{"abc"}, [], [X, Y, Z] \right)
\]

After this command X will be "abc", Y will be "ab", and Z will be an empty string. Rule 4 specifies that "(ab|a)" gets first shot at the input string and Rule 2 specifies that the "ab" sub-expression is checked before the "a" sub-expression. Thus the "b" has already been claimed before the "(b*)" component is checked and (b*) must match an empty string.
7.19 SWI-Prolog’s shlib library

This section discusses the functionality of the (autoload) library(shlib), providing an interface to manage shared libraries.

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:

```c
#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;
    }

    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.

```
[det]load_foreign_library(:FileSpec)
[det]load_foreign_library(:FileSpec, +Entry:atom)
```

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)), ...
```

`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/Yap`. 

...
See also use_foreign_library/1,2 are intended for use in directives.

[det]use_foreign_library(+FileSpec)

[det]use_foreign_library(+FileSpec, +Entry:atom)

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:

```prolog
:- 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.

[det]unload_foreign_library(+FileSpec)

[det]unload_foreign_library(+FileSpec, +Exit:atom)

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.

### 7.20 Splay Trees

Splay trees are explained in the paper "Self-adjusting Binary Search Trees", by D.D. Sleator and R.E. Tarjan, JACM, vol. 32, No.3, July 1985, p. 668. They are designed to support fast insertions, deletions and removals in binary search trees without the complexity of traditional balanced trees. The key idea is to allow the tree to become unbalanced. To make up for this, whenever we find a node, we move it up to the top. We use code by Vijay Saraswat originally posted to the Prolog mailing-list.

splay_access(-Return,+Key,?Val,+Tree,-NewTree)

If item Key is in tree Tree, return its Val and unify Return with true. Otherwise unify Return with null. The variable NewTree unifies with the new tree.

splay_delete(+Key,?Val,+Tree,-NewTree)

Delete item Key from tree Tree, assuming that it is present already. The variable Val unifies with a value for key Key, and the variable NewTree unifies with the new tree. The predicate will fail if Key is not present.

splay_init(-NewTree)

Initialize a new splay tree.

splay_insert(+Key,?Val,+Tree,-NewTree)

Insert item Key in tree Tree, assuming that it is not there already. The variable Val unifies with a value for key Key, and the variable NewTree unifies with the new tree. In our implementation, Key is not inserted if it is already there: rather it is unified with the item already in the tree.
splay_join(+LeftTree,+RightTree,-NewTree)
    Combine trees LeftTree and RightTree into a single tree NewTree containing all items from both trees. This operation assumes that all items in LeftTree are less than all those in RightTree and destroys both LeftTree and RightTree.

splay_split(+Key,?Val,+Tree,-LeftTree,-RightTree)
    Construct and return two trees LeftTree and RightTree, where LeftTree contains all items in Tree less than Key, and RightTree contains all items in Tree greater than Key. This operation destroys Tree.

7.21 Reading From and Writing To Strings

From Version 4.3.2 onwards YAP implements SICStus Prolog compatible String I/O. The library allows users to read from and write to a memory buffer as if it was a file. The memory buffer is built from or converted to a string of character codes by the routines in library. Therefore, if one wants to read from a string the string must be fully instantiated before the library built-in opens the string for reading. These commands are available through the use_module(library(charsio)) command.

format_to_chars(+Form, +Args, -Result)
    Execute the built-in procedure format/2 with form Form and arguments Args outputting the result to the string of character codes Result.

format_to_chars(+Form, +Args, -Result, -Result0)
    Execute the built-in procedure format/2 with form Form and arguments Args outputting the result to the difference list of character codes Result-Result0.

write_to_chars(+Term, -Result)
    Execute the built-in procedure write/1 with argument Term outputting the result to the string of character codes Result.

write_to_chars(+Term, -Result0, -Result)
    Execute the built-in procedure write/1 with argument Term outputting the result to the difference list of character codes Result-Result0.

atom_to_chars(+Atom, -Result)
    Convert the atom Atom to the string of character codes Result.

atom_to_chars(+Atom, -Result0, -Result)
    Convert the atom Atom to the difference list of character codes Result-Result0.

number_to_chars(+Number, -Result)
    Convert the number Number to the string of character codes Result.

number_to_chars(+Number, -Result0, -Result)
    Convert the atom Number to the difference list of character codes Result-Result0.

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.
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).

read_from_chars(+Chars, -Term)
Parse the list of character codes Chars and return the result in the term Term. The character codes to be read must terminate with a dot character such that either (i) the dot character is followed by blank characters; or (ii) the dot character is the last character in the string.

open_chars_stream(+Chars, -Stream)
Open the list of character codes Chars as a stream Stream.

with_output_to_chars(?Goal, -Chars)
Execute goal Goal such that its standard output will be sent to a memory buffer. After successful execution the contents of the memory buffer will be converted to the list of character codes Chars.

with_output_to_chars(?Goal, ?Chars0, -Chars)
Execute goal Goal such that its standard output will be sent to a memory buffer. After successful execution the contents of the memory buffer will be converted to the difference list of character codes Chars-Chars0.

with_output_to_chars(?Goal, -Stream, ?Chars0, -Chars)
Execute goal Goal such that its standard output will be sent to a memory buffer. After successful execution the contents of the memory buffer will be converted to the difference list of character codes Chars-Chars0 and Stream receives the stream corresponding to the memory buffer.

The implementation of the character IO operations relies on three YAP built-ins:

charsio:open_mem_read_stream(+String, -Stream)
Store a string in a memory buffer and output a stream that reads from this memory buffer.

charsio:open_mem_write_stream(-Stream)
Create a new memory buffer and output a stream that writes to it.

charsio:peek_mem_write_stream(-Stream, L0, L)
Convert the memory buffer associated with stream Stream to the difference list of character codes L-L0.

These built-ins are initialized to belong to the module charsio in init.yap. Novel procedures for manipulating strings by explicitly importing these built-ins.

YAP does not currently support opening a charsio stream in append mode, or seeking in such a stream.

7.22 Calling The Operating System from YAP

YAP now provides a library of system utilities compatible with the SICStus Prolog system library. This library extends and to some point replaces the functionality of Operating
System access routines. The library includes Unix/Linux and Win32 C code. They are available through the `use_module(library(system))` command.

```erlang
datetime(datime(-Year, -Month, -DayOfTheMonth, -Hour, -Minute, -Second)) The `datetime/1` procedure returns the current date and time, with information on `Year`, `Month`, `DayOfTheMonth`, `Hour`, `Minute`, and `Second`. The `Hour` is returned on local time. This function uses the WIN32 GetLocalTime function or the Unix `localtime` function.

?- datime(X).
X = datime(2001,5,28,15,29,46) ?
```

```erlang
mktime(datime(+Year, +Month, +DayOfTheMonth, +Hour, +Minute, +Second), -Seconds) The `mktime/1` procedure returns the number of `Seconds` elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). The user provides information on `Year`, `Month`, `DayOfTheMonth`, `Hour`, `Minute`, and `Second`. The `Hour` is given on local time. This function uses the WIN32 GetLocalTime function or the Unix `mktime` function.

X = 991081786 ? ;
```

```erlang
delete_file(+File)
The `delete_file/1` procedure removes file `File`. If `File` is a directory, remove the directory and all its subdirectories.

?- delete_file(x).
```

```erlang
delete_file(+File,+Opts)
The `delete_file/2` procedure removes file `File` according to options `Opts`. These options are `directory` if one should remove directories, `recursive` if one should remove directories recursively, and `ignore` if errors are not to be reported.

This example is equivalent to using the `delete_file/1` predicate:

?- delete_file(x, [recursive]).
```

```erlang
directory_files(+Dir,+List)
Given a directory `Dir`, `directory_files/2` procedures a listing of all files and directories in the directory:

?- directory_files('..',L), writeq(L).
["Makefile."~1~",'sys.so','Makefile','sys.o',x,..,'."
```

```erlang
file_exists(+File)
The atom `File` corresponds to an existing file.
```

```erlang
file_exists(+File,+Permissions)
The atom `File` corresponds to an existing file with permissions compatible with `Permissions`. YAP currently only accepts for permissions to be described as a number. The actual meaning of this number is Operating System dependent.
**file_property(+File,?Property)**

The atom File corresponds to an existing file, and Property will be unified with a property of this file. The properties are of the form type(Type), which gives whether the file is a regular file, a directory, a fifo file, or of unknown type; size(Size), with gives the size for a file, and mod_time(Time), which gives the last time a file was modified according to some Operating System dependent timestamp; mode(mode), gives the permission flags for the file, and linkto(FileNaName), gives the file pointed to by a symbolic link. Properties can be obtained through backtracking:

```prolog
?- file_property('Makefile', P).
```

```
P = type(regular) ;
P = size(2375) ;
P = mod_time(990826911) ;
```

**make_directory(+Dir)**

Create a directory Dir. The name of the directory must be an atom.

**rename_file(+OldFile,+NewFile)**

Create file OldFile to NewFile. This predicate uses the C built-in function rename.

**environ(?EnvVar,+EnvValue)**

Unify environment variable EnvVar with its value EnvValue, if there is one. This predicate is backtrackable in Unix systems, but not currently in Win32 configurations.

```prolog
?- environ('HOME', X).
```

```
X = 'C:\cygwin\home\administrator' ?
```

**host_id(-Id)**

Unify Id with an identifier of the current host. YAP uses the hostid function when available,

**host_name(-Name)**

Unify Name with a name for the current host. YAP uses the hostname function in Unix systems when available, and the GetComputerName function in WIN32 systems.

**kill(Id,+SIGNAL)**

Send signal SIGNAL to process Id. In Unix this predicate is a direct interface to kill so one can send signals to groups of processes. In WIN32 the predicate is an interface to TerminateProcess, so it kills Id independently of SIGNAL.

**mktemp(Spec,-File)**

Direct interface to mktemp: given a Spec, that is a file name with six X to it, create a file name File. Use tmpnam/1 instead.
Chapter 7: Library Predicates

135

pid(-Id)

Unify Id with the process identifier for the current process. An interface to the getpid function.

tmpnam(-File)

Interface with tmpnam: obtain a new, unique file name File.

tmp_file(-File)

Create a name for a temporary file. Base is an user provided 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.

exec(+Command, [+InputStream, +OutputStream, +ErrorStream], -PID)

Execute command Command with its streams connected to InputStream, OutputStream, and ErrorStream. The process that executes the command is returned as PID. The command is executed by the default shell bin/sh -c in Unix.

The following example demonstrates the use of exec/3 to send a command and process its output:

exec(ls,[std,pipe(S),null],P),repeat, get0(S,C), (C = -1, close(S) ! ; put(C)).

The streams may be one of standard stream, std, null stream, null, or pipe(S), where S is a pipe stream. Note that it is up to the user to close the pipe.

working_directory(-CurDir, ?NextDir)

Fetch the current directory at CurDir. If NextDir is bound to an atom, make its value the current working directory.

popen(+Command, +TYPE, -Stream)

Interface to the popen function. It opens a process by creating a pipe, forking and invoking Command on the current shell. Since a pipe is by definition unidirectional the Type argument may be read or write, not both. The stream should be closed using close/1, there is no need for a special pclose command.

The following example demonstrates the use of popen/3 to process the output of a command, as exec/3 would do:

?- popen(ls,read,X),repeat, get0(X,C), (C = -1, ! ; put(C)).

X = 'C:\cygwin\home\administrator' ?

The WIN32 implementation of popen/3 relies on exec/3.

shell(+Command)

Execute command Command under a new shell. YAP will be in background until the command completes. In Unix environments YAP uses the shell given by the environment variable SHELL. In WIN32 environment YAP will use COMSPEC if SHELL is undefined.

shell(+Command)

Execute command Command under a new shell. YAP will be in background until the command completes. In Unix environments YAP uses the shell given by the environment variable SHELL with the option " -c ". In WIN32 environment YAP will use COMSPEC if SHELL is undefined, in this case with the option " /c ".

shell(+Command,-Status)
    Execute command Command under a new shell and unify Status with the exit
    for the command. YAP will be in background until the command completes.
    In Unix environments YAP uses the shell given by the environment variable
    SHELL with the option " -c ". In WIN32 environment YAP will use COMSPEC if
    SHELL is undefined, in this case with the option " /c ".

sleep(+Time)
    Block the current thread for Time seconds. When YAP is compiled without
    multi-threading support, this predicate blocks the YAP process. The number
    of seconds must be a positive number, and it may an integer or a float. The
    Unix implementation uses usleep if the number of seconds is below one, and
    sleep if it is over a second. The WIN32 implementation uses Sleep for both
    cases.

system
    Start a new default shell and leave YAP in background until the shell completes.
    YAP uses /bin/sh in Unix systems and COMSPEC in WIN32.

system(+Command,-Res)
    Interface to system: execute command Command and unify Res with the result.

wait(+PID,-Status)
    Wait until process PID terminates, and return its exits Status.

7.23 Utilities On Terms
The next routines provide a set of commonly used utilities to manipulate terms. Most of
these utilities have been implemented in C for efficiency. They are available through the
use_module(library(terms)) command.

acyclic_term(?Term)
    Succeed if the argument Term is an acyclic term.

cyclic_term(?Term)
    Succeed if the argument Term is a cyclic term.

term_hash(+Term, ?Hash)
    If Term is ground unify Hash with a positive integer calculated from the struc-
ture of the term. Otherwise the argument Hash is left unbound. The range of
the positive integer is from 0 to, but not including, 33554432.

term_hash(+Term, +Depth, +Range, ?Hash)
    Unify Hash with a positive integer calculated from the structure of the term.
    The range of the positive integer is from 0 to, but not including, Range. If
    Depth is ~1 the whole term is considered. Otherwise, the term is considered
    only up to depth 1, where the constants and the principal functor have depth
    1, and an argument of a term with depth I has depth I+1.

term_variables(?Term, -Variables)
    Unify Variables with the list of all variables of term Term. The variables occur
    in the order of their first appearance when traversing the term depth-first, left-
variables_within_term(+Variables, ?Term, -OutputVariables)
Unify OutputVariables with the subset of the variables Variables that occurs in Term.

new_variables_in_term(+Variables, ?Term, -OutputVariables)
Unify OutputVariables with all variables occurring in Term that are not in the list Variables.

variant(?Term1, ?Term2)
Succeed if Term1 and Term2 are variant terms.

subsumes(?Term1, ?Term2)
Succeed if Term1 subsumes Term2. Variables in term Term1 are bound so that the two terms become equal.

subsumes_chk(?Term1, ?Term2)
Succeed if Term1 subsumes Term2 but does not bind any variable in Term1.

variable_in_term(?Term, ?Var)
Succeed if the second argument Var is a variable and occurs in term Term.

unifiable(?Term1, ?Term2, -Bindings)
Succeed if Term1 and Term2 are unifiable with substitution Bindings.

7.24 Trie DataStructure

The next routines provide a set of utilities to create and manipulate prefix trees of Prolog terms. Tries were originally proposed to implement tabling in Logic Programming, but can be used for other purposes. The tries will be stored in the Prolog database and can seen as alternative to assert and record family of primitives. Most of these utilities have been implemented in C for efficiency. They are available through the use_module(library(tries)) command.

trie_open(-Id)
Open a new trie with identifier Id.

trie_close(+Id)
Close trie with identifier Id.

trie_close_all
Close all available tries.

trie_mode(?Mode)
Unify Mode with trie operation mode. Allowed values are either std (default) or rev.

trie_put_entry(+Trie, +Term, -Ref)
Add term Term to trie Trie. The handle Ref gives a reference to the term.

trie_check_entry(+Trie, +Term, -Ref)
Succeeds if a variant of term Term is in trie Trie. An handle Ref gives a reference to the term.

trie_get_entry(+Ref, -Term)
Unify Term with the entry for handle Ref.
trie_remove_entry(+Ref)
Remove entry for handle Ref.

trie_remove_subtree(+Ref)
Remove subtree rooted at handle Ref.

trie_save(+Trie,+FileName)
Dump trie Trie into file FileName.

trie_load(+Trie,+FileName)
Load trie Trie from the contents of file FileName.

trie_stats(-Memory,-Tries,-Entries,-Nodes)
Give generic statistics on tries, including the amount of memory, Memory, the number of tries, Tries, the number of entries, Entries, and the total number of nodes, Nodes.

trie_max_stats(-Memory,-Tries,-Entries,-Nodes)
Give maximal statistics on tries, including the amount of memory, Memory, the number of tries, Tries, the number of entries, Entries, and the total number of nodes, Nodes.

trie_usage(+Trie,-Entries,-Nodes,-VirtualNodes)
Give statistics on trie Trie, the number of entries, Entries, and the total number of nodes, Nodes, and the number of VirtualNodes.

trie_print(+Trie)
Print trie Trie on standard output.

7.25 Call Cleanup

call_cleanup/1 and call_cleanup/2 allow predicates to register code for execution after the call is finished. Predicates can be declared to be fragile to ensure that call_cleanup is called for any Goal which needs it. This library is loaded with the use_module(library(cleanup)) command.

:- fragile P,....,Pn
Declares the predicate P=[module:]name/arity as a fragile predicate, module is optional, default is the current typein module. Whenever such a fragile predicate is used in a query it will be called through call_cleanup/1.

:- fragile foo/1,bar:baz/2.

call_cleanup(:Goal)
Execute goal Goal within a cleanup-context. Called predicates might register cleanup Goals which are called right after the end of the call to Goal. Cuts and exceptions inside Goal do not prevent the execution of the cleanup calls. call_cleanup might be nested.

call_cleanup(:Goal, :CleanUpGoal)
This is similar to call_cleanup/1 with an additional CleanUpGoal which gets called after Goal is finished.
setup_call_cleanup(:Setup, :Goal, :CleanupGoal)

Calls (Setup, Goal). For each successful execution of Setup, calling Goal, the cleanup handler Cleanup is guaranteed to be called exactly once. This will happen after Goal completes, either through failure, deterministic success, commit, or an exception. Setup will contain the goals that need to be protected from asynchronous interrupts such as the ones received from call_with_time_limit/2 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.

Typically, this predicate is used to cleanup permanent data storage required to execute Goal, close file-descriptors, etc. The example below provides a nondeterministic search for a term in a file, closing the stream as needed.

\[
\text{term_in_file}(Term, File) :-
\text{setup_call_cleanup(open(File, read, In),}
\text{term_in_stream(Term, In),}
\text{close(In) ).}
\]

\[
\text{term_in_stream}(Term, In) :-
\text{repeat,}
\text{read(In, T),}
( \ T == \text{end_of_file} -> !, \text{fail} ; \ T = \text{Term}
)\).
\]

Note that it is impossible to implement this predicate in Prolog other than by reading all terms into a list, close the file and call member/2. Without 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/2 can also be used to test determinism of a goal:

\[
?- \text{setup_call_cleanup(true, (X=1;X=2), Det=yes).}
\]

\[
X = 1 ;
X = 2,
Det = yes ;
\]

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, :CleanupGoal)

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. If this unification fails, Cleanup is not called.
on_cleanup(+CleanUpGoal)

Any Predicate might registers a CleanUpGoal. The CleanUpGoal is put onto the current cleanup context. All such CleanUpGoals are executed in reverse order of their registration when the surrounding cleanup-context ends. This call will throw an exception if a predicate tries to register a CleanUpGoal outside of any cleanup-context.

cleanup_all

Calls all pending CleanUpGoals and resets the cleanup-system to an initial state. Should only be used as one of the last calls in the main program.

There are some private predicates which could be used in special cases, such as manually setting up cleanup-contexts and registering CleanUpGoals for other than the current cleanup-context. Read the Source Luke.

7.26 Calls With Timeout

The time_out/3 command relies on the alarm/3 built-in to implement a call with a maximum time of execution. The command is available with the use_module(library(timeout)) command.

time_out(+Goal, +Timeout, -Result)

Execute goal Goal with time limited Timeout, where Timeout is measured in milliseconds. If the goal succeeds, unify Result with success. If the timer expires before the goal terminates, unify Result with time_out.

This command is implemented by activating an alarm at procedure entry. If the timer expires before the goal completes, the alarm will throw an exception timeout.

One should note that time_out/3 is not reentrant, that is, a goal called from time_out should never itself call time_out/3. Moreover, time_out/3 will deactivate any previous alarms set by alarm/3 and vice-versa, hence only one of these calls should be used in a program.

Last, even though the timer is set in milliseconds, the current implementation relies on alarm/3, and therefore can only offer precision on the scale of seconds.

7.27 Updatable Binary Trees

The following queue manipulation routines are available once included with the use_module(library(trees)) command.

get_label(+Index, +Tree, ?Label)

Treats the tree as an array of N elements and returns the Index-th.

list_to_tree(+List, -Tree)

Takes a given List of N elements and constructs a binary Tree.

map_tree(+Pred, +OldTree, -NewTree)

Holds when OldTree and NewTree are binary trees of the same shape and Pred(Old,New) is true for corresponding elements of the two trees.
put_label(+Index, +OldTree, +Label, -NewTree)
    constructs a new tree the same shape as the old which moreover has the same elements except that the Index-th one is Label.

tree_size(+Tree, -Size)
    Calculates the number of elements in the Tree.

tree_to_list(+Tree, -List)
    Is the converse operation to list_to_tree.

### 7.28 Unweighted Graphs

The following graph manipulation routines are based in code originally written by Richard O'Keefe. The code was then extended to be compatible with the SICStus Prolog ugraphs library. The routines assume directed graphs, undirected graphs may be implemented by using two edges. Graphs are represented in one of two ways:

- The P-representation of a graph is a list of (from-to) vertex pairs, where the pairs can be in any old order. This form is convenient for input/output.
- 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.

These built-ins are available once included with the `use_module(library(ugraphs))` command.

vertices_edges_to_ugraph(+Vertices, +Edges, -Graph)
    Given a graph with a set of vertices Vertices and a set of edges Edges, Graph must unify with the corresponding s-representation. Note that the 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 edges are defined implicitly. The next example shows three unconnected edges:

    ?- 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 Graph. In the next example:

    ?- vertices([1-[3, 5], 2-[4], 3-[], 4-[5], 5-[]], V).

    L = [1, 2, 3, 4, 5]

edges(+Graph, -Edges)
    Unify Edges with all edges appearing in graph Graph. In the next example:
?- vertices([1-[3,5],2-[4],3-[],4-[5],5-[]], V).

L = [1,2,3,4,5]

add_vertices(+Graph, +Vertices, -NewGraph)
Unify NewGraph with a new graph obtained by adding the list of vertices Vertices to the graph Graph. In the next example:

?- add_vertices([1-[3,5],2-[4],3-[],4-[5],
      5-[],6-[],7-[],8-[]],[0,2,9,10,11],
      NG).

NG = [0-[],1-[3,5],2-[4],3-[],4-[5],5-[],
      6-[],7-[],8-[],9-[],10-[],11-[]]

del_vertices(+Graph, +Vertices, -NewGraph)
Unify NewGraph with a new graph obtained by deleting the list of vertices Vertices and all the edges that start from or go to a vertex in Vertices to the graph Graph. In the next 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 Edges to the graph Graph. In the next example:

?- add_edges([1-[3,5],2-[4],3-[],4-[5],5-[],
      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 Edges from the graph Graph. Notice that no vertices are deleted. In the next 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). In the next 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-[]]
Notice that an undirected graph is its own transpose.

`neighbors(+Vertex, +Graph, -Vertices)`
Unify `Vertices` with the list of neighbors of vertex `Vertex` in `Graph`. If the vertex is not in the graph fail. In the next example:

```
?- neighbors(4,[1-[3,5],2-[4],3-[]],
   4-[1,2,7,5],5-[],6-[],7-[],8-[]], NL).
```

`NL = [1,2,7,5]`

`neighbours(+Vertex, +Graph, -Vertices)`
Unify `Vertices` with the list of neighbours of vertex `Vertex` in `Graph`. In the next example:

```
?- neighbours(4,[1-[3,5],2-[4],3-[]],
   4-[1,2,7,5],5-[],6-[],7-[],8-[]], NL).
```

`NL = [1,2,7,5]`

`complement(+Graph, -NewGraph)`
Unify `NewGraph` with the graph complementary to `Graph`. In the next 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 the graphs `LeftGraph` and `RightGraph` to form `NewGraph`. In the next example:

```
?- compose([1-[2],2-[3]], [2-[4],3-[1,2,4]], L).
```

`L = [1-[4],2-[1,2,4],3-[]]`

`top_sort(+Graph, -Sort)`
Generate the set of nodes `Sort` as a topological sorting of graph `Graph`, if one is possible. In the next example we show how topological sorting works for a linear graph:

```
?- top_sort([_138-[_219],_219-_139], _139-[]], L).
```

`L = [_138,_219,_139]`

`top_sort(+Graph, -Sort0, -Sort)`
Generate the difference list `Sort-Sort0` as a topological sorting of graph `Graph`, if one is possible.

`transitive_closure(+Graph, +Closure)`
Generate the graph `Closure` as the transitive closure of graph `Graph`. In the next 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(Node, Graph, Vertices)
Unify Vertices with the set of all vertices in graph Graph that are reachable from Node. In the next example:

?- reachable(1,[1-[3,5],2-[4],3-[[]],4-[5],5-[[]]],V).

V = [1,3,5]

### 7.29 Directed Graphs

The following graph manipulation routines use the red-black tree library to try to avoid linear-time scans of the graph for all graph operations. Graphs are represented as a red-black tree, where the key is the vertex, and the associated value is a list of vertices reachable from that vertex through an edge (i.e., a list of edges).

- **dgraph_new(Graph)**
  Create a new directed graph. This operation must be performed before trying to use the graph.

- **dgraph_vertices(Graph, Vertices)**
  Unify Vertices with all vertices appearing in graph Graph.

- **dgraph_edge(N1, N2, Graph)**
  Edge N1-N2 is an edge in directed graph Graph.

- **dgraph_edges(Graph, Edges)**
  Unify Edges with all edges appearing in graph Graph.

- **dgraph_add_vertices(Graph, Vertex, NewGraph)**
  Unify NewGraph with a new graph obtained by adding vertex Vertex to the graph Graph.

- **dgraph_add_vertices(Graph, Vertices, NewGraph)**
  Unify NewGraph with a new graph obtained by adding the list of vertices Vertices to the graph Graph.

- **dgraph_del_vertex(Graph, Vertex, NewGraph)**
  Unify NewGraph with a new graph obtained by deleting vertex Vertex and all the edges that start from or go to Vertex to the graph Graph.

- **dgraph_del_vertices(Graph, Vertices, NewGraph)**
  Unify NewGraph with a new graph obtained by deleting the list of vertices Vertices and all the edges that start from or go to a vertex in Vertices to the graph Graph.

- **dgraph_add_edge(Graph, N1, N2, NewGraph)**
  Unify NewGraph with a new graph obtained by adding the edge N1-N2 to the graph Graph.
dgraph_add_edges(+Graph, +Edges, -NewGraph)
  Unify NewGraph with a new graph obtained by adding the list of edges Edges to the graph Graph.

dgraph_del_edge(+Graph, +N1, +N2, -NewGraph)
  Succeeds if NewGraph unifies with a new graph obtained by removing the edge N1-N2 from the graph Graph. Notice that no vertices are deleted.

dgraph_del_edges(+Graph, +Edges, -NewGraph)
  Unify NewGraph with a new graph obtained by removing the list of edges Edges from the graph Graph. Notice that no vertices are deleted.

dgraph_to_ugraph(+Graph, -UGraph)
  Unify UGraph with the representation used by the ugraphs unweighted graphs library, that is, a list of the form V-Neighbors, where V is a node and Neighbors the nodes children.

ugraph_to_dgraph(+UGraph, -Graph)
  Unify Graph with the directed graph obtained from UGraph, represented in the form used in the ugraphs unweighted graphs library.

dgraph_neighbours(+Vertex, +Graph, -Vertices)
  Unify Vertices with the list of neighbours of vertex Vertex in Graph. If the vertex is not in the graph fail.

dgraph_neighbors(+Vertex, +Graph, -Vertices)
  Unify Vertices with the list of neighbors of vertex Vertex in Graph.

dgraph_complement(+Graph, -NewGraph)
  Unify NewGraph with the graph complementary to Graph.

dgraph_transpose(+Graph, -Transpose)
  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.

dgraph_compose(+Graph1, +Graph2, -ComposedGraph)
  Unify ComposedGraph with a new graph obtained by composing Graph1 and Graph2, i.e., ComposedGraph has an edge V1-V2 if and only if there is a V such that V1-V in Graph1 and V-V2 in Graph2.

dgraph_transitive_closure(+Graph, -Closure)
  Unify Closure with the transitive closure of graph Graph.

dgraph_symmetric_closure(+Graph, -Closure)
  Unify Closure with the symmetric closure of graph Graph, that is, if Closure contains an edge U-V it must also contain the edge V-U.

dgraph_top_sort(+Graph, -Vertices)
  Unify Vertices with the topological sort of graph Graph.

dgraph_top_sort(+Graph, -Vertices, ?Vertices0)
  Unify the difference list Vertices-Vertices0 with the topological sort of graph Graph.
dgraph_min_path(+V1, +V1, +Graph, -Path, ?Costt)
   Unify the list Path with the minimal cost path between nodes N1 and N2 in graph Graph. Path Path has cost Cost.

dgraph_max_path(+V1, +V1, +Graph, -Path, ?Costt)
   Unify the list Path with the maximal cost path between nodes N1 and N2 in graph Graph. Path Path has cost Cost.

dgraph_min_paths(+V1, +Graph, -Paths)
   Unify the list Paths with the minimal cost paths from node N1 to the nodes in graph Graph.

dgraph_isomorphic(+Vs, +NewVs, +G0, -GF)
   Unify the list GF with the graph isomorphic to G0 where vertices in Vs map to vertices in NewVs.

dgraph_path(+Vertex, +Graph, ?Path)
   The path Path is a path starting at vertex Vertex in graph Graph.

dgraph_reachable(+Vertex, +Graph, ?Edges)
   The path Path is a path starting at vertex Vertex in graph Graph.

7.30 Undirected Graphs

The following graph manipulation routines use the red-black tree graph library to implement undirected graphs. Mostly, this is done by having two directed edges per undirected edge.

undgraph_new(+Graph)
   Create a new directed graph. This operation must be performed before trying to use the graph.

undgraph_vertices(+Graph, -Vertices)
   Unify Vertices with all vertices appearing in graph Graph.

undgraph_edge(+N1, +N2, +Graph)
   Edge N1-N2 is an edge in undirected graph Graph.

undgraph_edges(+Graph, -Edges)
   Unify Edges with all edges appearing in graph Graph.

undgraph_add_vertices(+Graph, +Vertices, -NewGraph)
   Unify NewGraph with a new graph obtained by adding the list of vertices Vertices to the graph Graph.

undgraph_del_vertices(+Graph, +Vertices, -NewGraph)
   Unify NewGraph with a new graph obtained by deleting the list of vertices Vertices and all the edges that start from or go to a vertex in Vertices to the graph Graph.

undgraph_add_edges(+Graph, +Edges, -NewGraph)
   Unify NewGraph with a new graph obtained by adding the list of edges Edges to the graph Graph.

undgraph_del_edges(+Graph, +Edges, -NewGraph)
   Unify NewGraph with a new graph obtained by removing the list of edges Edges from the graph Graph. Notice that no vertices are deleted.
undgraph_neighbors(+Vertex, +Graph, -Vertices)
Unify Vertices with the list of neighbors of vertex Vertex in Graph. If the vertex is not in the graph fail.

undgraph_neighbours(+Vertex, +Graph, -Vertices)
Unify Vertices with the list of neighbours of vertex Vertex in Graph.

undgraph_complement(+Graph, -NewGraph)
Unify NewGraph with the graph complementary to Graph.

dgraph_to_undgraph( +DGraph, -UndGraph)
Unify UndGraph with the undirected graph obtained from the directed graph DGraph.

### 7.31 LAM

This library provides a set of utilities for interfacing with LAM MPI. The following routines are available once included with the `use_module(library(lam_mpi))` command. The yap should be invoked using the LAM mpiexec or mpirun commands (see LAM manual for more details).

**mpi_init**  
Sets up the mpi environment. This predicate should be called before any other MPI predicate.

**mpi_finalize**  
Terminates the MPI execution environment. Every process must call this predicate before exiting.

**mpi_comm_size(-Size)**  
Unifies Size with the number of processes in the MPI environment.

**mpi_comm_rank(-Rank)**  
Unifies Rank with the rank of the current process in the MPI environment.

**mpi_version(-Major, -Minor)**  
Unifies Major and Minor with, respectively, the major and minor version of the MPI.

**mpi_send(+Data, +Dest, +Tag)**  
Blocking communication predicate. The message in Data, with tag Tag, is sent immediately to the processor with rank Dest. The predicate succeeds after the message being sent.

**mpi_isend(+Data, +Dest, +Tag, -Handle)**  
Non blocking communication predicate. The message in Data, with tag Tag, is sent whenever possible to the processor with rank Dest. An Handle to the message is returned to be used to check for the status of the message, using the `mpi_wait` or `mpi_test` predicates. Until `mpi_wait` is called, the memory allocated for the buffer containing the message is not released.

**mpi_recv(?Source, ?Tag, -Data)**  
Blocking communication predicate. The predicate blocks until a message is received from processor with rank Source and tag Tag. The message is placed in Data.
mpi_irecv(?Source,?Tag,-Handle)
Non-blocking communication predicate. The predicate returns an Handle for a message that will be received from processor with rank Source and tag Tag. Note that the predicate succeeds immediately, even if no message has been received. The predicate mpi_wait_recv should be used to obtain the data associated to the handle.

mpi_wait_recv(?Handle,-Status,-Data)
Completes a non-blocking receive operation. The predicate blocks until a message associated with handle Hanlde is buffered. The predicate succeeds unifying Status with the status of the message and Data with the message itself.

mpi_test_recv(?Handle,-Status,-Data)
Provides information regarding a handle. If the message associated with handle Hanlde is buffered then the predicate succeeds unifying Status with the status of the message and Data with the message itself. Otherwise, the predicate fails.

mpi_wait(?Handle,-Status)
Completes a non-blocking operation. If the operation was a mpi_send, the predicate blocks until the message is buffered or sent by the runtime system. At this point the send buffer is released. If the operation was a mpi_recv, it waits until the message is copied to the receive buffer. Status is unified with the status of the message.

mpi_test(?Handle,-Status)
Provides information regarding the handle Handle, ie., if a communication operation has been completed. If the operation associate with Hanlde has been completed the predicate succeeds with the completion status in Status, otherwise it fails.

mpi_barrier
Collective communication predicate. Performs a barrier synchronization among all processes. Note that a collective communication means that all processes call the same predicate. To be able to use a regular mpi_recev to receive the messages, one should use mpi_bcast2.

mpi_bcast2(+Root,+Data)
Broadcasts the message Data from the process with rank Root to all other processes.

mpi_bcast3(+Root,+Data,+Tag)
Broadcasts the message Data with tag Tag from the process with rank Root to all other processes.

mpi_ibcast(+Root,+Data,+Tag)
Non-blocking operation. Broadcasts the message Data with tag Tag from the process with rank Root to all other processes.

mpi_gc
Attempts to perform garbage collection with all the open handles associated with send and non-blocking broadcasts. For each handle it tests it and the message has been delivered the handle and the buffer are released.
8 SWI-Prolog Emulation

This library provides a number of SWI-Prolog builtins that are not by default in YAP. This support is loaded with the `expects_dialect(swi)` command.

```prolog
append(?List1, ?List2, ?List3)
   Succeeds when ?List3 unifies with the concatenation of ?List1 and ?List2. The predicate can be used with any instantiation pattern (even three variables).
```

```prolog
between(+Low, +High, ?Value)
   Low and High are integers, High less or equal than Low. If Value is an integer, Low less or equal than Value less or equal than High. When Value is a variable it is successively bound to all integers between Low and High. If High is `inf`, between/3 is true iff Value less or equal than Low, a feature that is particularly interesting for generating integers from a certain value.
```

```prolog
chdir(+Dir)
   Compatibility predicate. New code should use `working_directory/2`.
```

```prolog
concat_atom(+List, -Atom)
   List is a list of atoms, integers or floating point numbers. Succeeds if Atom can be unified with the concatenated elements of List. If List has exactly 2 elements it is equivalent to `atom_concat/3`, allowing for variables in the list.
```

```prolog
concat_atom(?List, +Separator, ?Atom)
   Creates an atom just like `concat_atom/2`, but inserts Separator between each pair of atoms. For example:

   ```prolog
   \n   \n   ?- concat_atom([gnu, gnat], ', ', A).
   \n   A = 'gnu, gnat'
   ```

   (Unimplemented) This predicate can also be used to split atoms by instantiating Separator and Atom:

   ```prolog
   \n   \n   ?- concat_atom(L, -, 'gnu-gnat').
   L = [gnu, gnat]
   ```
```

```prolog
nth1(+Index, ?List, ?Elem)
   Succeeds when the Index-th element of List unifies with Elem. Counting starts at 1.
```

Set environment variable. Name and Value should 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`.

```prolog
setenv(+Name, +Value)
   Set environment variable. Name and Value should 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`.
```

```prolog
term_to_atom(?Term, ?Atom)
   Succeeds 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/1.

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.

@Term1 @= @Term2

True iff Term1 and Term2 are structurally equivalent. I.e. if Term1 and Term2 are variants of each other.

8.1 Invoking Predicates on all Members of a List

All the predicates in this section call a predicate on all members of a list or until the predicate called fails. The predicate is called via call/[2..], which implies common arguments can be put in front of the arguments obtained from the list(s). For example:

?- maplist(plus(1), [0, 1, 2], X).

X = [1, 2, 3]

we will phrase this as “Predicate is applied on ...”

maplist(+Pred,+List)

Pred is applied successively on each element of List until the end of the list or Pred fails. In the latter case maplist/2 fails.

maplist(+Pred,+List1,+List2)

Apply Pred on all successive pairs of elements from List1 and List2. Fails if Pred can not be applied to a pair. See the example above.

maplist(+Pred,+List1,+List2,+List3)

Apply Pred on all successive triples of elements from List1, List2 and List3. Fails if Pred can not be applied to a triple. See the example above.

8.2 Forall

forall(+Cond,+Action)

For all alternative bindings of Cond Action can be proven. The next 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).
9 SWI Global variables

SWI-Prolog global variables are associations between names (atoms) and terms. They differ in various ways from storing information using \texttt{assert/1} or \texttt{recorda/3}.

- The value lives on the Prolog (global) stack. This implies that lookup time is independent from 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 \texttt{nb_setval/2} and backtrackable assignment using \texttt{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 \texttt{b_setval/2} and \texttt{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 \texttt{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 \texttt{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.

\begin{verbatim}
b_setval(+Name,+Value)
Associate the term Value with the atom Name or replaces the currently associated value with Value. If Name does not refer to an existing global variable a variable with initial value $\texttt{[]}$ is created (the empty list). On backtracking the assignment is reversed.

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 \texttt{copy_term/2}) must be taken. The \texttt{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 \texttt{duplicate_term/2} with the atom Name. Note that this can be used to set an initial value other than $\texttt{[]}$ prior to backtrackable assignment.

nb_getval(+Name,-Value)
The \texttt{nb_getval/2} predicate is a synonym for \texttt{b_getval/2}, introduced for compatibility and symmetry. As most scenarios will use a particular global variable either using non-backtrackable or backtrackable assignment, using \texttt{nb_getval/2} can be used to document that the variable is used non-backtrackable.
\end{verbatim}
nb_current(\texttt{?Name,?Value})

Enumerate all defined variables with their value. The order of enumeration is undefined.

nb_delete(\texttt{?Name})

Delete the named global variable.

9.1 Compatibility of Global Variables

Global variables have been introduced by various Prolog implementations recently. YAP follows their implementation in SWI-Prolog, itself based on hProlog by Bart Demoen. Jan and Bart decided that the semantics if hProlog \texttt{nb_setval/2}, which is equivalent to \texttt{nb_linkval/2} is not acceptable for normal Prolog users as the behaviour is influenced by how builtin predicates constructing terms (\texttt{read/1}, \texttt{=../2}, etc.) are implemented.

GNU-Prolog provides a rich set of global variables, including arrays. Arrays can be implemented easily in SWI-Prolog using \texttt{functor/3} and \texttt{setarg/3} due to the unrestricted arity of compound terms.
10 Extensions to Prolog

YAP includes several extensions that are not enabled by default, but that can be used to extend the functionality of the system. These options can be set at compilation time by enabling the related compilation flag, as explained in the Makefile.

10.1 Rational Trees

Prolog unification is not a complete implementation. For efficiency considerations, Prolog systems do not perform occur checks while unifying terms. As an example, $X = a(X)$ will not fail but instead will create an infinite term of the form $a(a(a(a(\ldots))))$, or rational tree.

Rational trees are now supported by default in YAP. In previous versions, this was not the default and these terms could easily lead to infinite computation. For example, $X = a(X), X = X$ would enter an infinite loop.

The RATIONAL_TREES flag improves support for these terms. Internal primitives are now aware that these terms can exist, and will not enter infinite loops. Hence, the previous unification will succeed. Another example, $X = a(X), \text{ground}(X)$ will succeed instead of looping. Other affected built-ins include the term comparison primitives, numbervars/3, copy_term/2, and the internal data base routines. The support does not extend to Input/Output routines or to assert/1 YAP does not allow directly reading rational trees, and you need to use write_depth/2 to avoid entering an infinite cycle when trying to write an infinite term.

10.2 Co-routining

Prolog uses a simple left-to-right flow of control. It is sometimes convenient to change this control so that goals will only be executed when conditions are fulfilled. This may result in a more "data-driven" execution, or may be necessary to correctly implement extensions such as negation by default.

The COROUTINING flag enables this option. Note that the support for coroutining will in general slow down execution.

The following declaration is supported:

\[ \text{block/1} \]

The argument to block/1 is a condition on a goal or a conjunction of conditions, with each element separated by commas. Each condition is of the form \textit{predname}(C1,\ldots,CN), where \(N\) is the arity of the goal, and each \(CI\) is of the form \(\_\), if the argument must suspend until the variable is bound, or \(?\), otherwise.

\[ \text{wait/1} \]

The argument to wait/1 is a predicate descriptor or a conjunction of these predicates. These predicates will suspend until their first argument is bound.

The following primitives are supported:

\[ \text{dif}(X,Y) \]

Succeed if the two arguments do not unify. A call to dif/2 will suspend if unification may still succeed or fail, and will fail if they always unify.

\[ \text{freeze}(?X,;G) \]

Delay execution of goal \(G\) until the variable \(X\) is bound.
frozen(X,G)
    Unify G with a conjunction of goals suspended on variable X, or true if no goal has suspended.

when(+C, :G)
    Delay execution of goal G until the conditions C are satisfied. The conditions are of the following form:
    C1, C2    Delay until both conditions C1 and C2 are satisfied.
    C1 ; C2    Delay until either condition C1 or condition C2 is satisfied.
    ? = (V1, C2)   Delay until terms V1 and V1 have been unified.

nonvar(V)
    Delay until variable V is bound.

ground(V)
    Delay until variable V is ground.

Note that when/2 will fail if the conditions fail.

call_residue(:G,L)
    Call goal G. If subgoals of G are still blocked, return a list containing these goals and the variables they are blocked in. The goals are then considered as unblocked. The next example shows a case where dif/2 suspends twice, once outside call_residue/2, and the other inside:
    ?- dif(X,Y),
    call_residue((dif(X,Y),(X = f(Z) ; Y = f(Z))), L).

    X = f(Z),
    L = [[Y]-dif(f(Z),Y)],
    dif(f(Z),Y) ? ;

    Y = f(Z),
    L = [[X]-dif(X,f(Z))],
    dif(X,f(Z)) ? ;

    no

The system only reports one invocation of dif/2 as having suspended.

call_residue_vars(:G,L)
    Call goal G and unify L with a list of all constrained variables created during execution of G:
    ?- dif(X,Z), call_residue_vars(dif(X,Y),L).
    dif(X,Z), call_residue_vars(dif(X,Y),L).
    L = [Y],
    dif(X,Z),
    dif(X,Y) ? ;

    no
11 Attributed Variables

YAP supports attributed variables, originally developed at OFAI by Christian Holzbaur. Attributes are a means of declaring that an arbitrary term is a property for a variable. These properties can be updated during forward execution. Moreover, the unification algorithm is aware of attributed variables and will call user defined handlers when trying to unify these variables.

Attributed variables provide an elegant abstraction over which one can extend Prolog systems. Their main application so far has been in implementing constraint handlers, such as Holzbaur’s CLPQR, Fruewirth and Holzbaur’s CHR, and CLP(BN).

Different Prolog systems implement attributed variables in different ways. Traditionally, YAP has used the interface designed by SICStus Prolog. This interface is still available in the atts library, but from YAP-6.0.3 we recommend using the hProlog, SWI style interface. The main reason to do so is that most packages included in YAP that use attributed variables, such as CHR, CLP(FD), and CLP(QR), rely on the SWI-Prolog interface.

11.1 hProlog and SWI-Prolog style Attribute Declarations

The following documentation is taken from the SWI-Prolog manual.

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)).

donmain(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
    ```

%
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 it can be unified with. In the second clause first associates the domain with a fresh variable and then unifies X to this variable to deal with the possibility that X already has a domain. The predicate attr_unify_hook/2 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), simply 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.

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 whether or not the named attribute is present.

attr_unify_hook(+AttValue,+VarValue)
Hook that must be defined in the module an attributed variable refers to. Is 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 attribute 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.

attribute_goals(+Var,-Gs,+GsRest)
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.

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 variables. 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_atts/2 for a description of Attributes.

del_atts(+Var)
If Var is an attributed variable, delete all its attributes. In all other cases, this predicate succeeds without side-effects.

term_attvars(+Term,-AttVars)
AttVars is a list of all attributed variables in Term and its attributes. I.e., term_attvars/2 works recursively through attributes. This predicate is Cycle-safe.

copy_term(?TI,-TF,-Goals)
Term TF is a variant of the original term TI, such that for each variable V in the term TI there is a new variable V' in term TF without any attributes attached. Attributed variables are thus converted to standard variables. Goals is unified
with a list that represents the attributes. The goal `maplist(call,Goals)` can be called to recreate the attributes.

Before the actual copying, `copy_term/3` calls `attribute_goals/1` in the module where the attribute is defined.

```prolog
copy_term_nat(?TI,-TF)
```

As `copy_term/2`. Attributes however, are not copied but replaced by fresh variables.

### 11.2 SICStus Prolog style Attribute Declarations

Old style attribute declarations are activated through loading the library `atts`. The command

```prolog
| ?- use_module(library(atts)).
```

enables this form of use of attributed variables. The package provides the following functionality:

- Each attribute must be declared first. Attributes are described by a functor and are declared per module. Each Prolog module declares its own sets of attributes. Different modules may have different functors with the same module.
- The built-in `put_atts/2` adds or deletes attributes to a variable. The variable may be unbound or may be an attributed variable. In the latter case, YAP discards previous values for the attributes.
- The built-in `get_atts/2` can be used to check the values of an attribute associated with a variable.
- The unification algorithm calls the user-defined predicate `verify_attributes/3` before trying to bind an attributed variable. Unification will resume after this call.
- The user-defined predicate `attribute_goal/2` converts from an attribute to a goal.
- The user-defined predicate `project_attributes/2` is used from a set of variables into a set of constraints or goals. One application of `project_attributes/2` is in the top-level, where it is used to output the set of floundered constraints at the end of a query.

### 11.2.1 Attribute Declarations

Attributes are compound terms associated with a variable. Each attribute has a `name` which is `private` to the module in which the attribute was defined. Variables may have at most one attribute with a name. Attribute names are defined with the following declaration:

```prolog
:- attribute AttributeSpec, ..., AttributeSpec.
```

where each `AttributeSpec` has the form `(Name/Arity)`. One single such declaration is allowed per module `Module`.

Although the YAP module system is predicate based, attributes are local to modules. This is implemented by rewriting all calls to the built-ins that manipulate attributes so that attribute names are preprocessed depending on the module. The `user:goal_expansion/3` mechanism is used for this purpose.
11.2.2 Attribute Manipulation

The attribute manipulation predicates always work as follows:

1. The first argument is the unbound variable associated with attributes,
2. The second argument is a list of attributes. Each attribute will be a Prolog term or a constant, prefixed with the + and - unary operators. The prefix + may be dropped for convenience.

The following three procedures are available to the user. Notice that these built-ins are rewritten by the system into internal built-ins, and that the rewriting process depends on the module on which the built-ins have been invoked.

Module:get_atts(-Var, ?ListOfAttributes)
   Unify the list ?ListOfAttributes with the attributes for the unbound variable Var. Each member of the list must be a bound term of the form +(Attribute), -(Attribute) (the kbd prefix may be dropped). The meaning of + and - is:

+ (Attribute)
   Unifies Attribute with a corresponding attribute associated with Var, fails otherwise.

-(Attribute)
   Succeeds if a corresponding attribute is not associated with Var. The arguments of Attribute are ignored.

Module:put_atts(-Var, ?ListOfAttributes)
   Associate with or remove attributes from a variable Var. The attributes are given in ?ListOfAttributes, and the action depends on how they are prefixed:

+ (Attribute)
   Associate Var with Attribute. A previous value for the attribute is simply replace (like with set_mutable/2).

-(Attribute)
   Remove the attribute with the same name. If no such attribute existed, simply succeed.

11.2.3 Attributed Unification

The user-predicate predicate verify_attributes/3 is called when attempting to unify an attributed variable which might have attributes in some Module.

Module:verify_attributes(-Var, +Value, -Goals)
   The predicate is called when trying to unify the attributed variable Var with the Prolog term Value. Note that Value may be itself an attributed variable, or may contain attributed variables. The goal verify_attributes/3 is actually called before Var is unified with Value.

It is up to the user to define which actions may be performed by verify_attributes/3 but the procedure is expected to return in Goals a list of goals to be called after Var is unified with Value. If verify_attributes/3 fails, the unification will fail.
Notice that the `verify_attributes/3` may be called even if `Var` has no attributes in module `Module`. In this case the routine should simply succeed with `Goals` unified with the empty list.

`attvar(-Var)`
Succeed if `Var` is an attributed variable.

### 11.2.4 Displaying Attributes
Attributes are usually presented as goals. The following routines are used by built-in predicates such as `call_residue/2` and by the Prolog top-level to display attributes:

```prolog
Module:attribute_goal(-Var, -Goal)
```
User-defined procedure, called to convert the attributes in `Var` to a `Goal`. Should fail when no interpretation is available.

```prolog
Module:project_attributes(-QueryVars, +AttrVars)
```
User-defined procedure, called to project the attributes in the query, `AttrVars`, given that the set of variables in the query is `QueryVars`.

### 11.2.5 Projecting Attributes
Constraint solvers must be able to project a set of constraints to a set of variables. This is useful when displaying the solution to a goal, but may also be used to manipulate computations. The user-defined `project_attributes/2` is responsible for implementing this projection.

```prolog
Module:project_attributes(+QueryVars, +AttrVars)
```
Given a list of variables `QueryVars` and list of attributed variables `AttrVars`, project all attributes in `AttrVars` to `QueryVars`. Although projection is constraint system dependent, typically this will involve expressing all constraints in terms of `QueryVars` and considering all remaining variables as existentially quantified.

Projection interacts with `attribute_goal/2` at the Prolog top level. When the query succeeds, the system first calls `project_attributes/2`. The system then calls `attribute_goal/2` to get a user-level representation of the constraints. Typically, `attribute_goal/2` will convert from the original constraints into a set of new constraints on the projection, and these constraints are the ones that will have an `attribute_goal/2` handler.

### 11.2.6 Attribute Examples
The following two examples are taken from the SICStus Prolog manual. It sketches the implementation of a simple finite domain “solver”. Note that an industrial strength solver would have to provide a wider range of functionality and that it quite likely would utilize a more efficient representation for the domains proper. The module exports a single predicate `domain(-Var, ?Domain)` which associates `Domain` (a list of terms) with `Var`. A variable can be queried for its domain by leaving `Domain` unbound.

We do not present here a definition for `project_attributes/2`. Projecting finite domain constraints happens to be difficult.

```prolog
:- module(domain, [domain/2]).
```
:- use_module(library(atts)).
:- use_module(library(ordsets), [ord_intersection/3, ord_intersect/2, list_to_ord_set/2]).

:- attribute dom/1.

verify_attributes(Var, Other, Goals) :-
  get_atts(Var, dom(Da)), !, % are we involved?
  ( var(Other) -> % must be attributed then
    ( get_atts(Other, dom(Db)) -> % has a domain?
      ord_intersection(Da, Db, Dc),
      Dc = [El|Els], % at least one element
      ( Els = [] -> % exactly one element
        Goals = [Other=El] % implied binding
      ; Goals = [],
        put_atts(Other, dom(Dc))% rescue intersection
      )
    ; Goals = [],
      put_atts(Other, dom(Da))% rescue the domain
    )
  ; Goals = [],
    ord_intersect([Other], Da) % value in domain?
 ).

verify_attributes(_, _, []). % unification triggered
% because of attributes
% in other modules

attribute_goal(Var, domain(Var,Dom)) :- % interpretation as goal
  get_atts(Var, dom(Dom)).

domain(X, Dom) :-
  var(Dom), !,
  get_atts(X, dom(Dom)).

domain(X, List) :-
  list_to_ord_set(List, Set),
  Set = [El|Els], % at least one element
  ( Els = [] -> % exactly one element
    X = El % implied binding
  ; put_atts(Fresh, dom(Set)),
    X = Fresh % may call
  % verify_attributes/3
  ).
Note that the "implied binding" \( \text{Other}=\text{El} \) was deferred until after the completion of \( \text{verify_attribute/3} \). Otherwise, there might be a danger of recursively invoking \( \text{verify_attribute/3} \), which might bind \( \text{Var} \), which is not allowed inside the scope of \( \text{verify_attribute/3} \). Deferring unifications into the third argument of \( \text{verify_attribute/3} \) effectively serializes the calls to \( \text{verify_attribute/3} \).

Assuming that the code resides in the file ‘\text{domain.yap}’, we can use it via:

\[
| \text{?- use_module(domain).} \\
\]

Let’s test it:

\[
| \text{?- domain(X,\([5,6,7,1]\)), domain(Y,\([3,4,5,6]\)), domain(Z,\([1,6,7,8]\)).} \\
\]

\[
\text{domain(X,\([1,5,6,7]\)),} \\
\text{domain(Y,\([3,4,5,6]\)),} \\
\text{domain(Z,\([1,6,7,8]\)) ?} \\
\]

\[
\text{yes} \\
| \text{?- domain(X,\([5,6,7,1]\)), domain(Y,\([3,4,5,6]\)), domain(Z,\([1,6,7,8]\)),} \\
\text{X=Y.} \\
\]

\[
\text{Y = X,} \\
\text{domain(X,\([5,6]\)),} \\
\text{domain(Z,\([1,6,7,8]\)) ?} \\
\]

\[
\text{yes} \\
| \text{?- domain(X,\([5,6,7,1]\)), domain(Y,\([3,4,5,6]\)), domain(Z,\([1,6,7,8]\)),} \\
\text{X=Y, Y=Z.} \\
\]

\[
X = 6, \\
Y = 6, \\
Z = 6 \\
\]

To demonstrate the use of the \text{Goals} argument of \( \text{verify_attributes/3} \), we give an implementation of \( \text{freeze/2} \). We have to name it \( \text{myfreeze/2} \) in order to avoid a name clash with the built-in predicate of the same name.

\[
:- \text{module(myfreeze, [myfreeze/2]).} \\
\]

\[
:- \text{use_module(library(atts)).} \\
\]

\[
:- \text{attribute frozen/1.} \\
\]

\[
\text{verify_attributes(Var, Other, Goals) :-} \\
\text{\( \text{get_atts(Var, frozen(Fa))}, !, \) % are we involved?} \\
\text{\( \text{( var(Other) -}\to \) % must be attributed then} \\
\text{\( \text{( get_atts(Other, frozen(Fb)) \) } \) % has a pending goal?} \\
\text{\( -\to \text{ put_atts(Other, frozen((Fa,Fb))) } \) % rescue conjunction} \\
\text{\( ; \text{ put_atts(Other, frozen(Fa)) } \) % rescue the pending goal} \\
) \\
\]

Chapter 11: Attributed Variables

Goals = []
; Goals = [Fa]
).

verify_attributes(_, _, []).

attribute_goal(Var, Goal) :- % interpretation as goal
    get_atts(Var, frozen(Goal)).

myfreeze(X, Goal) :-
    put_atts(Fresh, frozen(Goal)),
    Fresh = X.

Assuming that this code lives in file ‘myfreeze.yap’, we would use it via:

| ?- use_module(myfreeze).
| ?- myfreeze(X,print(bound(x,X))), X=2.

bound(x,2) % side effect
X = 2 % bindings

The two solvers even work together:

| ?- myfreeze(X,print(bound(x,X))), domain(X,[1,2,3]),
    domain(Y,[2,10]), X=Y.

bound(x,2) % side effect
X = 2, % bindings
Y = 2

The two example solvers interact via bindings to shared attributed variables only. More complicated interactions are likely to be found in more sophisticated solvers. The corresponding verify_attributes/3 predicates would typically refer to the attributes from other known solvers/modules via the module prefix in Module:get_atts/2.
12 Constraint Logic Programming over Reals

YAP now uses the CLP(R) package developed by Leslie De Koninck, K.U. Leuven as part of a thesis with supervisor Bart Demoen and daily advisor Tom Schrijvers, and distributed with SWI-Prolog.

This CLP(R) system is a port of the CLP(Q,R) system of Sicstus Prolog and YAP 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, http://www.ai.univie.ac.at/cgi-bin/tr-online?number=95-09 This port only contains the part concerning real arithmetics. This manual is roughly based on the manual of the above mentioned CLP(QR) implementation.

Please note that the 'clpr' library is not an autoload library and therefore this library must be loaded explicitly before using it:

```prolog
:- use_module(library(clpr)).
```

12.1 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.

inf(+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.

min(+Expression)

 Minimizes Expression within the current constraint store. This is the same as computing the infimum and equating the expression to that infimum.

max(+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,-Vertext,+Eps)

 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)
The same as bb_inf/5 but without returning the values of the integers and with an eps of 0.001.

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.

12.2 Syntax of the predicate arguments
The arguments of the predicates defined in the subsection above are defined in the following table. Failing to meet the syntax rules will result in an exception.

<Constraints> ---> <Constraint> \ single constraint \n | <Constraint> , <Constraints> \ conjunction \n | <Constraint> ; <Constraints> \ disjunction \n
<Constraint> ---> <Expression> {<} <Expression> \ less than \n | <Expression> {>} <Expression> \ greater than \n | <Expression> {=} <Expression> \ less or equal \n | {=}(<Expression>, <Expression>) \ less or equal \n | <Expression> {=} <Expression> \ greater or equal \n | <Expression> {=} <Expression> \ not equal \n | <Expression> =:= <Expression> \ equal \n | <Expression> = <Expression> \ equal \n
<Expression> ---> <Variable> \ Prolog variable \n | <Number> \ Prolog number (float, integer) \n | +<Expression> \ unary plus \n | -<Expression> \ unary minus \n | <Expression> + <Expression> \ addition \n | <Expression> - <Expression> \ substraction \n | <Expression> * <Expression> \ multiplication \n | <Expression> / <Expression> \ division \n | abs(<Expression>) \ absolute value \n | sin(<Expression>) \ sine \n | cos(<Expression>) \ cosine \n | tan(<Expression>) \ tangent \n | exp(<Expression>) \ exponent \n | pow(<Expression>) \ exponent \n | <Expression> {^} <Expression> \ exponent \n | min(<Expression>, <Expression>) \ minimum \n
12.3 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\}
  X = Y

'Unification with a number'
  \{X =:= 5.0\}
  \{X = 5.0\}
  X = 5.0

12.4 Non-Linear Constraints

In this version, non-linear constraints do not get solved until certain conditions are satisfied. We call these conditions the isolation axioms. They are given in the following table.

\[
\begin{align*}
A = B \times C & \quad \text{when } B \text{ or } C \text{ is ground or } \quad A = B \times 4 \quad \text{or} \quad 20 = 5 \times C \quad \text{or} \quad 20 = B \times 4 \\
A = B / C & \quad \text{when } C \text{ is ground or } \quad A = B / 3 \quad \text{or} \quad 4 = 12 / C \\
X = \min(Y,Z) & \quad \text{when } Y \text{ and } Z \text{ are ground or } \quad X = \min(4,3) \\
X = \max(Y,Z) & \quad Y \text{ and } Z \text{ are ground } \quad X = \max(4,3) \\
X = \abs(Y) & \quad \text{Y is ground } \quad X = \abs(-7) \\
X = \pow(Y,Z) & \quad \text{when } X \text{ and } Y \text{ are ground or } \quad 8 = 2 \times Z \\
X = \exp(Y,Z) & \quad X \text{ and } Z \text{ are ground } \quad 8 = Y \times 3 \\
X = Y \times Z & \quad Y \text{ and } Z \text{ are ground } \quad X = 2 \times 3 \\
X = \sin(Y) & \quad \text{when } X \text{ is ground or } \quad 1 = \sin(Y) \\
X = \cos(Y) & \quad Y \text{ is ground } \quad X = \sin(1.5707) \\
X = \tan(Y) & \\
\end{align*}
\]
13 CHR: Constraint Handling Rules

This chapter is written by Tom Schrijvers, K.U. Leuven for the hProlog system. Adjusted by Jan Wielemaker to fit the SWI-Prolog documentation infrastructure and remove hProlog specific references.

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 licenced under compatible conditions with permission from the authors.

The main reference for SWI-Prolog’s CHR system is:


13.1 Introduction

Constraint Handling Rules (CHR) is a committed-choice bottom-up 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, 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 [Fruhwirth:98]. More background on CHR can be found at the CHR web site.

13.2 Syntax and Semantics

13.2.1 Syntax

The syntax of CHR rules in hProlog is the following:

```prolog
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 --> constraints, [atom('<=')]], guard, body.
```

```prolog
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 --> constraints, [atom('<=')]], guard, body.
```
propagation_rule --> constraints, [atom('==>')], guard, body.
simpagation_rule --> constraints, [atom('\')], constraints, [atom('<=>')],
guard, body.

constraints --> constraint, constraint_id.
constraints --> constraint, [atom(',')], constraints.

constraint --> compound_term.
constraint_id --> [].
constraint_id --> [atom('#')], variable.

guard --> [].
guard --> goal, [atom('|')].

body --> goal.
pragma --> [].
pragma --> [atom('pragma')], actual_pragmas.

actual_pragmas --> actualPragma.
actual_pragmas --> actualPragma, [atom(',')], actual_pragmas.

actualPragma --> [atom('passive(')], variable, [atom(')')]].

Additional syntax-related terminology:

- **head**: the constraints in an actual_rule before the arrow (either <= or ==>)

### 13.2.2 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 constraint can be found, the matching fails 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 nonvariable 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 their 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: \[\text{constraints}_1, \text{constraints}_2 \iff \text{constraints}_1, \text{body} \] Namely, in the simpagation form:

\[
\text{constraints}_1 \ \text{\textbackslash} \ \text{constraints}_2 \iff \text{body}
\]

constraint1 constraints are not called in the body.

**Rule Names**
Naming a rule is optional and has no semantical 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 act as a passive constraint in that rule.

Additional pragmas may be released in the future.

**Options**
It is possible to specify options that apply to all the CHR rules in the module. Options are specified with the `option/2` declaration:

\[
\text{option}(\text{Option}, \text{Value}).
\]

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.
optimize

This is an experimental option controlling 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 commandline option `-O` provides full CHR optimization. If optimization is enabled, debugging should be disabled.

debug

This options enables or disables the possibility to debug the CHR code. Possible values are on (default) and off. See `debugging` 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 should be disabled.

mode

This option specifies the mode for a particular constraint. The value is a term with functor and arity equal to that of a constraint. The arguments can be one of -, + or ?. The latter is the default. The meaning is the following:

- 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.

The declaration is used by the compiler for various optimizations. Note that it is up to the user the ensure that the mode declaration is correct with respect to the use of the constraint. This option may occur once for each constraint.

type_declaration

This option specifies the argument types for a particular constraint. The value is a term with functor and arity equal to that of a constraint. The arguments can be a user-defined type or one of the built-in types:

int

The corresponding argument of every occurrence of the constraint is an integer number.

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.

Currently, type declarations are only used to improve certain optimizations (guard simplification, occurrence subsumption, ...).

type_definition

This option defines a new user-defined type which can be used in type declarations. The value is a term of the form `type(name, list)`, where name is a term and list is a list of alternatives. Variables can be used to define generic types. Recursive definitions are allowed. Examples are
The mode, type_declarion and type_definition options are provided for backward compatibility. The new syntax is described below.

### 13.3 CHR in YAP Programs

#### 13.3.1 Embedding in Prolog Programs

The CHR constraints defined in a particular `chr` file are associated with a module. The default module is `user`. One should never load different `chr` files with the same CHR module name.

#### 13.3.2 Constraint declaration

Every constraint used in CHR rules has to be declared. There are two ways to do this. The old style is as follows:

```prolog
:- constraints foo(+,?), bar/0.
```

The new style is as follows:

```prolog
:- chr_type list(T) ---> []; [T|list(T)].
:- constraints foo(+list(int),?float), bar.
```

#### 13.3.3 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 of the format below.

```prolog
:- constraints ...
```

It is advised to define CHR rules in a module file, where the module declaration is immediately followed by including the `chr` library as exemplified below:

```prolog
:- module(zebra, [ zebra/0 ]).
:- use_module(library(chr)).
:- constraints ...
```

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.

### 13.4 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.

13.4.1 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, if it had
  been inserted.
- **try**: An active constraints tries a rule with possibly some passive constraints. The
  try port is entered just before committing to the rule.
- **apply**: An active constraints commits to a rule with possibly some passive constraints.
  The apply port is entered just after committing to the rule.

13.4.2 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 toplevel. See break/0.
abort
   Exit to the toplevel. See abort/0.
fail
   Insert failure in execution.
help
   Print the above available debug options.

13.4.3 CHR Debugging Predicates
The `chr` module contains several predicates that allow inspecting and printing the content of the constraint store.

```
chr_trace/0
   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/0
   De-activate the CHR tracer. By default the CHR tracer is activated and deactivated automatically by the Prolog predicates trace/0 and notrace/0.

chr_leash/0
   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 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 toplevel at the end of each query for every CHR module currently loaded. The prolog-flag chr_toplevel_show_store controls whether the toplevel shows the constraint stores. The value true enables it. Any other value disables it.
```

13.5 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.

```
:- module(leq,[cycle/3, leq/2]).
:- use_module(library(chr)).

:- constraints 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).
```
cycle(X,Y,Z):-
    leq(X,Y),
    leq(Y,Z),
    leq(Z,X).

- The program below implements a simple finite domain constraint solver.

```prolog
:- module(dom,[dom/2]).
:- use_module(library(chr)).
:- constraints dom/2.

dom(X,[]) <-> fail.

dom(X,[Y]) <-> X = Y.

dom(X,L1), dom(X,L2) <-> intersection(L1,L2,L3), dom(X,L3).

intersection([],_,[]).
intersection([H|T],L2,[H|L3]) :-
    member(H,L2), !,
    intersection(T,L2,L3).
intersection([_|T],L2,L3) :-
    intersection(T,L2,L3).
```

13.6 Compatibility with SICStus CHR

There are small differences between CHR in SWI-Prolog and newer YAPs and SICStus and older versions of YAP. Besides differences in available options and pragmas, the following differences should be noted:

[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.

[Sourcefile naming]

SICStus uses a two-step compiler, where ‘chr’ files are first translated into ‘pl’ files. For SWI-Prolog CHR rules may be defined in a file with any extension.

13.7 Guidelines

In this section we cover several guidelines on how to use CHR to write constraint solvers and how to do so efficiently.

[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 \setminus constraint \iff \text{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’).
14 Logtalk

The Logtalk object-oriented extension is available after running its standalone installer by using the `yap1gt` command in POSIX systems or by using the Logtalk - YAP shortcut in the Logtalk program group in the Start Menu on Windows systems. For more information please see the URL http://logtalk.org/.
15 Threads

YAP implements a SWI-Prolog compatible multithreading library. Like in SWI-Prolog, Prolog threads have their own stacks and only share the Prolog heap: predicates, records, flags and other global non-backtrackable data. The package is based on the POSIX thread standard (Butenhof:1997:PPT) used on most popular systems except for MS-Windows.

15.1 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. Currently defined options are:

- **stack** Set the limit in K-Bytes to which the Prolog stacks of this thread may grow. If omitted, the limit of the calling thread is used. See also the commandline -S option.
- **trail** Set the limit in K-Bytes to which the trail stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the commandline option -T.
- **alias** Associate an alias-name with the thread. This named may be used to refer to the thread and remains valid until the thread is joined (see thread_join/2).
- **at_exit** Define an exit hook for the thread. This hook is called when the thread terminates, no matter its exit status.
- **detached** If false (default), the thread can be waited for using thread_join/2. thread_join/2 must be called on this thread to reclaim the all resources associated to 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.

The Goal argument is copied to the new Prolog engine. This implies 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_create(:Goal, -Id)
Create a new Prolog thread using default options. See thread_create/3.

thread_create(:Goal)
Create a new Prolog detached thread using default options. See thread_create/3.

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 thread with 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 current_thread/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:

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 option at thread_create/3 at runtime. Id is the identifier of the thread placed in detached state.

One of the possible applications is to simplify debugging. Threads that are created as detached leave no traces if they crash. For not-detached threads the status can be inspected using current_thread/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_yield
Voluntarily relinquish the processor.

thread_exit(+Term)
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.

thread_at_exit(:Term)
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 ran regardless of why the execution of the thread has been completed. As these hooks are run, the return-code is already available through thread_property/2 using the result of thread_self/1 as thread-identifier. If you want to guarantee the execution of an exit hook no matter how the thread terminates (the thread can be aborted before reaching the thread_at_exit/1 call), consider using instead the at_exit/1 option of thread_create/3.
Chapter 15: Threads

**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.

**thread_sleep(+Time)**
Make current thread sleep for `Time` seconds. `Time` may be an integer or a floating point number. When time is zero or a negative value the call succeeds and returns immediately. This call should not be used if alarms are also being used.

### 15.2 Monitoring Threads

Normal multi-threaded applications should not need these 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.

**thread_property(?Id, ?Property)**
Enumerates the properties of the specified thread. Calling `thread_property/2` does not influence any thread. See also `thread_join/2`. For threads that have an alias-name, this name can be used in `Id` instead of the numerical thread identifier. `Property` is one of:

- **status(Status)**
  The thread status of a thread (see below).

- **alias(Alias)**
  The thread alias, if it exists.

- **at_exit(AtExit)**
  The thread exit hook, if defined (not available if the thread is already terminated).

- **detached(Boolean)**
  The detached state of the thread.

- **stack(Size)**
  The thread stack data-area size.

- **trail(Size)**
  The thread trail data-area size.

- **system(Size)**
  The thread system data-area size.

**current_thread(+Id, -Status)**
Enumerates identifiers and status of all currently known threads. Calling `current_thread/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 numerical thread identifier. `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).

thread_statistics(+Id, +Key, -Value)
Obtains statistical information on thread Id as statistics/2 does in single-threaded applications. This call returns all keys of statistics/2, although only information statistics about the stacks and CPU time yield different values for each thread.

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 times the calling thread has to wait for the mutex. The collision-count is not available on Windows as this would break portability to Windows-95/98/ME or significantly harm performance. Generally collision count is close to zero on single-CPU hardware.

threads  Prints a table of current threads and their status.

15.3 Thread communication

15.3.1 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/2.

thread_send_message(+Term)
Places Term in the message-queue of the thread running the goal. 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.

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 wakeup 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.

**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 and this predicate returns.

Please note that not-unifying messages remain in the queue. After the following has been executed, thread 1 has the term gnu in its queue and continues execution using A is gnat.

```prolog
<thread 1>
  thread_get_message(a(A)),

<thread 2>
  thread_send_message(b(gnu)),
  thread_send_message(a(gnat)),
```

See also **thread_peek_message/1**.

**message_queue_create(?Queue)**
If Queue is an atom, create a named queue. To avoid ambiguity on **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_destroy(+Queue)**
Destroy a message queue created with **message_queue_create/1**. It is not allows to destroy the queue of a thread. Neither is it allowed to destroy a queue other threads are waiting for or, for anonymous message queues, may try to wait for later.

**thread_get_message(+Queue, ?Term)**
As **thread_get_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.

**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.
thread_peek_message(+Queue, ?Term)
    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.

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 given Id and number of workers.
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.

% work(+Id, +Goal)
% % Post work to be done by the pool
work(Id, Goal) :-
    thread_send_message(Id, Goal).

15.3.2 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 multi-threaded 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 do 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 multi-threading, 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.

15.3.3 Threads and Dynamic Predicates

Besides queues threads can share and exchange data using dynamic predicates. The multi-threaded 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 primitive from thread synchronisation should be used to ensure 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)
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 as the thread terminates these are automatically reclaimed by the system. The thread_local property implies the property dynamic.

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 as 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).

15.4 Thread Synchronisation

All internal Prolog operations are thread-safe. This implies 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 MUTual EXclusive device, which implies 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:

```prolog
:- 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_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 choicepoints, 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/2`.

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, the mutex becomes available for locking by other threads. If another thread has locked the mutex the calling thread is suspended until to 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.

`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.

current_mutex(?MutexId, ?ThreadId, ?Count)
Enumerates all existing mutexes. If the mutex is held by some thread, ThreadId is unified with the identifier of the holding thread and Count with the recursive count of the mutex. Otherwise, ThreadId is [] and Count is 0.
16 Parallelism

There has been a sizeable amount of work on an or-parallel implementation for YAP, called YAPOr. Most of this work has been performed by Ricardo Rocha. In this system parallelism is exploited implicitly by running several alternatives in or-parallel. This option can be enabled from the configure script or by checking the system’s Makefile.

YAPOr is still a very experimental system, going through rapid development. The following restrictions are of note:

- **YAPOr** currently only supports the Linux/X86 and SPARC/Solaris platforms. Porting to other Unix-like platforms should be straightforward.
- **YAPOr** does not support parallel updates to the data-base.
- **YAPOr** does not support opening or closing of streams during parallel execution.
- Garbage collection and stack shifting are not supported in **YAPOr**.
- Built-ins that cause side-effects can only be executed when left-most in the search-tree. There are no primitives to provide asynchronous or cavalier execution of these built-ins, as in Aurora or Muse.
- **YAP** does not support voluntary suspension of work.

We expect that some of these restrictions will be removed in future releases.
17 Tabling

YAPTab is the tabling engine that extends YAP’s execution model to support tabled evaluation for definite programs. YAPTab was implemented by Ricardo Rocha and its implementation is largely based on the ground-breaking design of the XSB Prolog system, which implements the SLG-WAM. Tables are implemented using tries and YAPTab supports the dynamic intermixing of batched scheduling and local scheduling at the subgoal level. Currently, the following restrictions are of note:

- YAPTab does not handle tabled predicates with loops through negation (undefined behaviour).
- YAPTab does not handle tabled predicates with cuts (undefined behaviour).
- YAPTab does not support coroutining (configure error).
- YAPTab does not support tabling dynamic predicates (permission error).

To experiment with YAPTab use --enable-tabling in the configure script or add -DTABLING to YAP_EXTRAS in the system’s Makefile. We next describe the set of built-ins predicates designed to interact with YAPTab and control tabled execution:

`table +P` Declares predicate `P` (or a list of predicates `P1,...,Pn` or `[P1,...,Pn]`) as a tabled predicate. `P` must be written in the form `name/arity`. Examples:

```
:- table son/3.
:- table father/2.
:- table mother/2.
```

or

```
:- table son/3, father/2, mother/2.
```

or

```
:- table [son/3, father/2, mother/2].
```

`is_tabled(+P)` Succeeds if the predicate `P` (or a list of predicates `P1,...,Pn` or `[P1,...,Pn]`), of the form `name/arity`, is a tabled predicate.

`tabling_mode(+P,?Mode)` Sets or reads the default tabling mode for a tabled predicate `P` (or a list of predicates `P1,...,Pn` or `[P1,...,Pn]`). The list of `Mode` options includes:

- **batched** Defines that, by default, batched scheduling is the scheduling strategy to be used to evaluated calls to predicate `P`.
- **local** Defines that, by default, local scheduling is the scheduling strategy to be used to evaluated calls to predicate `P`.
- **exec_answers** Defines that, by default, when a call to predicate `P` is already evaluated (completed), answers are obtained by executing compiled WAM-like code directly from the trie data structure. This reduces the loading time when backtracking, but the order in which answers are obtained is undefined.
load_answers
Defines that, by default, when a call to predicate $P$ is already evaluated (completed), answers are obtained (as a consumer) by loading them from the trie data structure. This guarantees that answers are obtained in the same order as they were found. Somewhat less efficient but creates less choice-points.

The default tabling mode for a new tabled predicate is batched and exec_answers. To set the tabling mode for all predicates at once you can use the yap_flag/2 predicate as described next.

yap_flag(tabling_mode,?Mode)
Sets or reads the tabling mode for all tabled predicates. The list of $Mode$ options includes:

default
Defines that (i) all calls to tabled predicates are evaluated using the predicate default mode, and that (ii) answers for all completed calls are obtained using the predicate default mode.

batched
Defines that all calls to tabled predicates are evaluated using batched scheduling. This option ignores the default tabling mode of each predicate.

local
Defines that all calls to tabled predicates are evaluated using local scheduling. This option ignores the default tabling mode of each predicate.

eexec_answers
Defines that answers for all completed calls are obtained by executing compiled WAM-like code directly from the trie data structure. This option ignores the default tabling mode of each predicate.

load_answers
Defines that answers for all completed calls are obtained by loading them from the trie data structure. This option ignores the default tabling mode of each predicate.

abolish_table(+P)
Removes all the entries from the table space for predicate $P$ (or a list of predicates $P_1,\ldots,P_n$ or $[P_1,\ldots,P_n]$). The predicate remains as a tabled predicate.

abolish_all_tables/0
Removes all the entries from the table space for all tabled predicates. The predicates remain as tabled predicates.

show_table(+P)
Prints table contents (subgoals and answers) for predicate $P$ (or a list of predicates $P_1,\ldots,P_n$ or $[P_1,\ldots,P_n]$).

table_statistics(+P)
Prints table statistics (subgoals and answers) for predicate $P$ (or a list of predicates $P_1,\ldots,P_n$ or $[P_1,\ldots,P_n]$).

tabling_statistics/0
Prints statistics on space used by all tables.
18 Tracing at Low Level

It is possible to follow the flow at abstract machine level if YAP is compiled with the flag LOW_LEVEL_TRACER. Note that this option is of most interest to implementers, as it quickly generates an huge amount of information.

Low level tracing can be toggled from an interrupt handler by using the option T. There are also two built-ins that activate and deactivate low level tracing:

start_low_level_trace
    Begin display of messages at procedure entry and retry.

stop_low_level_trace
    Stop display of messages at procedure entry and retry.

Note that this compile-time option will slow down execution.
19 Profiling the Abstract Machine

Implementors may be interested in detecting on which abstract machine instructions are executed by a program. The ANALYST flag can give WAM level information. Note that this option slows down execution very substantially, and is only of interest to developers of the system internals, or to system debuggers.

`reset_op_counters`
Reinitialize all counters.

`show_op_counters(+A)`
Display the current value for the counters, using label A. The label must be an atom.

`show_ops_by_group(+A)`
Display the current value for the counters, organized by groups, using label A. The label must be an atom.
20 Debugging

20.1 Debugging Predicates

The following predicates are available to control the debugging of programs:

**debug** Switches the debugger on.

**debugging** Outputs status information about the debugger which includes the leash mode and the existing spy-points, when the debugger is on.

**nodebug** Switches the debugger off.

**spy +P** Sets spy-points on all the predicates represented by \( P \). \( P \) can either be a single specification or a list of specifications. Each one must be of the form \( Name/Arity \) or \( Name \). In the last case all predicates with the name \( Name \) will be spied. As in C-Prolog, system predicates and predicates written in C, cannot be spied.

**nospy +P** Removes spy-points from all predicates specified by \( P \). The possible forms for \( P \) are the same as in **spy P**.

**nospyall** Removes all existing spy-points.

**notrace** Switches off the debugger and stops tracing.

**leash(+M)** Sets leashing mode to \( M \). The mode can be specified as:

- **full** prompt on Call, Exit, Redo and Fail
- **tight** prompt on Call, Redo and Fail
- **half** prompt on Call and Redo
- **loose** prompt on Call
- **off** never prompt
- **none** never prompt, same as **off**

The initial leashing mode is **full**.

The user may also specify directly the debugger ports where he wants to be prompted. If the argument for leash is a number \( N \), each of lower four bits of the number is used to control prompting at one the ports of the box model. The debugger will prompt according to the following conditions:

- if \( N/\ 1 \div= 0 \) prompt on fail
- if \( N/\ 2 \div= 0 \) prompt on redo
- if \( N/\ 4 \div= 0 \) prompt on exit
- if \( N/\ 8 \div= 0 \) prompt on call
Therefore, `leash(15)` is equivalent to `leash(full)` and `leash(0)` is equivalent to `leash(off)`.

Another way of using `leash` is to give it a list with the names of the ports where the debugger should stop. For example, `leash([call, exit, redo, fail])` is the same as `leash(full)` or `leash(15)` and `leash([fail])` might be used instead of `leash(1).

\[\text{spy\_write}(\textbf{+} \text{Stream}, \textbf{Term})\]

If defined by the user, this predicate will be used to print goals by the debugger instead of `write/2`.

\[\text{trace}\]

Switches on the debugger and starts tracing.

\[\text{notrace}\]

Ends tracing and exits the debugger. This is the same as `nodebug/0`.

## 20.2 Interacting with the debugger

Debugging with YAP is similar to debugging with C-Prolog. Both systems include a procedural debugger, based on Byrd’s four port model. In this model, execution is seen at the procedure level: each activation of a procedure is seen as a box with control flowing into and out of that box.

In the four port model control is caught at four key points: before entering the procedure, after exiting the procedure (meaning successful evaluation of all queries activated by the procedure), after backtracking but before trying new alternative to the procedure and after failing the procedure. Each one of these points is named a port:

\[
\begin{align*}
\text{Call} & \quad \text{Exit} \\
\downarrow & \quad \downarrow \\
\text{descendant}(X,Y) :- \text{offspring}(X,Y). & \text{descendant}(X,Z) :- \\
\text{offspring}(X,Y), \text{descendant}(Y,Z). & \text{offspring}(X,Y), \text{descendant}(Y,Z).
\end{align*}
\]

\[\text{Call}\]

The call port is activated before initial invocation of procedure. Afterwards, execution will try to match the goal with the head of existing clauses for the procedure.

\[\text{Exit}\]

This port is activated if the procedure succeeds. Control will now leave the procedure and return to its ancestor.

\[\text{Redo}\]

if the goal, or goals, activated after the call port fail then backtracking will eventually return control to this procedure through the redo port.

\[\text{Fail}\]

If all clauses for this predicate fail, then the invocation fails, and control will try to redo the ancestor of this invocation.

To start debugging, the user will either call `trace` or spy the relevant procedures, entering debug mode, and start execution of the program. When finding the first spy-point, YAP’s debugger will take control and show a message of the form:

\[
* (1) \text{call: quicksort([1,2,3],_38) ?}
\]

The debugger message will be shown while creeping, or at spy-points, and it includes four or five fields:
• The first three characters are used to point out special states of the debugger. If the port is exit and the first character is '?', the current call is non-deterministic, that is, it still has alternatives to be tried. If the second character is a *, execution is at a spy-point. If the third character is a >, execution has returned either from a skip, a fail or a redo command.

• The second field is the activation number, and uniquely identifies the activation. The number will start from 1 and will be incremented for each activation found by the debugger.

• In the third field, the debugger shows the active port.

• The fourth field is the goal. The goal is written by write_term/3 on the standard error stream, using the options given by debugger_print_options.

If the active port is leashed, the debugger will prompt the user with a ?, and wait for a command. A debugger command is just a character, followed by a return. By default, only the call and redo entries are leashed, but the leash/1 predicate can be used in order to make the debugger stop where needed.

There are several commands available, but the user only needs to remember the help command, which is h. This command shows all the available options, which are:

c - creep this command makes YAP continue execution and stop at the next leashed port.

return - creeping
the same as c

l - leap YAP will execute until it meets a port for a spied predicate; this mode keeps all computation history for debugging purposes, so it is more expensive than standard execution. Use k or z for fast execution.

k - quasi-leap
similar to leap but faster since the computation history is not kept; useful when leap becomes too slow.

z - zip same as k

s - skip YAP will continue execution without showing any messages until returning to the current activation. Spy-points will be ignored in this mode. Note that this command keeps all debugging history, use t for fast execution. This command is meaningless, and therefore illegal, in the fail and exit ports.

t - fast-skip
similar to skip but faster since computation history is not kept; useful if skip becomes slow.

f [GoalId] - fail
If given no argument, forces YAP to fail the goal, skipping the fail port and backtracking to the parent. If f receives a goal number as the argument, the command fails all the way to the goal. If goal GoalId has completed execution, YAP fails until meeting the first active ancestor.

r [GoalId] - retry
This command forces YAP to jump back call to the port. Note that any side effects of the goal cannot be undone. This command is not available at the
call port. If \( f \) receives a goal number as the argument, the command retries goal \( \text{GoalId} \) instead. If goal \( \text{GoalId} \) has completed execution, YAP fails until meeting the first active ancestor.

**a** - abort
execution will be aborted, and the interpreter will return to the top-level. YAP disactivates debug mode, but spy-points are not removed.

**n** - nodebug
stop debugging and continue execution. The command will not clear active spy-points.

**e** - exit
leave YAP.

**h** - help
show the debugger commands.

! Query
execute a query. YAP will not show the result of the query.

**b** - break
break active execution and launch a break level. This is the same as ! break.

++ spy this goal
start spying the active goal. The same as ! spy \( G \) where \( G \) is the active goal.

-- nospy this goal
stop spying the active goal. The same as ! nospy \( G \) where \( G \) is the active goal.

**p** - print
shows the active goal using print/1

**d** - display
shows the active goal using display/1

<Depth - debugger write depth
sets the maximum write depth, both for composite terms and lists, that will be used by the debugger. For more information about write_depth/2 (see Section 6.9.7 [I/O Control], page 68).

<- full term
resets to the default of ten the debugger’s maximum write depth. For more information about write_depth/2 (see Section 6.9.7 [I/O Control], page 68).

**A** - alternatives
show the list of backtrack points in the current execution.

**g** \([N]\)
show the list of ancestors in the current debugging environment. If it receives \( N \), show the first \( N \) ancestors.

The debugging information, when fast-skip quasi-leap is used, will be lost.
21 Efficiency Considerations

We next discuss several issues on trying to make Prolog programs run fast in YAP. We assume two different programming styles:

- Execution of deterministic programs often boils down to a recursive loop of the form:

  \[
  \text{loop(Env)} : - \\
  \quad \text{do_something(Env,NewEnv)}, \\
  \quad \text{loop(NewEnv)}.
  \]

21.1 Deterministic Programs

21.2 Non-Deterministic Programs

21.3 Data-Base Operations

21.4 Indexing

21.5 Profiling

The indexation mechanism restricts the set of clauses to be tried in a procedure by using information about the status of a selected argument of the goal (in YAP, as in most compilers, the first argument). This argument is then used as a key, selecting a restricted set of clauses from all the clauses forming the procedure.

As an example, the two clauses for concatenate:

\[
\text{concatenate([],L,L).} \\
\text{concatenate([H|T],A,[H|NT]) :- concatenate(T,A,NT).}
\]

If the first argument for the goal is a list, then only the second clause is of interest. If the first argument is the nil atom, the system needs to look only for the first clause. The indexation generates instructions that test the value of the first argument, and then proceed to a selected clause, or group of clauses.

Note that if the first argument was a free variable, then both clauses should be tried. In general, indexation will not be useful if the first argument is a free variable.

When activating a predicate, a Prolog system needs to store state information. This information, stored in a structure known as choice point or fail point, is necessary when backtracking to other clauses for the predicate. The operations of creating and using a choice point are very expensive, both in the terms of space used and time spent. Creating a choice point is not necessary if there is only a clause for the predicate as there are no clauses to backtrack to. With indexation, this situation is extended: in the example, if the first argument was the atom nil, then only one clause would really be of interest, and it is pointless to create a choice point. This feature is even more useful if the first argument is a list: without indexation, execution would try the first clause, creating a choice point. The clause would fail, the choice point would then be used to restore the previous state of the computation and the second clause would be tried. The code generated by the indexation
mechanism would behave much more efficiently: it would test the first argument and see
whether it is a list, and then proceed directly to the second clause.

An important side effect concerns the use of "cut". In the above example, some pro-
grammers would use a "cut" in the first clause just to inform the system that the predicate
is not backtrackable and force the removal the choice point just created. As a result, less
space is needed but with a great loss in expressive power: the "cut" would prevent some
uses of the procedure, like generating lists through backtracking. Of course, with indexation
the "cut" becomes useless: the choice point is not even created.

Indexation is also very important for predicates with a large number of clauses that are
used like tables:

```
logician(aristoteles,greek).
logician(frege,german).
logician(russel,english).
logician(godel,german).
logician(whitehead,english).
```

An interpreter like C-Prolog, trying to answer the query:
```
?- logician(godel,X).
```
would blindly follow the standard Prolog strategy, trying first the first clause, then the
second, the third and finally finding the relevant clause. Also, as there are some more
clauses after the important one, a choice point has to be created, even if we know the next
clauses will certainly fail. A "cut" would be needed to prevent some possible uses for the
procedure, like generating all logicians. In this situation, the indexing mechanism generates
instructions that implement a search table. In this table, the value of the first argument
would be used as a key for fast search of possibly matching clauses. For the query of the
last example, the result of the search would be just the fourth clause, and again there would
be no need for a choice point.

If the first argument is a complex term, indexation will select clauses just by testing its
main functor. However, there is an important exception: if the first argument of a clause
is a list, the algorithm also uses the list's head if not a variable. For instance, with the
following clauses,
```
rules([],B,B).
rules([n(N)|T],I,O) :- rules_for_noun(N,I,N), rules(T,N,O).
rules([v(V)|T],I,O) :- rules_for_verb(V,I,N), rules(T,N,O).
rules([q(Q)|T],I,O) :- rules_for_qualifier(Q,I,N), rules(T,N,O).
```
if the first argument of the goal is a list, its head will be tested, and only the clauses
matching it will be tried during execution.

Some advice on how to take a good advantage of this mechanism:

- Try to make the first argument an input argument.
- Try to keep together all clauses whose first argument is not a variable, that will decrease
  the number of tests since the other clauses are always tried.
- Try to avoid predicates having a lot of clauses with the same key. For instance, the
  procedure:
```
type(n(mary),person).
type(n(john), person).
```
type(n(chair), object).
type(v(eat), active).
type(v(rest), passive).
becomes more efficient with:

\[
\begin{align*}
\text{type}(n(N), T) & : \text{type_of_noun}(N, T). \\
\text{type}(v(V), T) & : \text{type_of_verb}(V, T).
\end{align*}
\]

\[
\begin{align*}
\text{type_of_noun}(mary, \text{person}). \\
\text{type_of_noun}(john, \text{person}). \\
\text{type_of_noun}(chair, \text{object}). \\
\text{type_of_verb}(eat, \text{active}). \\
\text{type_of_verb}(rest, \text{passive}).
\end{align*}
\]
Chapter 22: C Language interface to YAP

22 C Language interface to YAP

YAP provides the user with the necessary facilities for writing predicates in a language other than Prolog. Since, under Unix systems, most language implementations are link-able to C, we will describe here only the YAP interface to the C language.

Before describing in full detail how to interface to C code, we will examine a brief example.

Assume the user requires a predicate `my_process_id(Id)` which succeeds when `Id` unifies with the number of the process under which YAP is running.

In this case we will create a `my_process.c` file containing the C-code described below.

```c
#include "YAP/YAPInterface.h"

static int my_process_id(void)
{
    YAP_Term pid = YAP_MkIntTerm(getpid());
    YAP_Term out = YAP_ARG1;
    return(YAP_Unify(out,pid));
}

void init_my_predicates()
{
    YAP_UserCPredicate("my_process_id",my_process_id,1);
}
```

The commands to compile the above file depend on the operating system. Under Linux (i386 and Alpha) you should use:

```
 gcc -c -shared -fPIC my_process.c
 ld -shared -o my_process.so my_process.o
```

Under WIN32 in a MINGW/CYGWIN environment, using the standard installation path you should use:

```
 gcc -mno-cygwin -I "c:/Yap/include" -c my_process.c
 gcc -mno-cygwin "c:/Yap/bin/yap.dll" --shared -o my_process.dll my_process.o
```

Under WIN32 in a pure CYGWIN environment, using the standard installation path, you should use:

```
 gcc -I/usr/local -c my_process.c
 gcc -shared -o my_process.dll my_process.o /usr/local/bin/yap.dll
```

Under Solaris2 it is sufficient to use:

```
 gcc -fPIC -c my_process.c
```

Under SunOS it is sufficient to use:

```
 gcc -c my_process.c
```

Under Digital Unix you need to create a `so` file. Use:
gcc tst.c -c -fpic
ld my_process.o -o my_process.so -shared -expect_unresolved '*'

and replace my process.so for my process.o in the remainder of the example. And could be loaded, under YAP, by executing the following Prolog goal

load_foreign_files([‘my_process’], [], init_my_predicates).

Note that since YAP4.3.3 you should not give the suffix for object files. YAP will deduce the correct suffix from the operating system it is running under.

After loading that file the following Prolog goal

my_process_id(N)

would unify N with the number of the process under which YAP is running.

Having presented a full example, we will now examine in more detail the contents of the C source code file presented above.

The include statement is used to make available to the C source code the macros for the handling of Prolog terms and also some YAP public definitions.

The function my_process_id is the implementation, in C, of the desired predicate. Note that it returns an integer denoting the success of failure of the goal and also that it has no arguments even though the predicate being defined has one. In fact the arguments of a Prolog predicate written in C are accessed through macros, defined in the include file, with names YAP_ARG1, YAP_ARG2, ..., YAP_ARG16 or with YAP_A(N) where N is the argument number (starting with 1). In the present case the function uses just one local variable of type YAP_Term, the type used for holding YAP terms, where the integer returned by the standard unix function getpid() is stored as an integer term (the conversion is done by YAP_MkIntTerm(Int)). Then it calls the pre-defined routine YAP_Unify(YAP_Term, YAP_Term) which in turn returns an integer denoting success or failure of the unification.

The role of the procedure init_my_predicates is to make known to YAP, by calling YAP_UserCPredicate, the predicates being defined in the file. This is in fact why, in the example above, init_my_predicates was passed as the third argument to load_foreign_files.

The rest of this appendix describes exhaustively how to interface C to YAP.

## 22.1 Terms

This section provides information about the primitives available to the C programmer for manipulating Prolog terms.

Several C typedefs are included in the header file yap/YAPInterface.h to describe, in a portable way, the C representation of Prolog terms. The user should write is programs using this macros to ensure portability of code across different versions of YAP.

The more important typedef is YAP_Term which is used to denote the type of a Prolog term.

Terms, from a point of view of the C-programmer, can be classified as follows

uninstantiated variables
instantiated variables
integers
floating-point numbers
database references
atoms
pairs (lists)
compound terms

The primitive

\[ \text{YAP}_\text{Bool} \text{ YAP\_IsVarTerm(YAP\_Term t)} \]
returns true iff its argument is an uninstantiated variable. Conversely the primitive

\[ \text{YAP}_\text{Bool} \text{ YAP\_NonVarTerm(YAP\_Term t)} \]
returns true iff its argument is not a variable.

The user can create a new uninstantiated variable using the primitive

\[ \text{YAP\_Term} \quad \text{YAP\_MkVarTerm()} \]

The following primitives can be used to discriminate among the different types of non-variable terms:

\[ \text{YAP}_\text{Bool} \text{ YAP\_IsIntTerm(YAP\_Term t)} \]
\[ \text{YAP}_\text{Bool} \text{ YAP\_IsFloatTerm(YAP\_Term t)} \]
\[ \text{YAP}_\text{Bool} \text{ YAP\_IsDbRefTerm(YAP\_Term t)} \]
\[ \text{YAP}_\text{Bool} \text{ YAP\_IsAtomTerm(YAP\_Term t)} \]
\[ \text{YAP}_\text{Bool} \text{ YAP\_IsPairTerm(YAP\_Term t)} \]
\[ \text{YAP}_\text{Bool} \text{ YAP\_IsApplTerm(YAP\_Term t)} \]

Next, we mention the primitives that allow one to destruct and construct terms. All the above primitives ensure that their result is dereferenced, i.e. that it is not a pointer to another term.

The following primitives are provided for creating an integer term from an integer and to access the value of an integer term.

\[ \text{YAP\_Term} \quad \text{YAP\_MkIntTerm(YAP\_Int i)} \]
\[ \text{YAP\_Int} \quad \text{YAP\_IntOfTerm(YAP\_Term t)} \]

where YAP\_Int is a typedef for the C integer type appropriate for the machine or compiler in question (normally a long integer). The size of the allowed integers is implementation dependent but is always greater or equal to 24 bits: usually 32 bits on 32 bit machines, and 64 on 64 bit machines.

The two following primitives play a similar role for floating-point terms

\[ \text{YAP\_Term} \quad \text{YAP\_MkFloatTerm(YAP\_flt double)} \]
\[ \text{YAP\_flt} \quad \text{YAP\_FloatOfTerm(YAP\_Term t)} \]

where flt is a typedef for the appropriate C floating point type, nowadays a double

The following primitives are provided for verifying whether a term is a big int, creating a term from a big integer and to access the value of a big int from a term.

\[ \text{YAP\_Bool} \text{ YAP\_IsBigNumTerm(YAP\_Term t)} \]
\[ \text{YAP\_Term} \quad \text{YAP\_MkBigNumTerm(void *b)} \]
\[ \text{void *} \quad \text{YAP\_BigNumOfTerm(YAP\_Term t, void *b)} \]

YAP must support bignum for the configuration you are using (check the YAP configuration and setup). For now, YAP only supports the GNU GMP library, and void * will be a cast
for mpz_t. Notice that YAP_BigNumOfTerm requires the number to be already initialised. As an example, we show how to print a bignum:

```c
static int
p_print_bignum(void)
{
    mpz_t mz;
    if (!YAP_IsBigNumTerm(YAP_ARG1))
        return FALSE;

    mpz_init(mz);
    YAP_BigNumOfTerm(YAP_ARG1, mz);
    gmp_printf("Shows up as %Zd\n", mz);
    mpz_clear(mz);
    return TRUE;
}
```

Currently, no primitives are supplied to users for manipulating data base references.

A special typedef YAP_Atom is provided to describe Prolog atoms (symbolic constants). The two following primitives can be used to manipulate atom terms

```c
YAP_Term YAP_MkAtomTerm(YAP_Atom at)
YAP_Atom YAP_AtomOfTerm(YAP_Term t)
```

The following primitives are available for associating atoms with their names

```c
YAP_Atom YAP_LookupAtom(char * s)
YAP_Atom YAP_FullLookupAtom(char * s)
char *YAP_AtomName(YAP_Atom t)
```

The function YAP_LookupAtom looks up an atom in the standard hash table. The function YAP_FullLookupAtom will also search if the atom had been "hidden": this is useful for system maintenance from C code. The functor YAP_AtomName returns a pointer to the string for the atom.

The following primitives handle constructing atoms from strings with wide characters, and vice-versa:

```c
YAP_Atom YAP_LookupWideAtom(wchar_t * s)
wchar_t *YAP_WideAtomName(YAP_Atom t)
```

The following primitive tells whether an atom needs wide atoms in its representation:

```c
int YAP_IsWideAtom(YAP_Atom t)
```

The following primitive can be used to obtain the size of an atom in a representation-independent way:

```c
int YAP_AtomNameLength(YAP_Atom t)
```

The next routines give users some control over the atom garbage collector. They allow the user to guarantee that an atom is not to be garbage collected (this is important if the atom is hold externally to the Prolog engine, allow it to be collected, and call a hook on garbage collection:

```c
int YAP_AtomGetHold(YAP_Atom at)
int YAP_AtomReleaseHold(YAP_Atom at)
```
A pair is a Prolog term which consists of a tuple of two Prolog terms designated as the head and the tail of the term. Pairs are most often used to build lists. The following primitives can be used to manipulate pairs:

- YAP_Term YAP_MkPairTerm(YAP_Term Head, YAP_Term Tail)
- YAP_Term YAP_MkNewPairTerm(void)
- YAP_Term YAP_HeadOfTerm(YAP_Term t)
- YAP_Term YAP_TailOfTerm(YAP_Term t)

One can construct a new pair from two terms, or one can just build a pair whose head and tail are new unbound variables. Finally, one can fetch the head or the tail.

A compound term consists of a functor and a sequence of terms with length equal to the arity of the functor. A functor, described in C by the typedef Functor, consists of an atom and an integer. The following primitives were designed to manipulate compound terms and functors:

- YAP_Term YAP_MkApplTerm(YAP_Functor f, unsigned long int n, YAP_Term[] args)
- YAP_Term YAP_MkNewApplTerm(YAP_Functor f, int n)
- YAP_Term YAP_ArgOfTerm(int argno, YAP_Term ts)
- YAP_Term *YAP_ArgsOfTerm(YAP_Term ts)
- YAP_Functor YAP_FunctorOfTerm(YAP_Term ts)

The YAP_MkApplTerm function constructs a new term, with functor f (of arity n), and using an array args of n terms with n equal to the arity of the functor. YAP_MkNewApplTerm builds up a compound term whose arguments are unbound variables. YAP_ArgOfTerm gives an argument to a compound term. argno should be greater or equal to 1 and less or equal to the arity of the functor. YAP_ArgsOfTerm returns a pointer to an array of arguments.

YAP allows one to manipulate the functors of compound term. The function YAP_FunctorOfTerm allows one to obtain a variable of type YAP_Functor with the functor to a term. The following functions then allow one to construct functors, and to obtain their name and arity.

- YAP_Functor YAP_MkFunctor(YAP_Atom a, unsigned long int arity)
- YAP_Atom YAP_NameOfFunctor(YAP_Functor f)
- YAP_Int YAP_ArityOfFunctor(YAP_Functor f)

Note that the functor is essentially a pair formed by an atom, and arity.

### 22.2 Unification

YAP provides a single routine to attempt the unification of two Prolog terms. The routine may succeed or fail:

- Int YAP_Unify(YAP_Term a, YAP_Term b)

The routine attempts to unify the terms a and b returning TRUE if the unification succeeds and FALSE otherwise.
22.3 Strings

The YAP C-interface now includes an utility routine to copy a string represented as a list of a character codes to a previously allocated buffer

```c
int YAP_StringToBuffer(YAP_Term String, char *buf, unsigned int bufsize)
```

The routine copies the list of character codes `String` to a previously allocated buffer `buf`. The string including a terminating null character must fit in `bufsize` characters, otherwise the routine will simply fail. The `StringToBuffer` routine fails and generates an exception if `String` is not a valid string.

The C-interface also includes utility routines to do the reverse, that is, to copy a from a buffer to a list of character codes, to a difference list, or to a list of character atoms. The routines work either on strings of characters or strings of wide characters:

```c
YAP_Term YAP_BufferToString(char *buf)
YAP_Term YAP_NBufferToString(char *buf, size_t len)
YAP_Term YAP_WideBufferToString(wchar_t *buf)
YAP_Term YAP_NWideBufferToString(wchar_t *buf, size_t len)
YAP_Term YAP_BufferToAtomList(char *buf)
YAP_Term YAP_NBufferToAtomList(char *buf, size_t len)
YAP_Term YAP_WideBufferToAtomList(wchar_t *buf)
YAP_Term YAP_NWideBufferToAtomList(wchar_t *buf, size_t len)
```

Users are advised to use the `N` version of the routines. Otherwise, the user-provided string must include a terminating null character.

The C-interface function calls the parser on a sequence of characters stored at `buf` and returns the resulting term.

```c
YAP_Term YAP_ReadBuffer(char *buf, YAP_Term *error)
```

The user-provided string must include a terminating null character. Syntax errors will cause returning `FALSE` and binding `error` to a Prolog term.

22.4 Memory Allocation

The next routine can be used to ask space from the Prolog data-base:

```c
void *YAP_AllocSpaceFromYAP(int size)
```

The routine returns a pointer to a buffer allocated from the code area, or `NULL` if sufficient space was not available.

The space allocated with `YAP_AllocSpaceFromYAP` can be released back to YAP by using:

```c
void YAP_FreeSpaceFromYAP(void *buf)
```

The routine releases a buffer allocated from the code area. The system may crash if `buf` is not a valid pointer to a buffer in the code area.

22.5 Controlling YAP Streams from C

The C-Interface also provides the C-application with a measure of control over the YAP Input/Output system. The first routine allows one to find a file number given a current stream:
int YAP_StreamToFileNo(YAP_Term stream)

This function gives the file descriptor for a currently available stream. Note that null streams and in memory streams do not have corresponding open streams, so the routine will return a negative. Moreover, YAP will not be aware of any direct operations on this stream, so information on, say, current stream position, may become stale.

A second routine that is sometimes useful is:

void YAP_CloseAllOpenStreams(void)

This routine closes the YAP Input/Output system except for the first three streams, that are always associated with the three standard Unix streams. It is most useful if you are doing fork().

Last, one may sometimes need to flush all streams:

void YAP_CloseAllOpenStreams(void)

It is also useful before you do a fork(), or otherwise you may have trouble with unflushed output.

The next routine allows a currently open file to become a stream. The routine receives as arguments a file descriptor, the true file name as a string, an atom with the user name, and a set of flags:

void YAP_OpenStream(void *FD, char *name, YAP_Term t, int flags)

The available flags are YAP_INPUT_STREAM, YAP_OUTPUT_STREAM, YAP_APPEND_STREAM, YAP_PIPE_STREAM, YAP_TTY_STREAM, YAP_POPEN_STREAM, YAP_BINARY_STREAM, and YAP_SEEKABLE_STREAM. By default, the stream is supposed to be at position 0. The argument name gives the name by which YAP should know the new stream.

22.6 Utility Functions in C

The C-Interface provides the C-application with a number of utility functions that are useful.

The first provides a way to insert a term into the data-base

void *YAP_Record(YAP_Term t)

This function returns a pointer to a copy of the term in the database (or to NULL if the operation fails.

The next functions provides a way to recover the term from the data-base:

YAP_Term YAP_Recorded(void *handle)

Notice that the semantics are the same as for recorded/3: this function creates a new copy of the term in the stack, with fresh variables. The function returns 0L if it cannot create a new term.

Last, the next function allows one to recover space:

int YAP_Erase(void *handle)

Notice that any accesses using handle after this operation may lead to a crash.

The following functions are often required to compare terms.

The first function succeeds if two terms are actually the same term, as ==/2:
int YAP_ExactlyEqual(YAP_Term t1, YAP_Term t2)

The second function succeeds if two terms are variant terms, and returns 0 otherwise, as \( = / 2 \):

int YAP_Variant(YAP_Term t1, YAP_Term t2)

The second function computes a hash function for a term, as in \( \text{term_hash}/4 \).

\[ \text{YAP_Int YAP_TermHash(YAP_Term t, YAP_Int range, YAP_Int depth, int ignore_variables);} \]

The first three arguments follow \( \text{term_has}/4 \). The last argument indicates what to do if we find a variable: if 0 fail, otherwise ignore the variable.

22.7 From C back to Prolog

There are several ways to call Prolog code from C-code. By default, the \( \text{YAP_RunGoal()} \) should be used for this task. It assumes the engine has been initialised before:

\[ \text{YAP_RunGoal(YAP_Term Goal)} \]

Execute query \( \text{Goal} \) and return 1 if the query succeeds, and 0 otherwise. The predicate returns 0 if failure, otherwise it will return an \( \text{YAP_Term} \).

Quite often, one wants to run a query once. In this case you should use \( \text{Goal} \):

\[ \text{YAP_RunGoalOnce(YAP_Term Goal)} \]

The \( \text{YAP_RunGoal()} \) function makes sure to recover stack space at the end of execution.

Prolog terms are pointers: a problem users often find is that the term \( \text{Goal} \) may actually be moved around during the execution of \( \text{YAP_RunGoal()} \), due to garbage collection or stack shifting. If this is possible, \( \text{Goal} \) will become invalid after executing \( \text{YAP_RunGoal()} \). In this case, it is a good idea to save \( \text{Goal slots} \), as shown next:

\[ \text{long sl = YAP_InitSlot(scoreTerm);} \]

\[ \text{out = YAP_RunGoal(t);} \]
\[ \text{t = YAP_GetFromSlot(sl);} \]
\[ \text{YAP_RecoverSlots(1);} \]
\[ \text{if (out == 0) return FALSE;} \]

Slots are safe houses in the stack, the garbage collector and the stack shifter know about them and make sure they have correct values. In this case, we use a slot to preserve \( t \) during the execution of \( \text{YAP_RunGoal()} \). When the execution of \( t \) is over we read the (possibly changed) value of \( t \) back from the slot \( sl \) and tell YAP that the slot \( sl \) is not needed and can be given back to the system. The slot functions are as follows:

\[ \text{YAP_Int YAP_NewSlots(int NumberOfSlots)} \]

Allocate \( \text{NumberOfSlots} \) from the stack and return an handle to the last one. The other handle can be obtained by decrementing the handle.

\[ \text{YAP_Int YAP_CurrentSlot(\text{void})} \]

Return a handle to the system’s default slot.

\[ \text{long YAP_InitSlot(YAP_Term t)} \]

Create a new slot, initialise it with \( t \), and return a handle to this slot, that also becomes the current slot.
YAP_Term *YAP_AddressFromSlot(YAP_Int slot)
Return the address of slot slot: please use with care.

void YAP_PutInSlot(YAP_Int slot, YAP_Term t)
Set the contents of slot slot to t.

int YAP_RecoverSlots(int HowMany)
Recover the space for HowMany slots: these will include the current default slot. Fails if no such slots exist.

int YAP_ArgsToSlots(int HowMany)
Store the current first HowMany arguments in new slots.

int YAP_SlotsToArgs(int HowMany, YAP_Int slot)
Set the first HowMany arguments to the HowMany slots starting at slot.

The following functions complement YAP_RunGoal:

Look for the next solution to the current query by forcing YAP to backtrack to the latest goal. Notice that slots allocated since the last YAP_RunGoal will become invalid.

int YAP_Reset(void)
Reset execution environment (similar to the abort/0 built-in). This is useful when you want to start a new query before asking all solutions to the previous query.

int YAP_ShutdownGoal(int backtrack)
Clean up the current goal. If backtrack is true, stack space will be recovered and bindings will be undone. In both cases, any slots allocated since the goal was created will become invalid.

YAP_Bool YAP_GoalHasException(YAP_Term *tp)
Check if the last goal generated an exception, and if so copy it to the space pointed to by tp.

void YAP_ClearExceptions(void)
Reset any exceptions left over by the system.

The YAP_RunGoal interface is designed to be very robust, but may not be the most efficient when repeated calls to the same goal are made and when there is no interest in processing exception. The YAP_EnterGoal interface should have lower-overhead:

YAP_PredEntryPtr YAP_FunctorToPred(YAP_Functor f)
Return the predicate whose main functor is f.

YAP_PredEntryPtr YAP_AtomToPred(YAP_Atom at)
Return the arity 0 predicate whose name is at.

YAP_Bool YAP_EnterGoal(YAP_PredEntryPtr pe,
YAP_Term *array, YAP_dogoalinfo *infop)
Execute a query for predicate pe. The query is given as an array of terms array. infop is the address of a goal handle that can be used to backtrack and to recover space. Succeeds if a solution was found.
Notice that you cannot create new slots if an YAP_EnterGoal goal is open.
YAP_Bool YAP_RetryGoal(YAP_dogoalinfo * infop)
   Backtrack to a query created by YAP_EnterGoal. The query is given by the
   handle infop. Returns whether a new solution could be be found.

YAP_Bool YAP_LeaveGoal(YAP_Bool backtrack,
    YAP_dogoalinfo * infop) Exit a query query created by YAP_EnterGoal. If
    backtrack is TRUE, variable bindings are undone and Heap space is recovered.
    Otherwise, only stack space is recovered, ie, LeaveGoal executes a cut.

Next, follows an example of how to use YAP_EnterGoal:

    void
    runall(YAP_Term g)
    {
        YAP_dogoalinfo goalInfo;
        YAP_Term *goalArgs = YAP_ArraysOfTerm(g);
        YAP_Functor *goalFunctor = YAP_FunctorOfTerm(g);
        YAP_PredEntryPtr goalPred = YAP_FunctorToPred(goalFunctor);

        result = YAP_EnterGoal( goalPred, goalArgs, &goalInfo );
        while (result)
            result = YAP_RetryGoal( &goalInfo );
        YAP_LeaveGoal(TRUE, &goalInfo);
    }

YAP allows calling a new Prolog interpreter from C. One way is to first construct a goal
G, and then it is sufficient to perform:

    YAP_Bool YAP_CallProlog(YAP_Term G)
    the result will be FALSE, if the goal failed, or TRUE, if the goal succeeded. In this case, the
    variables in G will store the values they have been unified with. Execution only proceeds
    until finding the first solution to the goal, but you can call findall/3 or friends if you need
    all the solutions.

    Notice that during execution, garbage collection or stack shifting may have moved the
    terms

22.8 Module Manipulation in C

YAP allows one to create a new module from C-code. To create the new code it is sufficient
to call:

    YAP_Module YAP_CreateModule(YAP_Atom ModuleName)
    Notice that the new module does not have any predicates associated and that it is not the
    current module. To find the current module, you can call:

    YAP_Module YAP_CurrentModule()

    Given a module, you may want to obtain the corresponding name. This is possible by
    using:

    YAP_Term YAP_ModuleName(YAP_Module mod)
    Notice that this function returns a term, and not an atom. You can YAP_AtomOfTerm to
    extract the corresponding Prolog atom.
22.9 Writing predicates in C

We will distinguish two kinds of predicates:

- **Deterministic predicates** which either fail or succeed but are not backtrackable, like the one in the introduction;

- **Backtrackable** predicates which can succeed more than once.

The first kind of predicates should be implemented as a C function with no arguments which should return zero if the predicate fails and a non-zero value otherwise. The predicate should be declared to YAP, in the initialization routine, with a call to

```c
void YAP_UserCPredicate(char *name, YAP_Bool *fn(), unsigned long int arity);
```

where `name` is the name of the predicate, `fn` is the C function implementing the predicate and `arity` is its arity.

For the second kind of predicates we need three C functions. The first one is called when the predicate is first activated; the second one is called on backtracking to provide (possibly) other solutions; the last one is called on pruning. Note also that we normally also need to preserve some information to find out the next solution.

In fact the role of the two functions can be better understood from the following Prolog definition

```prolog
p :- start.
p :- repeat, continue.
```

where `start` and `continue` correspond to the two C functions described above.

As an example we will consider implementing in C a predicate `n100(N)` which, when called with an instantiated argument should succeed if that argument is a numeral less or equal to 100, and, when called with an uninstantiated argument, should provide, by backtracking, all the positive integers less or equal to 100.

To do that we first declare a structure, which can only consist of Prolog terms, containing the information to be preserved on backtracking and a pointer variable to a structure of that type.

```c
#include "YAPInterface.h"

static int start_n100(void);
static int continue_n100(void);

typedef struct {
    YAP_Term next_solution; /* the next solution */
} n100_data_type;

n100_data_type *n100_data;
```

We now write the C function to handle the first call:

```c
static int start_n100(void)
{ /* include implementation */
```
YAP_Term t = YAP_ARG1;
YAP_PRESERVE_DATA(n100_data,n100_data_type);
if(YAP_IsVarTerm(t)) {
    n100_data->next_solution = YAP_MkIntTerm(0);
    return continue_n100();
}
if(!YAP_IsIntTerm(t) || YAP_IntOfTerm(t)<0 || YAP_IntOfTerm(t)>100) {
    YAP_cut_fail();
} else {
    YAP_cut_succeed();
}
}

The routine starts by getting the dereference value of the argument. The call to YAP_PRESERVE_DATA is used to initialize the memory which will hold the information to be preserved across backtracking. The first argument is the variable we shall use, and the second its type. Note that we can only use YAP_PRESERVE_DATA once, so often we will want the variable to be a structure.

If the argument of the predicate is a variable, the routine initializes the structure to be preserved across backtracking with the information required to provide the next solution, and exits by calling continue_n100 to provide that solution.

If the argument was not a variable, the routine then checks if it was an integer, and if so, if its value is positive and less than 100. In that case it exits, denoting success, with YAP_cut_succeed, or otherwise exits with YAP_cut_fail denoting failure.

The reason for using for using the functions YAP_cut_succeed and YAP_cut_fail instead of just returning a non-zero value in the first case, and zero in the second case, is that otherwise, if backtracking occurred later, the routine continue_n100 would be called to provide additional solutions.

The code required for the second function is

```c
static int continue_n100(void)
{
    int n;
    YAP_Term t;
    YAP_Term sol = YAP_ARG1;
    YAP_PRESERVED_DATA(n100_data,n100_data_type);
    n = YAP_IntOfTerm(n100_data->next_solution);
    if( n == 100) {
        t = YAP_MkIntTerm(n);
        YAP_Unify(sol,t);
        YAP_cut_succeed();
    }
    else {  
        YAP_Unify(sol,n100_data->next_solution);
        n100_data->next_solution = YAP_MkIntTerm(n+1);
        return(TRUE);
    }
}
```
Note that again the macro \texttt{YAP\_PRESERVED\_DATA} is used at the beginning of the function to access the data preserved from the previous solution. Then it checks if the last solution was found and in that case exits with \texttt{YAP\_cut\_succeed} in order to cut any further backtracking. If this is not the last solution then we save the value for the next solution in the data structure and exit normally with 1 denoting success. Note also that in any of the two cases we use the function \texttt{YAP\_unify} to bind the argument of the call to the value saved in \texttt{n100\_state->next\_solution}.

Note also that the only correct way to signal failure in a backtrackable predicate is to use the \texttt{YAP\_cut\_fail} macro.

Backtrackable predicates should be declared to YAP, in a way similar to what happened with deterministic ones, but using instead a call to

\begin{verbatim}
void YAP\_UserBackCutCPredicate(char *name, 
    int *init(), int *cont(), int *cut(),
    unsigned long int arity, unsigned int sizeof);
\end{verbatim}

where \texttt{name} is a string with the name of the predicate, \texttt{init}, \texttt{cont}, \texttt{cut} are the C functions used to start, continue and when pruning the execution of the predicate, \texttt{arity} is the predicate arity, and \texttt{sizeof} is the size of the data to be preserved in the stack. In this example, we would have something like

\begin{verbatim}
void
init\_n100(void)
{
    YAP\_UserBackCutCPredicate("n100", start\_n100, continue\_n100, NULL, 1, 1);
}
\end{verbatim}

Notice that we do not actually need to do anything on receiving a cut in this case.

\section{22.10 Loading Object Files}

The primitive predicate

\begin{verbatim}
load\_foreign\_files(Files,Libs,InitRoutine)
\end{verbatim}

should be used, from inside YAP, to load object files produced by the C compiler. The argument \texttt{ObjectFiles} should be a list of atoms specifying the object files to load, \texttt{Libs} is a list (possibly empty) of libraries to be passed to the unix loader (\texttt{ld}) and \texttt{InitRoutine} is the name of the C routine (to be called after the files are loaded) to perform the necessary declarations to YAP of the predicates defined in the files.

YAP will search for \texttt{ObjectFiles} in the current directory first. If it cannot find them it will search for the files using the environment variable \texttt{YAPLIBDIR}, if defined, or in the default library.

YAP also supports the SWI-Prolog interface to loading foreign code:

\begin{verbatim}
open\_shared\_object(+File, -Handle)
\end{verbatim}

File is the name of a shared object file (called dynamic load library in MS-Windows). This file is attached to the current process and \texttt{Handle} is unified with a handle to the library. Equivalent to \texttt{open\_shared\_object(File, [], Handle)}. See also \texttt{load\_foreign\_library/[1,2]}. 

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 lazily (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. In SWI-Prolog, normally this function installs foreign language predicates using calls to `PL_register_foreign()`.

22.11 Saving and Restoring
YAP4 currently does not support `save` and `restore` for object code loaded with `load_foreign_files`. We plan to support save and restore in future releases of YAP.

22.12 Changes to the C-Interface in YAP4
YAP4 includes several changes over the previous `load_foreign_files` interface. These changes were required to support the new binary code formats, such as ELF used in Solaris2 and Linux.

- All Names of YAP objects now start with `YAP_`. This is designed to avoid clashes with other code. Use `YAPInterface.h` to take advantage of the new interface. `c_interface.h` is still available if you cannot port the code to the new interface.

- Access to elements in the new interface always goes through `functions`. This includes access to the argument registers, `YAP_ARG1` to `YAP_ARG16`. This change breaks code such as `unify(&ARG1,&t)`, which is nowadays:
  ```c
  { 
    YAP_Unify(ARG1, t);
  }
  ```

- `cut_fail()` and `cut_succeed()` are now functions.

- The use of `Deref` is deprecated. All functions that return Prolog terms, including the ones that access arguments, already dereference their arguments.

- Space allocated with `PRESERVE_DATA` is ignored by garbage collection and stack shifting. As a result, any pointers to a Prolog stack object, including some terms, may be corrupted after garbage collection or stack shifting. Prolog terms should instead be stored as arguments to the backtrackable procedure.
23 Using YAP as a Library

YAP can be used as a library to be called from other programs. To do so, you must first create the YAP library:

```
make library
make install_library
```

This will install a file `libyap.a` in `LIBDIR` and the Prolog headers in `INCLUDEDIR`. The library contains all the functionality available in YAP, except the foreign function loader and for YAP's startup routines.

To actually use this library you must follow a five step process:

1. You must initialize the YAP environment. A single function, `YAP_FastInit` asks for a contiguous chunk in your memory space, fills it in with the data-base, and sets up YAP's stacks and execution registers. You can use a saved space from a standard system by calling `save_program/1`.

2. You then have to prepare a query to give to YAP. A query is a Prolog term, and you just have to use the same functions that are available in the C-interface.

3. You can then use `YAP_RunGoal(query)` to actually evaluate your query. The argument is the query term `query`, and the result is 1 if the query succeeded, and 0 if it failed.

4. You can use the term destructor functions to check how arguments were instantiated.

5. If you want extra solutions, you can use `YAP_RestartGoal()` to obtain the next solution.

The next program shows how to use this system. We assume the saved program contains two facts for the procedure `b`:

```c
#include <stdio.h>
#include "YAP/YAPIInterface.h"

int main(int argc, char *argv[]) {
    if (YAP_FastInit("saved_state") == YAP_BOOT_ERROR)
        exit(1);
    if (YAP_RunGoal(YAP_MkAtomTerm(YAP_LookupAtom("do")))) {
        printf("Success\n");
        while (YAP_RestartGoal())
            printf("Success\n");
    }
    printf("NO\n");
}
```

The program first initializes YAP, calls the query for the first time and succeeds, and then backtracks twice. The first time backtracking succeeds, the second it fails and exits.

To compile this program it should be sufficient to do:
You may need to adjust the libraries and library paths depending on the Operating System and your installation of YAP.

Note that YAP4.3.0 provides the first version of the interface. The interface may change and improve in the future.

The following C-functions are available from YAP:

- `YAP_CompileClause(YAP_Term Clause)` Compile the Prolog term `Clause` and assert it as the last clause for the corresponding procedure.
- `int YAP_ContinueGoal(void)` Continue execution from the point where it stopped.
- `void YAP_Error(int ID, YAP_Term Cause, char * error_description)` Generate an YAP System Error with description given by the string `error_description`. `ID` is the error ID, if known, or 0. `Cause` is the term that caused the crash.
- `void YAP_Exit(int exit_code)` Exit YAP immediately. The argument `exit_code` gives the error code and is supposed to be 0 after successful execution in Unix and Unix-like systems.
- `YAP_Term YAP_GetValue(Atom at)` Return the term value associated with the atom `at`. If no such term exists the function will return the empty list.
- `YAP_FastInit(char * SavedState)` Initialize a copy of YAP from `SavedState`. The copy is monolithic and currently must be loaded at the same address where it was saved. `YAP_FastInit` is a simpler version of `YAP_Init`.
- `YAP_Init(InitInfo)` Initialize YAP. The arguments are in a C structure of type `YAP_init_args`.
  The fields of `InitInfo` are `char * SavedState`, `int HeapSize`, `int StackSize`, `int TrailSize`, `int NumberofWorkers`, `int SchedulerLoop`, `int DelayedReleaseLoad`, `int argc`, `char ** argv`, `int ErrorNo`, and `char * ErrorCause`. The function returns an integer, which indicates the current status. If the result is `YAP_BOOT_ERROR` booting failed.
  If `SavedState` is not NULL, try to open and restore the file `SavedState`. Initially YAP will search in the current directory. If the saved state does not exist in the current directory YAP will use either the default library directory or the directory given by the environment variable `YAPLIBDIR`. Note that currently the saved state must be loaded at the same address where it was saved.
  If `HeapSize` is different from 0 use `HeapSize` as the minimum size of the Heap (or code space). If `StackSize` is different from 0 use `HeapSize` as the minimum size for the Stacks. If `TrailSize` is different from 0 use `TrailSize` as the minimum size for the Trails. The `NumberofWorkers`, `NumberofWorkers`, and `DelayedReleaseLoad` are only of interest to the or-parallel system.
  The argument count `argc` and string of arguments `argv` arguments are to be passed to user programs as the arguments used to call YAP.
  If booting failed you may consult `ErrorNo` and `ErrorCause` for the cause of the error, or call `YAP_Error(ErrorNo,0L,ErrorCause)` to do default processing.
- `void YAP_PutValue(Atom at, YAP_Term value)` Associate the term `value` with the atom `at`. The term `value` must be a constant. This functionality is used by YAP as a simple way for controlling and communicating with the Prolog run-time.
• **YAP_Term YAP_Read(int (*)(void) GetC)** Parse a Term using the function GetC to input characters.

• **YAP_Term YAP_Write(YAP_Term t)** Copy a Term t and all associated constraints. May call the garbage collector and returns 0L on error (such as no space being available).

• **void YAP_Write(YAP_Term t, void (*)(int) PutC, int flags)** Write a Term t using the function PutC to output characters. The term is written according to a mask of the following flags in the flag argument: YAP_WRITE_QUOTED, YAP_WRITE_HANDLE_VARS, and YAP_WRITE_IGNORE_OPS.

• **void YAP_WriteBuffer(YAP_Term t, char * buff, unsigned int size, int flags)** Write a YAP_Term t to buffer buff with size size. The term is written according to a mask of the following flags in the flag argument: YAP_WRITE_QUOTED, YAP_WRITE_HANDLE_VARS, and YAP_WRITE_IGNORE_OPS.

• **void YAP_InitConsult(int mode, char * filename)** Enter consult mode on file filename. This mode maintains a few data-structures internally, for instance to know whether a predicate before or not. It is still possible to execute goals in consult mode. If mode is TRUE the file will be reconsulted, otherwise just consulted. In practice, this function is most useful for bootstrapping Prolog, as otherwise one may call the Prolog predicate compile/1 or consult/1 to do compilation.

  Note that it is up to the user to open the file filename. The YAP_InitConsult function only uses the file name for internal bookkeeping.

• **void YAP_EndConsult(void)** Finish consult mode.

Some observations:

• The system will core dump if you try to load the saved state in a different address from where it was made. This may be a problem if your program uses mmap. This problem will be addressed in future versions of YAP.

• Currently, the YAP library will pollute the name space for your program.

• The initial library includes the complete YAP system. In the future we plan to split this library into several smaller libraries (e.g. if you do not want to perform I/O).

• You can generate your own saved states. Look at the boot.yap and init.yap files.
24 Compatibility with Other Prolog systems

YAP has been designed to be as compatible as possible with other Prolog systems, and initially with C-Prolog. More recent work on YAP has included features initially proposed for the Quintus and SICStus Prolog systems.

Developments since YAP4.1.6 we have striven at making YAP compatible with the ISO-Prolog standard.

24.1 Compatibility with the C-Prolog interpreter

24.1.1 Major Differences between YAP and C-Prolog.

YAP includes several extensions over the original C-Prolog system. Even so, most C-Prolog programs should run under YAP without changes.

The most important difference between YAP and C-Prolog is that, being YAP a compiler, some changes should be made if predicates such as `assert`, `clause` and `retract` are used. First predicates which will change during execution should be declared as `dynamic` by using commands like:

```
:- dynamic f/n.
```

where `f` is the predicate name and `n` is the arity of the predicate. Note that several such predicates can be declared in a single command:

```
:- dynamic f/2, ..., g/1.
```

Primitive predicates such as `retract` apply only to dynamic predicates. Finally note that not all the C-Prolog primitive predicates are implemented in YAP. They can easily be detected using the `unknown` system predicate provided by YAP.

Last, by default YAP enables character escapes in strings. You can disable the special interpretation for the escape character by using:

```
:- yap_flag(character_escapes,off).
```

or by using:

```
:- yap_flag(language,cprolog).
```

24.1.2 YAP predicates fully compatible with C-Prolog

These are the Prolog built-ins that are fully compatible in both C-Prolog and YAP:

```
!                     !
!/0.......................... 33  <                       <
```

```
,                     =       =
./2.......................... 33  =../2.......................... 41
```

```
;                     =:=/2.......................... 52
`/2.......................... 33  ==</2.......................... 52
```

```
\=\=/2.......................... 53
```
24.1.3 YAP predicates not strictly compatible with C-Prolog

These are YAP built-ins that are also available in C-Prolog, but that are not fully compatible:

24.1.4 YAP predicates not available in C-Prolog

These are YAP built-ins not available in C-Prolog.
Chapter 24: Compatibility with Other Prolog systems

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>del_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_attr/2</td>
<td>157</td>
</tr>
<tr>
<td>del_attrs/1</td>
<td>157</td>
</tr>
<tr>
<td>del_edges/3</td>
<td>142</td>
</tr>
<tr>
<td>del_max_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_min_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_vertices/3</td>
<td>142</td>
</tr>
<tr>
<td>delete/3</td>
<td>110</td>
</tr>
<tr>
<td>delete_file/1</td>
<td>133</td>
</tr>
<tr>
<td>delete_file/2</td>
<td>133</td>
</tr>
<tr>
<td>dgraph_add_edge/4</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_add_edges/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_add_vertex/3</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_add_vertices/3</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_complement/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_compose/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_del_edge/4</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_del_edges/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_del_vertex/3</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_del_vertices/3</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_edge/2</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_edges/2</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_isomorphic/4</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_max_path/5</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_min_path/5</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_min_paths/3</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_neighbors/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_neighbours/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_new/1</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_path/3</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_symmetric_closure/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_to_ugraph/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_to_undgraph/2</td>
<td>147</td>
</tr>
<tr>
<td>dgraph_top_sort/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_top_sort/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_transitive_closure/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_transpose/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_vertices/2</td>
<td>144</td>
</tr>
<tr>
<td>dif/2</td>
<td>153</td>
</tr>
<tr>
<td>directory_files/2</td>
<td>133</td>
</tr>
<tr>
<td>discontiguous/1 (directive)</td>
<td>25</td>
</tr>
<tr>
<td>display/1</td>
<td>60</td>
</tr>
<tr>
<td>display/2</td>
<td>66</td>
</tr>
<tr>
<td>do_not_compile_expressions/0</td>
<td>23</td>
</tr>
<tr>
<td>duplicate_term/2</td>
<td>42</td>
</tr>
<tr>
<td>dynamic/1</td>
<td>71</td>
</tr>
<tr>
<td>dynamic_predicate/2</td>
<td>71</td>
</tr>
<tr>
<td>eraseall/1</td>
<td>76</td>
</tr>
<tr>
<td>exception/3</td>
<td>38</td>
</tr>
<tr>
<td>exclude/3</td>
<td>114</td>
</tr>
<tr>
<td>exec/3</td>
<td>135</td>
</tr>
</tbody>
</table>

**F**

false/0 ........................................... 33
fields/2 .......................................... 13
fields/3 .......................................... 113
file_base_name/2 .................................. 56
file_exists/1 ..................................... 133
file_exists/2 ..................................... 133
file_filter/3 ..................................... 113
file_filter_with_init/5 ............................ 113
file_name_extension/3 ............................... 56
file_property/2 ................................... 134
file_search_path/2 ................................ 25
filter/3 .......................................... 113
findall/4 .......................................... 78
flatten/2 ......................................... 110
float/1 ........................................... 40
flush_output/0 .................................... 56
flush_output/1 .................................... 56
forall/2 ........................................... 36
foreach/2 ......................................... 150
format/1 .......................................... 64
format/2 .......................................... 61
format/3 .......................................... 64
format_to_chars/3 .................................. 131
format_to_chars/4 .................................. 131
fragile ............................................ 138
free_variables/4 ................................... 106
freeze/2 .......................................... 153
frozen/2 .......................................... 154

**G**

garbage_collect/0 ................................ 37
garbage_collectAtoms/0 ............................... 37
gc/0 ............................................. 37
gen_assoc/3 ...................................... 107
get/2 ............................................ 67
get_assoc/3 ....................................... 108
get_assoc/5 ....................................... 107
get_attr/3 ....................................... 156
get_atts/2 ........................................ 157
get_atts/2 ........................................ 159
get_byte/1 ....................................... 65
get_byte/2 ....................................... 67
get_char/1 ....................................... 65
get_char/2 ....................................... 67
get_code/1 ....................................... 65
get_code/2 ....................................... 67
get_from_heap/4 ................................... 109
get_label/3 ....................................... 140
get Mutable/2 .................................... 84
get_next_assoc/4 .................................. 107
Chapter 24: Compatibility with Other Prolog systems 231

matrix_minarg/2 ........................................ 117
matrix_ndims/2 ........................................ 116
matrix_new/3 ........................................... 116
matrix_new/4 ........................................... 116
matrix_new_set/4 ........................................ 116
matrix_offset_to_arg/3 ................................ 117
matrix_op/4 ............................................. 117
matrix_op_to_cols/4 ..................................... 117
matrix_op_to_lines/4 .................................... 117
matrix_reorder/3 ....................................... 118
matrix_row/3 ............................................ 118
matrix_select/4 ........................................ 118
matrix_set/3 ............................................. 116
matrix_set_all/2 ......................................... 116
matrix_shuffle/3 ........................................ 117
matrix_size/2 ........................................... 116
matrix_sum/2 ........................................... 117
matrix_to_list/2 ....................................... 116
matrix_type/2 .......................................... 116
max_assoc/3 ............................................. 108
max_list/2 .............................................. 112
member/2 ............................................... 110
memberchk/2 ........................................... 110
merge/3 .................................................. 121
message_hook/3 ......................................... 40
message_queue_create/1 ................................. 185
message_queue_destroy/1 ............................... 185
message_to_string/2 .................................... 40
meta_predicate/1 (directive) ............................. 31
min_assoc/3 ............................................. 108
min_list/2 .............................................. 112
min_of_heap/3 ......................................... 109
min_of_heap/5 ......................................... 109
mktemp/2 ............................................... 134
mktime/2 ............................................... 133
module/1 ............................................... 30
module/2 (directive) .................................... 30
module/3 (directive) .................................... 30
module_transparent/1 (directive) ....................... 32
mpi_barrier/0 .......................................... 148
mpi_broadcast/2 ......................................... 148
mpi_bcast/3 ............................................. 148
mpi_comm_rank/1 ........................................ 147
mpi_comm_size/1 ........................................ 147
mpi_finalize/0 ........................................... 147
mpi_gc/0 ............................................... 148
mpi_init/0 .............................................. 147
mpi_irecv/3 ............................................. 148
mpi_isend/4 ............................................. 147
mpi_recv/3 ............................................. 147
mpi_send/3 ............................................. 147
mpi_test/2 .............................................. 148
mpi_test_recv/3 ......................................... 148
mpi_version/2 .......................................... 147
mpi_wait/2 ............................................. 148
mpi_wait_recv/3 ......................................... 148
multifile/1 (directive) ................................ 24
mutex_create/1 ......................................... 188
mutex_destroy/1 ....................................... 188
mutex_lock/1 .......................................... 188
mutex_statistics/0 ..................................... 184
mutex_trylock/1 ........................................ 188
mutex_unlock/1 ........................................ 189
mutex_unlock_all/0 .................................... 189

N

nb_beam/1 ............................................... 121
nb_beam_add/3 .......................................... 121
nb_beam_close/1 ........................................ 121
nb_beam_del/3 .......................................... 121
nb_beam_empty/1 ....................................... 121
nb_beam Peek/3 ......................................... 121
nb_beam_size/2 ......................................... 121
nb_current/2 .......................................... 86, 152
nb_delete/1 ........................................... 152
nb_delete/2 ........................................... 86
nb_getval/2 ........................................... 85, 151
nb_heap/1 .............................................. 120
nb_heap_add/3 ......................................... 120
nb_heap_close/1 ....................................... 120
nb_heap_del/3 ......................................... 120
nb_heap_empty/1 ....................................... 121
nb_heap Peek/3 ......................................... 120
nb_heap_size/2 ......................................... 120
nb_linkarg/3 ........................................... 86
nb_linkval/2 ........................................... 85
nb_queue/1 ............................................. 120
nb_queue_close/3 ...................................... 120
nb_queue_dequeue/2 ................................... 120
nb_queue_empty/1 ...................................... 120
nb_queue_enqueue/2 ................................... 120
nb_queue Peek/2 ........................................ 120
nb_queue_size/2 ........................................ 120
nb_set_shared_arg/3 .................................. 86
nb_set_shared_val/2 ................................... 85
nb_setarg/3 ........................................... 86
nb_setval/2 ............................................ 85, 151
nb_source/0 ............................................ 23
nb_target/0 ............................................ 37
nb_target/0 ............................................ 199
nth/3 ................................................... 110
nth/4 ................................................... 111
nth_instance/3 ......................................... 76
nth_instance/4 ......................................... 76
nth0/3 .................................................. 110
nth0/4 .................................................. 110
nth1/3 .................................................. 110, 149
nth1/4 .................................................. 111
number_atom/2 ......................................... 44
number_chars/2 ......................................... 44
number_codes/2 ......................................... 44
<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>number_to_chars/2</td>
<td>131</td>
</tr>
<tr>
<td>number_to_chars/3</td>
<td>131</td>
</tr>
<tr>
<td>numchars/3</td>
<td>41</td>
</tr>
<tr>
<td>numlist/3</td>
<td>112</td>
</tr>
<tr>
<td>on_cleanup/1</td>
<td>140</td>
</tr>
<tr>
<td>on_signal/3</td>
<td>83</td>
</tr>
<tr>
<td>once/1</td>
<td>36</td>
</tr>
<tr>
<td>open/3</td>
<td>53</td>
</tr>
<tr>
<td>open/4</td>
<td>54</td>
</tr>
<tr>
<td>open_chars_stream/2</td>
<td>132</td>
</tr>
<tr>
<td>open_shared_object/2</td>
<td>219</td>
</tr>
<tr>
<td>open_shared_object/3</td>
<td>220</td>
</tr>
<tr>
<td>ord_add_element/3</td>
<td>121</td>
</tr>
<tr>
<td>ord_del_element/3</td>
<td>121</td>
</tr>
<tr>
<td>ord_disjoint/2</td>
<td>121</td>
</tr>
<tr>
<td>ord_insert/3</td>
<td>122</td>
</tr>
<tr>
<td>ord_intersect/2</td>
<td>122</td>
</tr>
<tr>
<td>ord_intersect/3</td>
<td>122</td>
</tr>
<tr>
<td>ord_intersect/4</td>
<td>122</td>
</tr>
<tr>
<td>ord_list_to_assoc/2</td>
<td>108</td>
</tr>
<tr>
<td>ord_member/2</td>
<td>121</td>
</tr>
<tr>
<td>ord_seteq/2</td>
<td>122</td>
</tr>
<tr>
<td>ord_setproduct/3</td>
<td>122</td>
</tr>
<tr>
<td>ord_subtract/3</td>
<td>122</td>
</tr>
<tr>
<td>ord_symdiff/3</td>
<td>122</td>
</tr>
<tr>
<td>ord_union/2</td>
<td>122</td>
</tr>
<tr>
<td>ord_union/3</td>
<td>122</td>
</tr>
<tr>
<td>ord_union/4</td>
<td>122</td>
</tr>
<tr>
<td>ordsupset/2</td>
<td>122</td>
</tr>
<tr>
<td>partition/4</td>
<td>114</td>
</tr>
<tr>
<td>partition/5</td>
<td>115</td>
</tr>
<tr>
<td>path/1</td>
<td>24</td>
</tr>
<tr>
<td>peek_byte/1</td>
<td>66</td>
</tr>
<tr>
<td>peek_byte/2</td>
<td>67</td>
</tr>
<tr>
<td>peek_char/1</td>
<td>66</td>
</tr>
<tr>
<td>peek_char/2</td>
<td>67</td>
</tr>
<tr>
<td>peek_code/1</td>
<td>66</td>
</tr>
<tr>
<td>peek_code/2</td>
<td>67</td>
</tr>
<tr>
<td>permutation/2</td>
<td>111</td>
</tr>
<tr>
<td>phrase/2</td>
<td>80</td>
</tr>
<tr>
<td>phrase/3</td>
<td>80</td>
</tr>
<tr>
<td>pid/1</td>
<td>135</td>
</tr>
<tr>
<td>plus/3</td>
<td>53</td>
</tr>
<tr>
<td>popen/3</td>
<td>135</td>
</tr>
<tr>
<td>portray_clause/1</td>
<td>73</td>
</tr>
<tr>
<td>portray_clause/2</td>
<td>73</td>
</tr>
<tr>
<td>predicate_property/2</td>
<td>73</td>
</tr>
<tr>
<td>presort/3</td>
<td>47</td>
</tr>
<tr>
<td>print/2</td>
<td>66</td>
</tr>
<tr>
<td>print_message/2</td>
<td>39</td>
</tr>
<tr>
<td>print_message_lines/3</td>
<td>40</td>
</tr>
<tr>
<td>process/2</td>
<td>113</td>
</tr>
<tr>
<td>profile_data/3</td>
<td>87</td>
</tr>
<tr>
<td>profiled_reset/0</td>
<td>88</td>
</tr>
<tr>
<td>profinit/0</td>
<td>88</td>
</tr>
<tr>
<td>profoff/0</td>
<td>88</td>
</tr>
<tr>
<td>profon/0</td>
<td>88</td>
</tr>
<tr>
<td>project_attributes/2</td>
<td>160</td>
</tr>
<tr>
<td>prolog_file_name/2</td>
<td>25</td>
</tr>
<tr>
<td>prolog_flag/3</td>
<td>102</td>
</tr>
<tr>
<td>prolog_initialization/1</td>
<td>103</td>
</tr>
<tr>
<td>prolog_load_context/2</td>
<td>103</td>
</tr>
<tr>
<td>public/1 (directive)</td>
<td>25</td>
</tr>
<tr>
<td>put/2</td>
<td>67</td>
</tr>
<tr>
<td>put_assoc/4</td>
<td>108</td>
</tr>
<tr>
<td>put_attr/3</td>
<td>156</td>
</tr>
<tr>
<td>put_att/2</td>
<td>157</td>
</tr>
<tr>
<td>put_att/2</td>
<td>159</td>
</tr>
<tr>
<td>put_byte/1</td>
<td>65</td>
</tr>
<tr>
<td>put_byte/2</td>
<td>67</td>
</tr>
<tr>
<td>put_char/1</td>
<td>65</td>
</tr>
<tr>
<td>put_char/2</td>
<td>67</td>
</tr>
<tr>
<td>put_code/1</td>
<td>65</td>
</tr>
<tr>
<td>put_code/2</td>
<td>67</td>
</tr>
<tr>
<td>put_label/4</td>
<td>141</td>
</tr>
<tr>
<td>putenv/2</td>
<td>81</td>
</tr>
<tr>
<td>queue_to_list/2</td>
<td>123</td>
</tr>
<tr>
<td>random/1</td>
<td>124</td>
</tr>
<tr>
<td>random/3</td>
<td>124</td>
</tr>
<tr>
<td>randseq/3</td>
<td>124</td>
</tr>
<tr>
<td>randset/3</td>
<td>124</td>
</tr>
<tr>
<td>ramnum/1</td>
<td>122</td>
</tr>
<tr>
<td>ranstart/0</td>
<td>123</td>
</tr>
<tr>
<td>ranstart/1</td>
<td>123</td>
</tr>
<tr>
<td>ranunif/2</td>
<td>123</td>
</tr>
<tr>
<td>rb_apply/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_clone/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_del_max/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_del_min/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_del/3</td>
<td>125</td>
</tr>
<tr>
<td>rb_del/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_empty/1</td>
<td>125</td>
</tr>
<tr>
<td>rb_insert/4</td>
<td>125</td>
</tr>
<tr>
<td>rb_insert_new/4</td>
<td>125</td>
</tr>
<tr>
<td>rb_keys/2</td>
<td>126</td>
</tr>
<tr>
<td>rb_lookup/3</td>
<td>125</td>
</tr>
<tr>
<td>rb_lookupall/3</td>
<td>125</td>
</tr>
<tr>
<td>rb_map/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_max/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_min/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_new/1</td>
<td>125</td>
</tr>
<tr>
<td>rb_next/4</td>
<td>127</td>
</tr>
<tr>
<td>rb_partial_map/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_previous/4</td>
<td>127</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>rb_size/2</td>
<td>126</td>
</tr>
<tr>
<td>rb_update/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_visit/2</td>
<td>126</td>
</tr>
<tr>
<td>reachable/3</td>
<td>144</td>
</tr>
<tr>
<td>read/2</td>
<td>66</td>
</tr>
<tr>
<td>read_file_to_codes/3</td>
<td>125</td>
</tr>
<tr>
<td>read_file_to_terms/3</td>
<td>125</td>
</tr>
<tr>
<td>read_from_char/2</td>
<td>132</td>
</tr>
<tr>
<td>read_line_to_codes/2</td>
<td>124</td>
</tr>
<tr>
<td>read_line_to_codes/3</td>
<td>124</td>
</tr>
<tr>
<td>read_stream_to_codes/3</td>
<td>125</td>
</tr>
<tr>
<td>read_term/2</td>
<td>59</td>
</tr>
<tr>
<td>read_term/3</td>
<td>66</td>
</tr>
<tr>
<td>recorda/3</td>
<td>75</td>
</tr>
<tr>
<td>recorda_at/3</td>
<td>75</td>
</tr>
<tr>
<td>recorda_ifnot/3</td>
<td>75, 77</td>
</tr>
<tr>
<td>recordz_at/3</td>
<td>75</td>
</tr>
<tr>
<td>recordz_ifnot/3</td>
<td>76</td>
</tr>
<tr>
<td>reexport/1</td>
<td>32</td>
</tr>
<tr>
<td>reexport/2</td>
<td>32</td>
</tr>
<tr>
<td>regexp/3</td>
<td>127</td>
</tr>
<tr>
<td>regexp/4</td>
<td>128</td>
</tr>
<tr>
<td>remove_duplicates/2</td>
<td>111</td>
</tr>
<tr>
<td>remove_from_path/1</td>
<td>24</td>
</tr>
<tr>
<td>rename_file/2</td>
<td>134</td>
</tr>
<tr>
<td>reset_op_counters/0</td>
<td>197</td>
</tr>
<tr>
<td>reset_static_array/1</td>
<td>90</td>
</tr>
<tr>
<td>resize_static_array/3</td>
<td>91</td>
</tr>
<tr>
<td>restore/1</td>
<td>27</td>
</tr>
<tr>
<td>retract/1</td>
<td>73</td>
</tr>
<tr>
<td>retractall/1</td>
<td>73</td>
</tr>
<tr>
<td>reverse/2</td>
<td>111</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>same_length/2</td>
<td>111</td>
</tr>
<tr>
<td>save/2</td>
<td>26</td>
</tr>
<tr>
<td>save_program/1</td>
<td>26</td>
</tr>
<tr>
<td>save_program/2</td>
<td>27</td>
</tr>
<tr>
<td>scan_integer/3</td>
<td>112</td>
</tr>
<tr>
<td>scan_natural/3</td>
<td>112</td>
</tr>
<tr>
<td>search_for/2</td>
<td>112</td>
</tr>
<tr>
<td>select/3</td>
<td>111</td>
</tr>
<tr>
<td>selectchk/3</td>
<td>111</td>
</tr>
<tr>
<td>selectlist/3</td>
<td>114</td>
</tr>
<tr>
<td>serve_queue/3</td>
<td>123</td>
</tr>
<tr>
<td>set_input/1</td>
<td>56</td>
</tr>
<tr>
<td>set_output/1</td>
<td>56</td>
</tr>
<tr>
<td>set_prolog_flag/2</td>
<td>102</td>
</tr>
<tr>
<td>set_stream_position/2</td>
<td>57</td>
</tr>
<tr>
<td>set_value/2</td>
<td>76</td>
</tr>
<tr>
<td>setarg/3n</td>
<td>83</td>
</tr>
<tr>
<td>setenv/2</td>
<td>149</td>
</tr>
<tr>
<td>setrand/1</td>
<td>124</td>
</tr>
<tr>
<td>setup_call_catcher_cleanup/4</td>
<td>139</td>
</tr>
<tr>
<td>setup_call_cleanup/3</td>
<td>139</td>
</tr>
<tr>
<td>shell/0</td>
<td>135</td>
</tr>
<tr>
<td>shell/1</td>
<td>135, 136</td>
</tr>
</tbody>
</table>

**Chapter 24: Compatibility with Other Prolog systems**

show_op_counters/1                  197
show_ops_by_group/1                  197
show_table/1                           194
showprofes/0                          88
showprofes/1                          88
simple/1                               41
skip/2                                 67
sleep/1                                136
socket/2                               69
socket/4                               69
socket_accept/2                       70
socket_accept/3                       70
socket_bind/2                          69
socket_buffering/4                    70
socket_close/1                         69
socket_connect/3                      70
socket_disconnect/3                   70
socket_listen/2                        70
socket_select/5                        70
source/0                               23
source_file/1                           104
source_file/2                           104
source_location/2                      104
source_mode/2                           23
splay_access/5                          130
splay_delete/4                         130
splay_init/3                            130
splay_insert/4                          130
splay_join/3                            131
splay_split/5                           131
split/2                                 112
split/3                                 112
spy_write/2                             200
random/1                                53
start_low_level_trace/0                195
start_matlab/1                          118
static_array/3                          90, 91
static_array_location/4                90
static_array_properties/3              90
static_array_to_term/3                 91
statistics/2                            93
stream_position/2                      58
stream_position_data/3                 58
stream_property/2                      57
stream_select/3                         56
style_check/1                           24
sub_atom/5                              44
sublist/2                               111
subunames/2                             137
subunames_chk/2                         137
succe/3                                53
suffix/2                                111
sum_list/2                              111
sum_list/3                              111
sumargs/4                               114
sumlist/2                               112
sumlist/4                               114
sumnodes/4                              114
system/0                                136
24.1.5 YAP predicates not available in C-Prolog

These are C-Prolog built-ins not available in YAP:

- 'LC' The following Prolog text uses lower case letters.
- 'NOLC' The following Prolog text uses upper case letters only.

24.2 Compatibility with the Quintus and SICStus Prolog systems

The Quintus Prolog system was the first Prolog compiler to use Warren’s Abstract Machine. This system was very influential in the Prolog community. Quintus Prolog implemented compilation into an abstract machine code, which was then emulated. Quintus Prolog also included several new built-ins, an extensive library, and in later releases a garbage collector.

The SICStus Prolog system, developed at SICS (Swedish Institute of Computer Science), is an emulator based Prolog system largely compatible with Quintus Prolog. SICStus Prolog has evolved through several versions. The current version includes several extensions, such as an object implementation, co-routining, and constraints.

Recent work in YAP has been influenced by work in Quintus and SICStus Prolog. Wherever possible, we have tried to make YAP compatible with recent versions of these systems, and specifically of SICStus Prolog. You should use

```prolog
:- yap_flag(language, sicstus).
```

for maximum compatibility with SICStus Prolog.

24.2.1 Major Differences between YAP and SICStus Prolog

Both YAP and SICStus Prolog obey the Edinburgh Syntax and are based on the WAM. Even so, there are quite a few important differences:

- Differently from SICStus Prolog, YAP does not have a notion of interpreted code. All code in YAP is compiled.
- YAP does not support an intermediate byte-code representation, so the `fcompile/1` and `load/1` built-ins are not available in YAP.
- YAP implements escape sequences as in the ISO standard. SICStus Prolog implements Unix-like escape sequences.
- YAP implements `initialization/1` as per the ISO standard. Use `prolog_initialization/1` for the SICStus Prolog compatible built-in.
- Prolog flags are different in SICStus Prolog and in YAP.
• The SICStus Prolog on_exception/3 and raise_exception built-ins correspond to
the ISO built-ins catch/3 and throw/1.
• The following SICStus Prolog v3 built-ins are not (currently) implemented in YAP
(note that this is only a partial list): file_search_path/2, stream_interrupt/3,
reinitialize/0, help/0, help/1, trimcore/0, load_files/1, load_files/2, and
require/1.

The previous list is incomplete. We also cannot guarantee full compatibility for other
built-ins (although we will try to address any such incompatibilities). Last, SICStus
Prolog is an evolving system, so one can be expect new incompatibilities to be intro-
duced in future releases of SICStus Prolog.
• YAP allows asserting and abolishing static code during execution through the assert_
static/1 and abolish/1 built-ins. This is not allowed in Quintus Prolog or SICStus
Prolog.
• The socket predicates, although designed to be compatible with SICStus Prolog, are
built-ins, not library predicates, in YAP.
• This list is incomplete.

The following differences only exist if the language flag is set to yap (the default):
• The consult/1 predicate in YAP follows C-Prolog semantics. That is, it adds clauses
to the data base, even for preexisting procedures. This is different from consult/1 in
SICStus Prolog.
• By default, the data-base in YAP follows "immediate update semantics", instead of
"logical update semantics", as Quintus Prolog or SICStus Prolog do. The difference is
depicted in the next example:
  :- dynamic a/1.
  ?- assert(a(1)).
  :- retract(a(X)), X1 is X +1, assertz(a(X)).

With immediate semantics, new clauses or entries to the data base are visible in back-
tracking. In this example, the first call to retract/1 will succeed. The call to assertz/1
will then succeed. On backtracking, the system will retry retract/1. Because the
newly asserted goal is visible to retract/1, it can be retracted from the data base,
and retract(a(X)) will succeed again. The process will continue generating integers
for ever. Immediate semantics were used in C-Prolog.
With logical update semantics, any additions or deletions of clauses for a goal will not
affect previous activations of the goal. In the example, the call to assertz/1 will not
see the update performed by the assertz/1, and the query will have a single solution.
Calling yap_flag(update_semantics,logical) will switch YAP to use logical update
semantics.
• dynamic/1 is a built-in, not a directive, in YAP.
• By default, YAP fails on undefined predicates. To follow default SICStus Prolog use:
  :- yap_flag(unknown,error).
• By default, directives in YAP can be called from the top level.
24.2.2 YAP predicates fully compatible with SICStus Prolog

These are the Prolog built-ins that are fully compatible in both SICStus Prolog and YAP:

! .......................... 33
end_cleanup/2 .......................... 107
add/2 .......................... 43
atomic/1 .......................... 40
BB_delete/2 .......................... 78
attr_goal/2 .......................... 160
BB_get/2 .......................... 78
BB_put/2 .......................... 78
BB_update/3 .......................... 78
between/3 .......................... 53
break/0 .......................... 36

B

aggregate/3 .......................... 106
aggregate/4 .......................... 106
aggregate_all/3 .......................... 106
aggregate_all/4 .......................... 106
append/2 .......................... 110
append/3 .......................... 110
arg/3 .......................... 41
assor_to_list/2 .......................... 107
at_end_of_stream/0 .......................... 57
at_end_of_stream/1 .......................... 57
atom/1 .......................... 40
atom_codes/2 .......................... 43
atom_concat/2 .......................... 43
atom_number/2 .......................... 44
atom_to_chars/2 .......................... 131
atom_to_chars/3 .......................... 131
atom_to_term/3 .......................... 131
attribute/1 .......................... 40

C

call/1 .......................... 35
call_cleanup/1 .......................... 138
call_cleanup/2 .......................... 138
call_residue/2 .......................... 154
call_residue_vars/2 .......................... 154
callable/1 .......................... 41
call_char/2 .......................... 60
call_conversion/2 .......................... 58
call_character/2 .......................... 140
call_close/1 .......................... 55
call_compare/3 .......................... 46
call_compile/1 .......................... 21, 22
call_complement/2 .......................... 143
call_compose/3 .......................... 143
call_compound/1 .......................... 40
call_copy_term/2 .......................... 42
call_copy_term/3 .......................... 157
call_copy_term_v/2 .......................... 158
call_create/2 .......................... 83
call_current_atom/1 .......................... 73
call_current_char/2 .......................... 60

A

abort/0 .......................... 36
absolute_file_name/2 .......................... 56
absolute_file_name/3 .......................... 55
add_edges/3 .......................... 142
add_to_heap/4 .......................... 109
add_vertices/3 .......................... 142
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_input/1</td>
<td>57</td>
</tr>
<tr>
<td>current_key/2</td>
<td>76</td>
</tr>
<tr>
<td>current_module/1</td>
<td>92</td>
</tr>
<tr>
<td>current_module/2</td>
<td>92</td>
</tr>
<tr>
<td>current_op/3</td>
<td>103</td>
</tr>
<tr>
<td>current_output/1</td>
<td>57</td>
</tr>
<tr>
<td>current_predicate/1</td>
<td>73</td>
</tr>
<tr>
<td>current_predicate/2</td>
<td>73</td>
</tr>
<tr>
<td>current_stream/3</td>
<td>56</td>
</tr>
<tr>
<td>cyclic_term/1</td>
<td>136</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>datetime/1</td>
<td>133</td>
</tr>
<tr>
<td>db_reference/1</td>
<td>40</td>
</tr>
<tr>
<td>debugging/0</td>
<td>199</td>
</tr>
<tr>
<td>del_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_edges/3</td>
<td>142</td>
</tr>
<tr>
<td>del_max_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_min_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_vertices/3</td>
<td>142</td>
</tr>
<tr>
<td>delete/3</td>
<td>110</td>
</tr>
<tr>
<td>delete_file/1</td>
<td>133</td>
</tr>
<tr>
<td>delete_file/2</td>
<td>133</td>
</tr>
<tr>
<td>dgraph_del_vertex/3</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_del_vertices/3</td>
<td>144</td>
</tr>
<tr>
<td>dif/2</td>
<td>153</td>
</tr>
<tr>
<td>directory_files/2</td>
<td>133</td>
</tr>
<tr>
<td>discontiguous/1 (directive)</td>
<td>25</td>
</tr>
<tr>
<td>display/1</td>
<td>60</td>
</tr>
<tr>
<td>display/2</td>
<td>66</td>
</tr>
<tr>
<td>duplicate_term/2</td>
<td>42</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>edges/2</td>
<td>141</td>
</tr>
<tr>
<td>empty_assoc/1</td>
<td>107</td>
</tr>
<tr>
<td>empty_heap/1</td>
<td>109</td>
</tr>
<tr>
<td>empty_queue/1</td>
<td>123</td>
</tr>
<tr>
<td>environ/2</td>
<td>81, 134</td>
</tr>
<tr>
<td>exception/3</td>
<td>38</td>
</tr>
<tr>
<td>exec/3</td>
<td>135</td>
</tr>
<tr>
<td>expand_term/2</td>
<td>80</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>fail/0</td>
<td>33</td>
</tr>
<tr>
<td>false/0</td>
<td>33</td>
</tr>
<tr>
<td>file_exists/1</td>
<td>133</td>
</tr>
<tr>
<td>file_exists/2</td>
<td>133</td>
</tr>
<tr>
<td>file_property/2</td>
<td>134</td>
</tr>
<tr>
<td>file_search_path/2</td>
<td>25</td>
</tr>
<tr>
<td>fileerrors/0</td>
<td>68</td>
</tr>
<tr>
<td>findall/3</td>
<td>78</td>
</tr>
<tr>
<td>findall/4</td>
<td>78</td>
</tr>
<tr>
<td>flatten/2</td>
<td>110</td>
</tr>
<tr>
<td>float/1</td>
<td>40</td>
</tr>
<tr>
<td>flush_output/0</td>
<td>56</td>
</tr>
<tr>
<td>flush_output/1</td>
<td>56</td>
</tr>
<tr>
<td>foreach/2</td>
<td>106</td>
</tr>
<tr>
<td>format_to_chars/3</td>
<td>131</td>
</tr>
<tr>
<td>format_to_chars/4</td>
<td>131</td>
</tr>
<tr>
<td>fragile</td>
<td>138</td>
</tr>
<tr>
<td>free_variables/4</td>
<td>106</td>
</tr>
<tr>
<td>freeze/2</td>
<td>153</td>
</tr>
<tr>
<td>frozen/2</td>
<td>154</td>
</tr>
<tr>
<td>functor/3</td>
<td>41</td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>garbage_collect/0</td>
<td>37</td>
</tr>
<tr>
<td>garbage_collect_atoms/0</td>
<td>37</td>
</tr>
<tr>
<td>gc/0</td>
<td>37</td>
</tr>
<tr>
<td>gen_assoc/3</td>
<td>107</td>
</tr>
<tr>
<td>get/1</td>
<td>65</td>
</tr>
<tr>
<td>get/2</td>
<td>67</td>
</tr>
<tr>
<td>get_assoc/3</td>
<td>108</td>
</tr>
<tr>
<td>get_assoc/5</td>
<td>107</td>
</tr>
<tr>
<td>get_atts/2</td>
<td>159</td>
</tr>
<tr>
<td>get_from_heap/4</td>
<td>109</td>
</tr>
<tr>
<td>get_label/3</td>
<td>140</td>
</tr>
<tr>
<td>get_mutable/2</td>
<td>84</td>
</tr>
<tr>
<td>get_next_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>get_prev_assoc/4</td>
<td>108</td>
</tr>
<tr>
<td>geto/1</td>
<td>65</td>
</tr>
<tr>
<td>geto/2</td>
<td>67</td>
</tr>
<tr>
<td>getrand/1</td>
<td>124</td>
</tr>
<tr>
<td>ground/1</td>
<td>41</td>
</tr>
<tr>
<td>H</td>
<td></td>
</tr>
<tr>
<td>halt/0</td>
<td>36</td>
</tr>
<tr>
<td>halt/1</td>
<td>37</td>
</tr>
<tr>
<td>head_queue/2</td>
<td>123</td>
</tr>
<tr>
<td>heap_size/2</td>
<td>109</td>
</tr>
<tr>
<td>heap_to_list/2</td>
<td>109</td>
</tr>
<tr>
<td>host_id/1</td>
<td>134</td>
</tr>
<tr>
<td>host_name/1</td>
<td>134</td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>if/3</td>
<td>35</td>
</tr>
<tr>
<td>incore/1</td>
<td>35</td>
</tr>
<tr>
<td>initialization/0</td>
<td>103</td>
</tr>
<tr>
<td>integer/1</td>
<td>41</td>
</tr>
<tr>
<td>intersection/3</td>
<td>112</td>
</tr>
<tr>
<td>is/2</td>
<td>52</td>
</tr>
<tr>
<td>is_assoc/1</td>
<td>108</td>
</tr>
<tr>
<td>is_list/1</td>
<td>42</td>
</tr>
<tr>
<td>is_mutable/1</td>
<td>84</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>join_queue/3</td>
<td>123</td>
</tr>
<tr>
<td>jump_queue/3</td>
<td>123</td>
</tr>
</tbody>
</table>
### Chapter 24: Compatibility with Other Prolog systems

#### K
- `keysort/2` ........................................... 47
- `kill/2` ............................................ 134

#### L
- `last/2` ............................................. 110
- `leash/1` .......................................... 199
- `length/2` ......................................... 47
- `length_queue/2` ................................. 123
- `line_count/2` ................................... 58
- `line_position/2` ................................. 58
- `list_join_queue/3` .............................. 123
- `list_jump_queue/3` .............................. 123
- `list_to_assoc/2` ................................ 108
- `list_to_heap/2` .................................. 109
- `list_to_ord_set/2` ............................ 121
- `list_to_queue/2` .................................. 123
- `list_to_tree/2` ................................... 140
- `listing/1` .......................................... 73
- `load_files/2` ..................................... 21

#### M
- `make_directory/2` ............................... 134
- `make_queue/1` .................................... 123
- `map_assoc/2` ...................................... 108
- `map_assoc/3` ...................................... 108
- `map_tree/3` ....................................... 140
- `max_assoc/3` ...................................... 108
- `max_list/2` ........................................ 112
- `member/2` ......................................... 110
- `memberchk/2` ..................................... 110
- `merge/3` .......................................... 121
- `message_hook/3` .................................. 40
- `meta_predicate/1` (directive) ............... 31
- `min_assoc/3` ...................................... 108
- `min_list/2` ........................................ 112
- `min_of_heap/3` .................................... 109
- `min_of_heap/5` .................................... 109
- `mktemp/2` .......................................... 134
- `module/1` ......................................... 30
- `module/2` (directive) ........................... 30
- `module/3` (directive) ........................... 30
- `module_transparent/1` (directive) ........ 32
- `multifile/1` (directive) ....................... 24

#### N
- `name/2` ............................................. 42
- `neighbors/3` ...................................... 143
- `neighbours/3` .................................... 143
- `nl/0` ............................................... 66
- `nl/1` ............................................... 67
- `nodebug/0` ........................................ 199
- `nofileerrors/0` ................................... 68
- `nogc/0` ............................................. 37
- `nonvar` ........................................... 41
- `nospy` ............................................ 199
- `nospyall` ......................................... 199
- `notrace/0` ........................................ 200
- `nth/3` ............................................. 110
- `nth/4` ............................................. 111
- `nth0/3` .......................................... 110
- `nth0/4` .......................................... 110
- `nth1/3` .......................................... 110
- `nth1/4` .......................................... 111
- `number/1` ........................................ 41
- `number_codes/2` ................................ 44
- `number_to_chars/2` ............................ 131
- `number_to_chars/3` ............................ 131
- `numbervars/3` .................................... 41
- `numlist/3` ......................................... 112

#### O
- `on_cleanup/1` ..................................... 140
- `op/3` ............................................. 103
- `open/3` .......................................... 53
- `open_chars_stream/2` .......................... 132
- `ord_add_element/3` ............................ 121
- `ord_del_element/3` ............................ 121
- `ord_disjoint/2` .................................. 121
- `ord_insert/3` ..................................... 122
- `ord_intersect/2` .................................. 122
- `ord_intersect/3` .................................. 122
- `ord_intersect/4` .................................. 122
- `ord_list_to_assoc/2` ........................... 108
- `ord_member/2` ..................................... 121
- `ord_seteq/2` ..................................... 122
- `ord_setproduct/3` ............................... 122
- `ord_subtract/3` ................................... 122
- `ord_symdiff/3` ................................... 122
- `ord_union/2` ...................................... 122
- `ord_union/3` ...................................... 122
- `ord_union/4` ...................................... 122
- `ordsubset/2` ...................................... 122

#### P
- `peek_char/1` ...................................... 66
- `permutation/2` .................................... 111
- `phrase/2` ......................................... 80
- `phrase/3` ......................................... 80
- `pid/1` ............................................. 135
- `plus/3` ............................................ 53
- `popen/3` .......................................... 135
- `portray_clause/1` ............................... 73
- `portray_clause/2` ............................... 73
- `primitive/1` ...................................... 41
- `print/1` .......................................... 61
- `print/2` .......................................... 66
- `print_message/2` ................................. 39
- `print_message_lines/3` ........................ 40
- `project_attributes/2` ........................... 160
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog_file_name/1</td>
<td>25</td>
</tr>
<tr>
<td>prolog_flag/3</td>
<td>102</td>
</tr>
<tr>
<td>prolog_load_context/2</td>
<td>103</td>
</tr>
<tr>
<td>prompt/2</td>
<td>103</td>
</tr>
<tr>
<td>put/1</td>
<td>65</td>
</tr>
<tr>
<td>put/2</td>
<td>67</td>
</tr>
<tr>
<td>put_assoc/4</td>
<td>108</td>
</tr>
<tr>
<td>put_atts/2</td>
<td>159</td>
</tr>
<tr>
<td>put_label/4</td>
<td>141</td>
</tr>
<tr>
<td>queue_to_list/2</td>
<td>123</td>
</tr>
<tr>
<td>random/1</td>
<td>124</td>
</tr>
<tr>
<td>random/3</td>
<td>124</td>
</tr>
<tr>
<td>randseq/3</td>
<td>124</td>
</tr>
<tr>
<td>randset/3</td>
<td>124</td>
</tr>
<tr>
<td>rational/1</td>
<td>40</td>
</tr>
<tr>
<td>reachable/3</td>
<td>144</td>
</tr>
<tr>
<td>read/1</td>
<td>59</td>
</tr>
<tr>
<td>read/2</td>
<td>66</td>
</tr>
<tr>
<td>read_from_chars/2</td>
<td>132</td>
</tr>
<tr>
<td>remove_duplicates/2</td>
<td>111</td>
</tr>
<tr>
<td>rename_file/2</td>
<td>134</td>
</tr>
<tr>
<td>repeat/0</td>
<td>34</td>
</tr>
<tr>
<td>restore/1</td>
<td>27</td>
</tr>
<tr>
<td>reverse/2</td>
<td>111</td>
</tr>
<tr>
<td>same_length/2</td>
<td>111</td>
</tr>
<tr>
<td>save_program/1</td>
<td>26</td>
</tr>
<tr>
<td>save_program/2</td>
<td>27</td>
</tr>
<tr>
<td>see/1</td>
<td>59</td>
</tr>
<tr>
<td>seeing/1</td>
<td>59</td>
</tr>
<tr>
<td>seen/0</td>
<td>59</td>
</tr>
<tr>
<td>select/3</td>
<td>111</td>
</tr>
<tr>
<td>serve_queue/3</td>
<td>123</td>
</tr>
<tr>
<td>set_input/1</td>
<td>56</td>
</tr>
<tr>
<td>set_output/1</td>
<td>56</td>
</tr>
<tr>
<td>set_stream_position/2</td>
<td>57</td>
</tr>
<tr>
<td>setrand/1</td>
<td>124</td>
</tr>
<tr>
<td>shell/0</td>
<td>135</td>
</tr>
<tr>
<td>shell/1</td>
<td>135, 136</td>
</tr>
<tr>
<td>simple/1</td>
<td>41</td>
</tr>
<tr>
<td>skip/1</td>
<td>66</td>
</tr>
<tr>
<td>skip/2</td>
<td>67</td>
</tr>
<tr>
<td>sleep/1</td>
<td>136</td>
</tr>
<tr>
<td>socket/2</td>
<td>69</td>
</tr>
<tr>
<td>socket/4</td>
<td>69</td>
</tr>
<tr>
<td>socket_accept/2</td>
<td>70</td>
</tr>
<tr>
<td>socket_accept/3</td>
<td>70</td>
</tr>
<tr>
<td>socket_bind/2</td>
<td>69</td>
</tr>
<tr>
<td>socket_buffering/2</td>
<td>70</td>
</tr>
<tr>
<td>socket_close/1</td>
<td>69</td>
</tr>
<tr>
<td>socket_connect/3</td>
<td>70</td>
</tr>
<tr>
<td>socket_listen/2</td>
<td>70</td>
</tr>
<tr>
<td>socket_select/5</td>
<td>70</td>
</tr>
<tr>
<td>sort/2</td>
<td>47</td>
</tr>
<tr>
<td>source_file/1</td>
<td>104</td>
</tr>
<tr>
<td>source_file/2</td>
<td>104</td>
</tr>
<tr>
<td>source_location/2</td>
<td>104</td>
</tr>
<tr>
<td>spy/1</td>
<td>199</td>
</tr>
<tr>
<td>stream_position/2</td>
<td>58</td>
</tr>
<tr>
<td>stream_position_data/3</td>
<td>58</td>
</tr>
<tr>
<td>stream_select/3</td>
<td>56</td>
</tr>
<tr>
<td>sublist/2</td>
<td>111</td>
</tr>
<tr>
<td>subsumes/2</td>
<td>137</td>
</tr>
<tr>
<td>subsumes_chk/2</td>
<td>137</td>
</tr>
<tr>
<td>succ/3</td>
<td>53</td>
</tr>
<tr>
<td>suffix/2</td>
<td>111</td>
</tr>
<tr>
<td>sum_list/2</td>
<td>111</td>
</tr>
<tr>
<td>sum_list/3</td>
<td>111</td>
</tr>
<tr>
<td>sumlist/2</td>
<td>112</td>
</tr>
<tr>
<td>system/0</td>
<td>136</td>
</tr>
<tr>
<td>system/2</td>
<td>136</td>
</tr>
<tr>
<td>system_predicate/2</td>
<td>73</td>
</tr>
<tr>
<td>tab/1</td>
<td>66</td>
</tr>
<tr>
<td>tab/2</td>
<td>67</td>
</tr>
<tr>
<td>tell/1</td>
<td>58</td>
</tr>
<tr>
<td>telling/1</td>
<td>59</td>
</tr>
<tr>
<td>term_expansion/2</td>
<td>80</td>
</tr>
<tr>
<td>term_hash/2</td>
<td>136</td>
</tr>
<tr>
<td>term_hash/4</td>
<td>136</td>
</tr>
<tr>
<td>term_to_atom/2</td>
<td>132</td>
</tr>
<tr>
<td>term_variables/2</td>
<td>136</td>
</tr>
<tr>
<td>time_out/3</td>
<td>140</td>
</tr>
<tr>
<td>tmpnam/1</td>
<td>135</td>
</tr>
<tr>
<td>told/0</td>
<td>59</td>
</tr>
<tr>
<td>top_sort/2</td>
<td>143</td>
</tr>
<tr>
<td>top_sort/3</td>
<td>143</td>
</tr>
<tr>
<td>trace/0</td>
<td>200</td>
</tr>
<tr>
<td>transitive_closure/2</td>
<td>143</td>
</tr>
<tr>
<td>transpose/3</td>
<td>142</td>
</tr>
<tr>
<td>tree_size/2</td>
<td>141</td>
</tr>
<tr>
<td>tree_to_list/2</td>
<td>141</td>
</tr>
<tr>
<td>true/0</td>
<td>33</td>
</tr>
<tr>
<td>ttyget/1</td>
<td>68</td>
</tr>
<tr>
<td>ttyget0/1</td>
<td>68</td>
</tr>
<tr>
<td>ttyin/0</td>
<td>68</td>
</tr>
<tr>
<td>ttyout/1</td>
<td>68</td>
</tr>
<tr>
<td>ttyput/1</td>
<td>68</td>
</tr>
<tr>
<td>ttysock/1</td>
<td>68</td>
</tr>
<tr>
<td>ttytab/1</td>
<td>68</td>
</tr>
<tr>
<td>undgraph_del_vertices/3</td>
<td>146</td>
</tr>
<tr>
<td>unreliable/3</td>
<td>137</td>
</tr>
<tr>
<td>unify_with_occurs_check/2</td>
<td>42</td>
</tr>
<tr>
<td>unknown_predicate_handler/3</td>
<td>38</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>update_mutable/2</td>
<td>84</td>
</tr>
<tr>
<td>use_module/1</td>
<td>30</td>
</tr>
<tr>
<td>use_module/2</td>
<td>31</td>
</tr>
<tr>
<td>use_module/3</td>
<td>31</td>
</tr>
</tbody>
</table>

V

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>var/1</td>
<td>40</td>
</tr>
<tr>
<td>variant/2</td>
<td>137</td>
</tr>
<tr>
<td>verify_attributes/3</td>
<td>159</td>
</tr>
<tr>
<td>version/1</td>
<td>103</td>
</tr>
<tr>
<td>vertices/2</td>
<td>141</td>
</tr>
<tr>
<td>vertices_edges_to_ugraph/3</td>
<td>141</td>
</tr>
</tbody>
</table>

W

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait/2</td>
<td>136</td>
</tr>
<tr>
<td>when/2</td>
<td>154</td>
</tr>
<tr>
<td>with_output_to_chars/2</td>
<td>132</td>
</tr>
<tr>
<td>with_output_to_chars/3</td>
<td>132</td>
</tr>
<tr>
<td>with_output_to_chars/4</td>
<td>132</td>
</tr>
<tr>
<td>working_directory/2</td>
<td>135</td>
</tr>
<tr>
<td>write/1</td>
<td>60</td>
</tr>
<tr>
<td>write/2</td>
<td>66</td>
</tr>
<tr>
<td>write_canonical/1</td>
<td>66</td>
</tr>
<tr>
<td>write_canonical/2</td>
<td>66</td>
</tr>
<tr>
<td>write_term/2</td>
<td>60</td>
</tr>
<tr>
<td>write_term/3</td>
<td>66</td>
</tr>
<tr>
<td>write_to_chars/2</td>
<td>131</td>
</tr>
<tr>
<td>write_to_chars/3</td>
<td>131</td>
</tr>
<tr>
<td>writeq/1</td>
<td>61</td>
</tr>
<tr>
<td>writeq/2</td>
<td>66</td>
</tr>
</tbody>
</table>
24.2.3 YAP predicates not strictly compatible with SICStus Prolog

These are YAP built-ins that are also available in SICStus Prolog, but that are not fully compatible:

- [ ]/1
- []/1

abolish/1
abolish/2
assert/1
assert/2
asserta/1
asserta/2
assertz/1
assertz/2
atom_chars/2
bagof/3
clause/2
clause/3
close/2
current_line_number/1
current_line_number/2
debug/0
dynamic/1
erase/1
erased/1
format/1
format/2
format/3
instance/2
listing/0
nth_clause/3
nth_instance/3
nth_instance/4
number_chars/2
open/4
predicate_property/2
prolog_initialization/1
read_term/2
read_term/3
recorda/3
recordaifnot/3
recorded/3
recordz/3
retract/1
retract/2
retractall/1
setof/3
statistics/0
statistics/2
unknown/2
version/0
with_output_to/2
### 24.2.4 YAP predicates not available in SICStus Prolog

These are YAP built-ins not available in SICStus Prolog.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abolish_all_tables/0</td>
<td>194</td>
</tr>
<tr>
<td>abolish_table/1</td>
<td>194</td>
</tr>
<tr>
<td>add_to_array_element/4</td>
<td>91</td>
</tr>
<tr>
<td>alarm/3</td>
<td>82</td>
</tr>
<tr>
<td>all/3</td>
<td>79</td>
</tr>
<tr>
<td>always_prompt_user/0</td>
<td>69</td>
</tr>
<tr>
<td>append/3</td>
<td>149</td>
</tr>
<tr>
<td>array/2</td>
<td>90</td>
</tr>
<tr>
<td>array_element/3</td>
<td>91</td>
</tr>
<tr>
<td>assert_static/1</td>
<td>72</td>
</tr>
<tr>
<td>asserta_static/1</td>
<td>72</td>
</tr>
<tr>
<td>assertz_static/1</td>
<td>72</td>
</tr>
<tr>
<td>atom_concat/3</td>
<td>44</td>
</tr>
<tr>
<td>atom_length/2</td>
<td>43</td>
</tr>
<tr>
<td>atomic_concat/2</td>
<td>43</td>
</tr>
<tr>
<td>atomic_list_concat/2</td>
<td>43</td>
</tr>
<tr>
<td>atomic_list_concat/3</td>
<td>43</td>
</tr>
<tr>
<td>attr_portray_hook/2</td>
<td>157</td>
</tr>
<tr>
<td>attr_unify_hook/2</td>
<td>157</td>
</tr>
<tr>
<td>attribute_goals/2</td>
<td>157</td>
</tr>
<tr>
<td>attvar/1</td>
<td>156, 160</td>
</tr>
<tr>
<td>avl_insert/4</td>
<td>109</td>
</tr>
<tr>
<td>avl_lookup/3</td>
<td>109</td>
</tr>
<tr>
<td>avl_new/1</td>
<td>108</td>
</tr>
<tr>
<td>b_getval/2</td>
<td>84, 151</td>
</tr>
<tr>
<td>b_setval/2</td>
<td>84, 151</td>
</tr>
<tr>
<td>between/3</td>
<td>149</td>
</tr>
<tr>
<td>call/n</td>
<td>35</td>
</tr>
<tr>
<td>call_count_data/0</td>
<td>88</td>
</tr>
<tr>
<td>call_count_data/3</td>
<td>88</td>
</tr>
<tr>
<td>call_shared_object_function/2</td>
<td>220</td>
</tr>
<tr>
<td>call_with_args/n</td>
<td>35</td>
</tr>
<tr>
<td>catch/3</td>
<td>37</td>
</tr>
<tr>
<td>cd/1</td>
<td>81</td>
</tr>
<tr>
<td>char_type/2</td>
<td>45</td>
</tr>
<tr>
<td>chdir/1</td>
<td>149</td>
</tr>
<tr>
<td>checklist/2</td>
<td>114</td>
</tr>
<tr>
<td>checknodes/3</td>
<td>114</td>
</tr>
<tr>
<td>close_matlab/0</td>
<td>118</td>
</tr>
<tr>
<td>close_shared_object/1</td>
<td>220</td>
</tr>
<tr>
<td>close_static_array/1</td>
<td>91</td>
</tr>
<tr>
<td>code_type/2</td>
<td>46</td>
</tr>
<tr>
<td>compile/1 (directive)</td>
<td>22</td>
</tr>
<tr>
<td>compile_expressions/0</td>
<td>23</td>
</tr>
<tr>
<td>concat_atom/2</td>
<td>149</td>
</tr>
<tr>
<td>concat_atom/3</td>
<td>149</td>
</tr>
<tr>
<td>consult/1</td>
<td>21</td>
</tr>
<tr>
<td>convlist/3</td>
<td>114</td>
</tr>
<tr>
<td>copy_line/2</td>
<td>113</td>
</tr>
<tr>
<td>create_prolog_flag/2</td>
<td>103</td>
</tr>
<tr>
<td>current_foreign_library/2</td>
<td>130</td>
</tr>
<tr>
<td>current_mutex/3</td>
<td>189</td>
</tr>
<tr>
<td>current_prolog_flag/2</td>
<td>102</td>
</tr>
<tr>
<td>current_thread/2</td>
<td>183</td>
</tr>
<tr>
<td>del_attr/2</td>
<td>157</td>
</tr>
<tr>
<td>del_attr/1</td>
<td>157</td>
</tr>
<tr>
<td>dgraph_add_edge/4</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_add_edges/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_add_vertex/3</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_add_vertices/3</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_complement/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_compose/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_del_edge/4</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_del_edges/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_del_edges/4</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_del_edges/5</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_edges/2</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_isomorphic/4</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_max_path/5</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_min_path/5</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_min_paths/3</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_neighbors/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_neighbours/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_new/1</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_path/3</td>
<td>146</td>
</tr>
<tr>
<td>dgraph_symmetric_closure/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_to_digraph/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_to_undigraph/2</td>
<td>147</td>
</tr>
<tr>
<td>dgraph_top_sort/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_top_sort/3</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_transitive_closure/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_transpose/2</td>
<td>145</td>
</tr>
<tr>
<td>dgraph_vertices/2</td>
<td>144</td>
</tr>
<tr>
<td>do_not_compile_expressions/0</td>
<td>23</td>
</tr>
<tr>
<td>dynamic_predicate/2</td>
<td>71</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>matrix_set/3</td>
<td>116</td>
</tr>
<tr>
<td>matrix_set_all/2</td>
<td>116</td>
</tr>
<tr>
<td>matrix_shuffle/3</td>
<td>117</td>
</tr>
<tr>
<td>matrix_size/2</td>
<td>116</td>
</tr>
<tr>
<td>matrix_sum/2</td>
<td>117</td>
</tr>
<tr>
<td>matrix_to_list/2</td>
<td>116</td>
</tr>
<tr>
<td>matrix_type/2</td>
<td>116</td>
</tr>
<tr>
<td>message_queue_create/1</td>
<td>185</td>
</tr>
<tr>
<td>message_queue_destroy/1</td>
<td>185</td>
</tr>
<tr>
<td>message_to_string/2</td>
<td>40</td>
</tr>
<tr>
<td>mktime/2</td>
<td>133</td>
</tr>
<tr>
<td>mpi_barrier/0</td>
<td>148</td>
</tr>
<tr>
<td>mpi_bcast/2</td>
<td>148</td>
</tr>
<tr>
<td>mpi_bcast/3</td>
<td>148</td>
</tr>
<tr>
<td>mpi_comm_rank/1</td>
<td>147</td>
</tr>
<tr>
<td>mpi_comm_size/1</td>
<td>147</td>
</tr>
<tr>
<td>mpi_finalize/0</td>
<td>147</td>
</tr>
<tr>
<td>mpi_gc/0</td>
<td>148</td>
</tr>
<tr>
<td>mpi_init/0</td>
<td>147</td>
</tr>
<tr>
<td>mpi_irecv/3</td>
<td>148</td>
</tr>
<tr>
<td>mpi_isend/4</td>
<td>147</td>
</tr>
<tr>
<td>mpi_recv/3</td>
<td>147</td>
</tr>
<tr>
<td>mpi_send/3</td>
<td>147</td>
</tr>
<tr>
<td>mpi_test/2</td>
<td>148</td>
</tr>
<tr>
<td>mpi_test_recv/3</td>
<td>148</td>
</tr>
<tr>
<td>mpi_version/2</td>
<td>147</td>
</tr>
<tr>
<td>mpi_wait/2</td>
<td>148</td>
</tr>
<tr>
<td>mpi_wait_recv/3</td>
<td>148</td>
</tr>
<tr>
<td>mutex_create/1</td>
<td>188</td>
</tr>
<tr>
<td>mutex_destroy/1</td>
<td>188</td>
</tr>
<tr>
<td>mutex_lock/1</td>
<td>188</td>
</tr>
<tr>
<td>mutex_statistics/0</td>
<td>184</td>
</tr>
<tr>
<td>mutex_trylock/1</td>
<td>188</td>
</tr>
<tr>
<td>mutex_unlock/1</td>
<td>189</td>
</tr>
<tr>
<td>mutex_unlock_all/0</td>
<td>189</td>
</tr>
<tr>
<td>nb_beam/1</td>
<td>121</td>
</tr>
<tr>
<td>nb_beam_add/3</td>
<td>121</td>
</tr>
<tr>
<td>nb_beam_close/1</td>
<td>121</td>
</tr>
<tr>
<td>nb_beam_del/3</td>
<td>121</td>
</tr>
<tr>
<td>nb_beam_empty/1</td>
<td>121</td>
</tr>
<tr>
<td>nb_beam_poke/3</td>
<td>121</td>
</tr>
<tr>
<td>nb_beam_size/2</td>
<td>121</td>
</tr>
<tr>
<td>nb_current/2</td>
<td>86, 152</td>
</tr>
<tr>
<td>nb_delete/1</td>
<td>152</td>
</tr>
<tr>
<td>nb_delete/2</td>
<td>86</td>
</tr>
<tr>
<td>nb_getval/2</td>
<td>85, 151</td>
</tr>
<tr>
<td>nb_heap/1</td>
<td>120</td>
</tr>
<tr>
<td>nb_heap_add/3</td>
<td>120</td>
</tr>
<tr>
<td>nb_heap_close/1</td>
<td>120</td>
</tr>
<tr>
<td>nb_heap_del/3</td>
<td>120</td>
</tr>
<tr>
<td>nb_heap_empty/1</td>
<td>121</td>
</tr>
<tr>
<td>nb_heap_poke/3</td>
<td>120</td>
</tr>
<tr>
<td>nb_heap_size/2</td>
<td>120</td>
</tr>
<tr>
<td>nb_linkval/2</td>
<td>85</td>
</tr>
<tr>
<td>nb_linkarg/3</td>
<td>86</td>
</tr>
<tr>
<td>nb_queue/1</td>
<td>120</td>
</tr>
<tr>
<td>nb_queue_close/3</td>
<td>120</td>
</tr>
<tr>
<td>nb_queue_deinit/2</td>
<td>120</td>
</tr>
<tr>
<td>nb_queue_empty/1</td>
<td>120</td>
</tr>
<tr>
<td>nb_queue_enqueue/2</td>
<td>120</td>
</tr>
<tr>
<td>nb_queue_peek/2</td>
<td>120</td>
</tr>
<tr>
<td>nb_queue_size/2</td>
<td>120</td>
</tr>
<tr>
<td>nb_set_shared_arg/3</td>
<td>86</td>
</tr>
<tr>
<td>nb_set_shared_val/2</td>
<td>85</td>
</tr>
<tr>
<td>nb_setarg/3</td>
<td>86</td>
</tr>
<tr>
<td>nb_setval/2</td>
<td>85, 151</td>
</tr>
<tr>
<td>new_variables_in_term/3</td>
<td>137</td>
</tr>
<tr>
<td>no_source/0</td>
<td>23</td>
</tr>
<tr>
<td>not/1</td>
<td>34</td>
</tr>
<tr>
<td>nthl/3</td>
<td>149</td>
</tr>
<tr>
<td>number_atom/2</td>
<td>44</td>
</tr>
<tr>
<td>on_signal/3</td>
<td>83</td>
</tr>
<tr>
<td>once/1</td>
<td>36</td>
</tr>
<tr>
<td>open_shared_object/2</td>
<td>219</td>
</tr>
<tr>
<td>open_shared_object/3</td>
<td>220</td>
</tr>
<tr>
<td>partition/4</td>
<td>114</td>
</tr>
<tr>
<td>partition/5</td>
<td>115</td>
</tr>
<tr>
<td>path/1</td>
<td>24</td>
</tr>
<tr>
<td>peek_byte/1</td>
<td>66</td>
</tr>
<tr>
<td>peek_byte/2</td>
<td>67</td>
</tr>
<tr>
<td>peek_char/2</td>
<td>67</td>
</tr>
<tr>
<td>peek_code/1</td>
<td>66</td>
</tr>
<tr>
<td>peek_code/2</td>
<td>67</td>
</tr>
<tr>
<td>presort/3</td>
<td>47</td>
</tr>
<tr>
<td>process/2</td>
<td>113</td>
</tr>
<tr>
<td>profile_data/3</td>
<td>87</td>
</tr>
<tr>
<td>profiled_reset/0</td>
<td>88</td>
</tr>
<tr>
<td>profinit/0</td>
<td>88</td>
</tr>
<tr>
<td>profoff/0</td>
<td>88</td>
</tr>
<tr>
<td>profon/0</td>
<td>88</td>
</tr>
<tr>
<td>public/1 (directive)</td>
<td>25</td>
</tr>
<tr>
<td>put_attr/3</td>
<td>156</td>
</tr>
<tr>
<td>put_attr/2</td>
<td>157</td>
</tr>
<tr>
<td>put_byte/1</td>
<td>65</td>
</tr>
<tr>
<td>put_byte/1</td>
<td>67</td>
</tr>
<tr>
<td>put_char/1</td>
<td>65</td>
</tr>
<tr>
<td>put_char/2</td>
<td>67</td>
</tr>
<tr>
<td>put_code/1</td>
<td>65</td>
</tr>
<tr>
<td>put_code/2</td>
<td>67</td>
</tr>
<tr>
<td>putenv/2</td>
<td>81</td>
</tr>
<tr>
<td>rannum/1</td>
<td>122</td>
</tr>
<tr>
<td>ranstart/0</td>
<td>123</td>
</tr>
<tr>
<td>ranstart/1</td>
<td>123</td>
</tr>
<tr>
<td>ranunif/2</td>
<td>123</td>
</tr>
</tbody>
</table>
24.3 Compatibility with the ISO Prolog standard

The Prolog standard was developed by ISO/IEC JTC1/SC22/WG17, the international standardization working group for the programming language Prolog. The book "Prolog: The Standard" by Deransart, Ed-Dbali and Cervoni gives a complete description of this standard. Development in YAP from YAP4.1.6 onwards have striven at making YAP compatible with ISO Prolog. As such:

- YAP now supports all of the built-ins required by the ISO-standard, and,
- Error-handling is as required by the standard.

YAP by default is not fully ISO standard compliant. You can set the language flag to iso to obtain very good compatibility. Setting this flag changes the following:

- By default, YAP uses "immediate update semantics" for its database, and not "logical update semantics", as per the standard, (see Section 24.2 [SICStus Prolog], page 235). This affects assert/1, retract/1, and friends. Calling set_prolog_flag(update_semantics,logical) will switch YAP to use logical update semantics.
- By default, YAP implements the atom_chars/2 (see Section 6.4 [Testing Terms], page 40), and number_chars/2, (see Section 6.4 [Testing Terms], page 40), built-ins as per the original Quintus Prolog definition, and not as per the ISO definition.
Calling `set_prolog_flag(to_chars_mode,iso)` will switch YAP to use the ISO definition for `atom_chars/2` and `number_chars/2`.

- By default, YAP fails on undefined predicates. To follow the ISO Prolog standard use:
  ```prolog
  :- set_prolog_flag(unknown,error).
  ```
- By default, YAP allows executable goals in directives. In ISO mode most directives can only be called from top level (the exceptions are `set_prolog_flag/2` and `op/3`).
- Error checking for meta-calls under ISO Prolog mode is stricter than by default.
- The `strict_iso` flag automatically enables the ISO Prolog standard. This feature should disable all features not present in the standard.

The following incompatibilities between YAP and the ISO standard are known to still exist:

- Currently, YAP does not handle overflow errors in integer operations, and handles floating-point errors only in some architectures. Otherwise, YAP follows IEEE arithmetic.

Please inform the authors on other incompatibilities that may still exist.
Appendix A: Summary of YAP Predefined Operators

The Prolog syntax caters for operators of three main kinds:

- prefix;
- infix;
- postfix.

Each operator has precedence in the range 1 to 1200, and this precedence is used to disambiguate expressions where the structure of the term denoted is not made explicit using brackets. The operator of higher precedence is the main functor.

If there are two operators with the highest precedence, the ambiguity is solved analyzing the types of the operators. The possible infix types are: xfx, xfy, and yfx.

With an operator of type xfx both sub-expressions must have lower precedence than the operator itself, unless they are bracketed (which assigns to them zero precedence). With an operator type xfy only the left-hand sub-expression must have lower precedence. The opposite happens for yfx type.

A prefix operator can be of type fx or fy. A postfix operator can be of type xf or yf. The meaning of the notation is analogous to the above.

\[ a + b * c \]

means

\[ a + (b * c) \]
as + and * have the following types and precedences:

\[-op(500, yfx, \'+\').\]
\[-op(400, yfx, \'*\').\]

Now defining

\[-op(700, xfy, '++').\]
\[-op(700, xfx, '=':=').\]
a ++ b =:= c

means

\[ a ++ (b =:= c) \]

The following is the list of the declarations of the predefined operators:

\[-op(1200, fx, [?\?-', ':-']).
- op(1200, xfx, [':-', '->']).
- op(1150, fx, [block, dynamic, mode, public, multifile, meta_predicate, sequential, table, initialization]).
- op(1100, xfy, [';', '|']).
- op(1050, xfy, '->').
- op(999, xfy, ').'
- op(900, fy, ['\', not]).
- op(900, fx, [nospy, spy]).
- op(700, xfx, [@>=, @=<, @<, @>, <, =, >, =:=, =\=, =\=, =\=, @>, @<, @<, @>, <, =, >, =:=, =\=, =\=, =\=, @>, @<, @<, @>, <, =, >, =:=, =\=, =\=, =\=, @>, @<, @<, @>, <, =, >, =:=, =\=, =\=, =\=, @>, @<, @<, @>, <, =, >]).
- op(500, yfx, ['\'/', '\\', '+', '-']).\]
:-op(500,fx,[’+’,’-’]).
:-op(400,yfx,[’<<’,’>>’,’//’,’*’,’/’]).
:-op(300,xfx,mod).
:-op(200,xfy,[’-’,’**’]).
:-op(50,xfx,same).
Predicate Index

A
abolish/1................................. 72
abolish/2................................. 72
abolish_all_tables/0..................... 194
abolish_table/1......................... 194
abort/0.................................. 36
absolute_file_name/2..................... 56
absolute_file_name/3..................... 55
add_edges/3.............................. 142
add_to_array_element/4.................. 91
add_to_heap/4............................ 109
add_to_path/1............................ 24
add_to_path/2............................ 24
add_vertices/3........................... 142
agc_margin (yap_flag/2 option)......... 94
aggregate/3.............................. 106
aggregate/4.............................. 106
aggregate_all/3......................... 106
aggregate_all/4......................... 106
alarm/3.................................. 82
all/3...................................... 79
always_prompt_user/0.................... 69
append/2................................ 110
append/3.................................. 110, 149
arg/3.................................... 41
argv (yap_flag/2 option)................ 94
array/2.................................. 90
array_element/3.......................... 91
assert/1.................................. 71
assert/2.................................. 74
assert_static/1.......................... 72
assertta/1............................... 72
assertta/2............................... 74
assertta_static/1......................... 72
asserttz/1............................... 72
asserttz/2............................... 74
asserttz_static/1......................... 72
assoc_to_list/2.......................... 107
at_end_of_stream/0...................... 57
at_end_of_stream/1...................... 57
atom/1.................................... 40
atom_chars/2............................. 43
atom_codes/2............................. 43
atom_concat/2............................ 43
atom_concat/3............................ 44
atom_length/2............................ 43
atom_number/2............................ 44
atom_to_chars/2.......................... 131
atom_to_chars/3.......................... 131
atom_to_term/3........................... 131
atomic/1.................................. 40
atomic_concat/2.......................... 43
atomic_list_concat/2...................... 43
atomic_list_concat/3...................... 43
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoms (statistics/2 option)</td>
<td>93</td>
</tr>
<tr>
<td>attr_portray_hook/2</td>
<td>157</td>
</tr>
<tr>
<td>attr_unify_hook/2</td>
<td>157</td>
</tr>
<tr>
<td>attribute/1 (declaration)</td>
<td>158</td>
</tr>
<tr>
<td>attribute_goal/2</td>
<td>160</td>
</tr>
<tr>
<td>attribute_goals/2</td>
<td>157</td>
</tr>
<tr>
<td>attrvar/1</td>
<td>156, 160</td>
</tr>
<tr>
<td>avl_insert/4</td>
<td>109</td>
</tr>
<tr>
<td>avl_lookup/3</td>
<td>109</td>
</tr>
<tr>
<td>avl_new/1</td>
<td>108</td>
</tr>
<tr>
<td>b_getval/2</td>
<td>84, 151</td>
</tr>
<tr>
<td>b_setval/2</td>
<td>84, 151</td>
</tr>
<tr>
<td>bagof/3</td>
<td>79</td>
</tr>
<tr>
<td>bb_delete/2</td>
<td>78</td>
</tr>
<tr>
<td>bb_get/2</td>
<td>78</td>
</tr>
<tr>
<td>bb_put/2</td>
<td>78</td>
</tr>
<tr>
<td>bb_update/3</td>
<td>78</td>
</tr>
<tr>
<td>between/3</td>
<td>53, 149</td>
</tr>
<tr>
<td>bounded (yap_flag/2 option)</td>
<td>94</td>
</tr>
<tr>
<td>break/0</td>
<td>36</td>
</tr>
<tr>
<td>C/3</td>
<td>80</td>
</tr>
<tr>
<td>call/1</td>
<td>35</td>
</tr>
<tr>
<td>call/2</td>
<td>35</td>
</tr>
<tr>
<td>call_cleanup/1</td>
<td>138</td>
</tr>
<tr>
<td>call_cleanup/2</td>
<td>138</td>
</tr>
<tr>
<td>call_count_data/0</td>
<td>88</td>
</tr>
<tr>
<td>call_count_data/3</td>
<td>88</td>
</tr>
<tr>
<td>call_counting (yap_flag/2 option)</td>
<td>94</td>
</tr>
<tr>
<td>call_residue/2</td>
<td>154</td>
</tr>
<tr>
<td>call_residue_vars/2</td>
<td>154</td>
</tr>
<tr>
<td>call_shared_object_function/2</td>
<td>220</td>
</tr>
<tr>
<td>call_with_args/n</td>
<td>35</td>
</tr>
<tr>
<td>callable/1</td>
<td>41</td>
</tr>
<tr>
<td>catch/3</td>
<td>37</td>
</tr>
<tr>
<td>cd/1</td>
<td>81</td>
</tr>
<tr>
<td>char_code/2</td>
<td>45</td>
</tr>
<tr>
<td>char_conversion (yap_flag/2 option)</td>
<td>95</td>
</tr>
<tr>
<td>char_conversion/2</td>
<td>60</td>
</tr>
<tr>
<td>char_type/2</td>
<td>45</td>
</tr>
<tr>
<td>character_count/2</td>
<td>58</td>
</tr>
<tr>
<td>character_escapes (yap_flag/2 option)</td>
<td>95</td>
</tr>
<tr>
<td>chdir/1</td>
<td>149</td>
</tr>
<tr>
<td>checklist/2</td>
<td>114</td>
</tr>
<tr>
<td>checknodes/3</td>
<td>114</td>
</tr>
<tr>
<td>clause/2</td>
<td>72</td>
</tr>
<tr>
<td>clause/3</td>
<td>72</td>
</tr>
<tr>
<td>cleanup_all/0</td>
<td>140</td>
</tr>
<tr>
<td>close/1</td>
<td>55</td>
</tr>
<tr>
<td>close/2</td>
<td>55</td>
</tr>
<tr>
<td>close_matlab/0</td>
<td>118</td>
</tr>
<tr>
<td>close_shared_object/1</td>
<td>220</td>
</tr>
<tr>
<td>close_static_array/1</td>
<td>91</td>
</tr>
<tr>
<td>code_type/2</td>
<td>46</td>
</tr>
<tr>
<td>compare/3</td>
<td>46</td>
</tr>
<tr>
<td>compile/1</td>
<td>21</td>
</tr>
<tr>
<td>compile_expressions/0</td>
<td>23</td>
</tr>
<tr>
<td>complement/2</td>
<td>143</td>
</tr>
<tr>
<td>compose/3</td>
<td>143</td>
</tr>
<tr>
<td>compound/1</td>
<td>40</td>
</tr>
<tr>
<td>concat_atom/2</td>
<td>149</td>
</tr>
<tr>
<td>concat_atom/3</td>
<td>149</td>
</tr>
<tr>
<td>consult/1</td>
<td>21</td>
</tr>
<tr>
<td>convlist/3</td>
<td>114</td>
</tr>
<tr>
<td>copy_line/2</td>
<td>113</td>
</tr>
<tr>
<td>copy_term/2</td>
<td>42</td>
</tr>
<tr>
<td>copy_term/3</td>
<td>157</td>
</tr>
<tr>
<td>copy_term_nat/2</td>
<td>158</td>
</tr>
<tr>
<td>cputime (statistics/2 option)</td>
<td>93</td>
</tr>
<tr>
<td>create_mutable/2</td>
<td>83</td>
</tr>
<tr>
<td>create_prolog_flag/2</td>
<td>103</td>
</tr>
<tr>
<td>current_atom/1</td>
<td>73</td>
</tr>
<tr>
<td>current_char_conversion/2</td>
<td>60</td>
</tr>
<tr>
<td>current_foreign_library/2</td>
<td>130</td>
</tr>
<tr>
<td>current_input/1</td>
<td>57</td>
</tr>
<tr>
<td>current_key/2</td>
<td>76</td>
</tr>
<tr>
<td>current_line_number/1</td>
<td>58</td>
</tr>
<tr>
<td>current_line_number/2</td>
<td>58</td>
</tr>
<tr>
<td>current_module/1</td>
<td>92</td>
</tr>
<tr>
<td>current_module/2</td>
<td>92</td>
</tr>
<tr>
<td>current_mutex/3</td>
<td>189</td>
</tr>
<tr>
<td>current_op/3</td>
<td>103</td>
</tr>
<tr>
<td>current_output/1</td>
<td>57</td>
</tr>
<tr>
<td>current_predicate/1</td>
<td>73</td>
</tr>
<tr>
<td>current_predicate/2</td>
<td>73</td>
</tr>
<tr>
<td>current_prolog_flag/2</td>
<td>102</td>
</tr>
<tr>
<td>current_stream/3</td>
<td>56</td>
</tr>
<tr>
<td>current_thread/2</td>
<td>183</td>
</tr>
<tr>
<td>cyclic_term/1</td>
<td>136</td>
</tr>
<tr>
<td>D/3</td>
<td>133</td>
</tr>
<tr>
<td>db_reference/1C</td>
<td>40</td>
</tr>
<tr>
<td>debug (yap_flag/2 option)</td>
<td>95</td>
</tr>
<tr>
<td>debug/0</td>
<td>199</td>
</tr>
<tr>
<td>debugger_print_options (yap_flag/2 option)</td>
<td>95</td>
</tr>
<tr>
<td>debugging/0</td>
<td>199</td>
</tr>
<tr>
<td>del_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_attr/2</td>
<td>157</td>
</tr>
<tr>
<td>del_atrs/1</td>
<td>157</td>
</tr>
<tr>
<td>del_edges/3</td>
<td>142</td>
</tr>
<tr>
<td>del_max_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_min_assoc/4</td>
<td>107</td>
</tr>
<tr>
<td>del_vertices/3</td>
<td>142</td>
</tr>
<tr>
<td>delete/3</td>
<td>110</td>
</tr>
<tr>
<td>delete_file/1</td>
<td>133</td>
</tr>
<tr>
<td>delete_file/2</td>
<td>133</td>
</tr>
<tr>
<td>dgraph_add_edge/4</td>
<td>144</td>
</tr>
<tr>
<td>dgraph_add_edges/3</td>
<td>145</td>
</tr>
</tbody>
</table>
dgraph_add_vertex/3 .......................... 144
dgraph_add_vertices/3 ........................ 144
dgraph_complement/2 .......................... 145
dgraph_compose/3 .............................. 145
dgraph_del_edge/4 .............................. 145
dgraph_del_edges/3 ............................. 145
dgraph_del_vertex/3 ........................... 144
dgraph_del_vertices/3 ......................... 144
dgraph_edge/2 ................................. 144
dgraph_edges/2 ................................ 144
dgraph_isomorphic/4 ........................... 146
dgraph_max_path/5 .............................. 146
dgraph_min_path/5 .............................. 146
dgraph_min_paths/3 ............................. 146
dgraph_neighbors/3 ............................. 145
dgraph_neighbours/3 ........................... 145
dgraph_new/1 ................................. 144
dgraph_path/3 ................................. 146
dgraph_symmetric_closure/2 ................. 145
dgraph_to_digraph/2 .......................... 145
dgraph_to_undigraph/2 ......................... 147
dgraph_top_sort/2 ............................. 145
dgraph_top_sort/3 ............................. 144
dgraph_transpose/2 ............................ 145
dgraph_vertices/2 ............................. 144
dialect (yap_flag/2 option) .................. 95
dif/2 ........................................ 153
directory (prolog_load_context/2 option) .... 103
directory_files/2 .............................. 133
discontiguous/1 (directive) .................. 25
discontiguous_warnings (yap_flag/2 option) . 95

E
edges/2 ........................................ 141
e1if/1 directive .............................. 26
e2lso/0 directive .............................. 26
empty_assoc/1 ............................... 107
empty_heap/1 ................................ 109
empty_queue/1 ............................... 123
endif/0 directive ............................ 26
ensure_loaded/1 ............................... 22
envir/2 ...................................... 81, 134
erase/1 ...................................... 75
eraseall/1 ................................... 76
erased/1 ..................................... 75
exception/3 .................................. 38
exclude/3 .................................... 114
exec/3 ....................................... 135
executable(yap_flag/2 option) ............... 95
eexists/1 ..................................... 68
expand_exprs/2 ................................ 24
expand_terms/2 ................................ 80

F
fail/0 ........................................... 33
false/0 ........................................ 33
fast (yap_flag/2 option) ...................... 95
fields/2 ....................................... 113
fields/3 ....................................... 113
file (prolog_load_context/2 option) ........ 103, 104
file_base_name/2 .............................. 56
file_exists/1 ................................ 133
file_exists/2 ................................ 133
file_filter/3 .................................. 113
file_filter_with_init/5 ........................ 113
file_name_extension/3 ........................ 56
file_property/2 ................................ 134
file_search_path/2 ............................ 25
fileerrors (yap_flag/2 option) ............... 96
fileerrors/0 ................................. 68
filter/3 ....................................... 113
findall/3 ..................................... 78
findall/4 ..................................... 78
flatten/2 ..................................... 110
float/1 ....................................... 40
float_format (yap_flag/2 option) ............ 96
flush_output/0 ............................... 56
flush_output/1 ............................... 56
forall/2 ...................................... 36, 150
foreach/2 ..................................... 106
format/1 ...................................... 64
format/2 ...................................... 61
format/3 ...................................... 64
format_to_chars/3 ............................ 131
format_to_chars/4 ............................ 131
fragile ........................................ 138
free_variables/4 ................................ 106
freeze/2 ...................................... 153
frozen/2 ...................................... 154
functor/3 ..................................... 41

G
garbage_col/0 ................................. 37
garbage_collect_atoms/0 ..................... 37
garbage_collection (statistics/2 option) ... 93
gc (yap_flag/2 option) ....................... 96
gc/0 .......................................... 37
gc_margin (yap_flag/2 option) ............... 96
gc_trace (yap_flag/2 option) ............... 96
gen_assoc/3 ................................. 107
max_arity (yap_flag/2 option) ........................................... 97
max_assoc/3 ................................................................. 108
max_integer (yap_flag/2 option) ........................................ 97
max_list/2 ................................................................. 112
max_tagged_integer (yap_flag/2 option) ............................. 97
max_threads (yap_flag/2 option) ....................................... 102
max_workers (yap_flag/2 option) ....................................... 102
member/2 ................................................................. 110
memberchk/2 ............................................................. 110
merge/3 ................................................................. 121
message_hook/3 ....................................................... 120
message_queue_create/1 ................................................. 185
message_queue_destroy/1 .............................................. 185
message_to_string/2 ................................................... 120
meta_predicate/1 (directive) ........................................... 31
min_assoc/3 ............................................................... 108
min_integer (yap_flag/2 option) ........................................ 97
min_list/2 ............................................................... 112
min_of_heap/3 .......................................................... 109
min_of_heap/5 .......................................................... 109
module (prolog_load_context/2 option) .............................. 103
module/1 ................................................................. 109
module/2 (directive) ..................................................... 30
module/3 (directive) ..................................................... 30
module_transparent/1 (directive) ..................................... 32
mpi_barrier/0 ............................................................ 148
mpi_bcast/2 ............................................................. 148
mpi_bcast/3 ............................................................. 148
mpi_comm_rank/1 ........................................................ 147
mpi_comm_size/1 ........................................................ 147
mpi_finalize/0 ........................................................... 147
mpi_gc/0 ................................................................. 148
mpi_init/0 ............................................................... 147
mpi_irecv/3 ............................................................. 148
mpi_isend/4 ............................................................. 147
mpi_recv/3 .............................................................. 148
mpi_send/3 .............................................................. 147
mpi_test/2 ............................................................... 148
mpi_test_recv/3 ........................................................ 148
mpi_version/2 ........................................................... 147
mpi_wait/2 ............................................................... 148
mpi_wait_recv/3 ........................................................ 148
multifile/1 (directive) .................................................. 24
mutex_create/1 .......................................................... 188
mutex_destroy/1 ........................................................ 188
mutex_lock/1 ............................................................ 188
mutex_statistics/0 ....................................................... 184
mutex_trylock/1 ........................................................ 188
mutex_unlock/1 .......................................................... 189
mutex_unlock_all/0 ...................................................... 189

N

n_of_integer_keys_in_bb (yap_flag/2 option) ....................... 97
n_of_integer_keys_in_db (yap_flag/2 option) ....................... 97
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate_property/2</td>
<td>73</td>
</tr>
<tr>
<td>predsort/3</td>
<td>47</td>
</tr>
<tr>
<td>primitive/1</td>
<td>41</td>
</tr>
<tr>
<td>print/1</td>
<td>61</td>
</tr>
<tr>
<td>print/2</td>
<td>66</td>
</tr>
<tr>
<td>print_message/2</td>
<td>39</td>
</tr>
<tr>
<td>print_message_lines/3</td>
<td>40</td>
</tr>
<tr>
<td>process/2</td>
<td>113</td>
</tr>
<tr>
<td>profile_data/3</td>
<td>87</td>
</tr>
<tr>
<td>profiled_reset/0</td>
<td>88</td>
</tr>
<tr>
<td>profiling (yap_flag/2 option)</td>
<td>98</td>
</tr>
<tr>
<td>profinit/0</td>
<td>88</td>
</tr>
<tr>
<td>profoff/0</td>
<td>88</td>
</tr>
<tr>
<td>profon/0</td>
<td>88</td>
</tr>
<tr>
<td>program (statistics/2 option)</td>
<td>93</td>
</tr>
<tr>
<td>project_attributes/2</td>
<td>160</td>
</tr>
<tr>
<td>prolog_file_name/2</td>
<td>25</td>
</tr>
<tr>
<td>prolog_flag/3</td>
<td>102</td>
</tr>
<tr>
<td>prolog_initialization/1</td>
<td>103</td>
</tr>
<tr>
<td>prolog_load_context/2</td>
<td>103</td>
</tr>
<tr>
<td>prompt/2</td>
<td>103</td>
</tr>
<tr>
<td>prompt_alternatives_on (yap_flag/2 option)</td>
<td>98</td>
</tr>
<tr>
<td>public/1 (directive)</td>
<td>25</td>
</tr>
<tr>
<td>put/1</td>
<td>65</td>
</tr>
<tr>
<td>put/2</td>
<td>67</td>
</tr>
<tr>
<td>put_assoc/4</td>
<td>108</td>
</tr>
<tr>
<td>put_attr/3</td>
<td>156</td>
</tr>
<tr>
<td>put_atrs/2</td>
<td>157</td>
</tr>
<tr>
<td>put_atts/2</td>
<td>159</td>
</tr>
<tr>
<td>put_byte/1</td>
<td>65</td>
</tr>
<tr>
<td>put_byte/2</td>
<td>67</td>
</tr>
<tr>
<td>put_char/1</td>
<td>65</td>
</tr>
<tr>
<td>put_char/2</td>
<td>67</td>
</tr>
<tr>
<td>put_code/1</td>
<td>65</td>
</tr>
<tr>
<td>put_code/2</td>
<td>67</td>
</tr>
<tr>
<td>put_label/4</td>
<td>141</td>
</tr>
<tr>
<td>putenv/2</td>
<td>81</td>
</tr>
<tr>
<td>same_length/2</td>
<td>111</td>
</tr>
<tr>
<td>random/1</td>
<td>124</td>
</tr>
<tr>
<td>random/3</td>
<td>124</td>
</tr>
<tr>
<td>randseq/3</td>
<td>124</td>
</tr>
<tr>
<td>randset/3</td>
<td>124</td>
</tr>
<tr>
<td>ranum/1</td>
<td>122</td>
</tr>
<tr>
<td>ranstart/0</td>
<td>123</td>
</tr>
<tr>
<td>ranstart/1</td>
<td>123</td>
</tr>
<tr>
<td>ranunif/2</td>
<td>123</td>
</tr>
<tr>
<td>rational/1</td>
<td>40</td>
</tr>
<tr>
<td>rb_apply/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_clone/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_del_max/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_del_min/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_del/3</td>
<td>125</td>
</tr>
<tr>
<td>rb_del/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_empty/1</td>
<td>125</td>
</tr>
<tr>
<td>rb_insert/4</td>
<td>125</td>
</tr>
<tr>
<td>rb_keys/2</td>
<td>126</td>
</tr>
<tr>
<td>rb_lookup/3</td>
<td>125</td>
</tr>
<tr>
<td>rb_lookupall/3</td>
<td>125</td>
</tr>
<tr>
<td>rb_map/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_max/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_min/3</td>
<td>126</td>
</tr>
<tr>
<td>rb_new/1</td>
<td>125</td>
</tr>
<tr>
<td>rb_next/4</td>
<td>127</td>
</tr>
<tr>
<td>rb_partial_map/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_previous/4</td>
<td>127</td>
</tr>
<tr>
<td>rb_size/2</td>
<td>126</td>
</tr>
<tr>
<td>rb_update/4</td>
<td>126</td>
</tr>
<tr>
<td>rb_visit/2</td>
<td>126</td>
</tr>
<tr>
<td>reachable/3</td>
<td>144</td>
</tr>
<tr>
<td>read/1</td>
<td>59</td>
</tr>
<tr>
<td>read/2</td>
<td>66</td>
</tr>
<tr>
<td>read_file_to_codes/3</td>
<td>125</td>
</tr>
<tr>
<td>read_file_to_terms/3</td>
<td>125</td>
</tr>
<tr>
<td>read_from_chars/2</td>
<td>132</td>
</tr>
<tr>
<td>read_line_to_codes/2</td>
<td>124</td>
</tr>
<tr>
<td>read_line_to_codes/3</td>
<td>124</td>
</tr>
<tr>
<td>read_stream_to_codes/3</td>
<td>125</td>
</tr>
<tr>
<td>read_term/2</td>
<td>59</td>
</tr>
<tr>
<td>read_term/3</td>
<td>66</td>
</tr>
<tr>
<td>reconsult/1</td>
<td>21</td>
</tr>
<tr>
<td>records/3</td>
<td>75</td>
</tr>
<tr>
<td>recorda/3</td>
<td>75</td>
</tr>
<tr>
<td>recorda_at/3</td>
<td>75</td>
</tr>
<tr>
<td>recordaifnot/3</td>
<td>75, 77</td>
</tr>
<tr>
<td>recorded/3</td>
<td>75</td>
</tr>
<tr>
<td>recordz/3</td>
<td>75</td>
</tr>
<tr>
<td>recordz_at/3</td>
<td>75</td>
</tr>
<tr>
<td>recordzifnot/3</td>
<td>76</td>
</tr>
<tr>
<td>reexport/1</td>
<td>32</td>
</tr>
<tr>
<td>reexport/2</td>
<td>32</td>
</tr>
<tr>
<td>regexp/3</td>
<td>127</td>
</tr>
<tr>
<td>regexp/4</td>
<td>128</td>
</tr>
<tr>
<td>remove_duplicates/2</td>
<td>111</td>
</tr>
<tr>
<td>remove_from_path/1</td>
<td>24</td>
</tr>
<tr>
<td>rename/2</td>
<td>81</td>
</tr>
<tr>
<td>rename_file/2</td>
<td>134</td>
</tr>
<tr>
<td>repeat/0</td>
<td>34</td>
</tr>
<tr>
<td>reset_op_counters/0</td>
<td>197</td>
</tr>
<tr>
<td>reset_static_array/1</td>
<td>90</td>
</tr>
<tr>
<td>resize_static_array/3</td>
<td>91</td>
</tr>
<tr>
<td>restore/1</td>
<td>27</td>
</tr>
<tr>
<td>retract/1</td>
<td>73</td>
</tr>
<tr>
<td>retract/2</td>
<td>74</td>
</tr>
<tr>
<td>retractall/1</td>
<td>73</td>
</tr>
<tr>
<td>reverse/2</td>
<td>111</td>
</tr>
<tr>
<td>runtime (statistics/2 option)</td>
<td>93</td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>same_length/2</td>
<td>111</td>
</tr>
</tbody>
</table>
save/1 ........................................... 26
save/2 ........................................... 26
save_program/1 .................................. 26
save_program/2 .................................. 27
scan_integer/3 .................................. 112
scan_natural/3 .................................. 112
search_for/2 .................................... 112
see/1 ........................................... 59
seeing/1 ......................................... 59
seen/0 ........................................... 59
select/3 ......................................... 111
selectchk/3 ...................................... 111
selectlist/3 ...................................... 114
serve_queue/3 ................................... 123
set_input/1 ...................................... 56
set_output/1 ..................................... 56
set_prolog_flag/2 ................................ 102
set_stream_position/2 ........................... 57
set_value/2 ...................................... 76
setarg/3n ........................................ 83
setenv/2 ......................................... 149
setof/3 .......................................... 79
setrand/1 ........................................ 124
setup_call_catcher_cleanup/4 ................. 139
setup_call_cleanup/3 ........................... 139
sh/0 ................................................ 81
shared_object_search_path (yap_flag/2
... option) ......................................... 98
shell/0 ........................................... 135
shell/1 ........................................... 135, 136
show_op_counters/1 .............................. 197
show_ops_by_group/1 ............................. 197
show_table/1 ..................................... 194
showprofes/0 .................................... 88
showprofes/1 .................................... 88
simple/1 ......................................... 41
single_var_warnings (yap_flag/2 option) ..... 98
singleton/1 (read_term/2 option) ................ 60
skip/1 ............................................ 66
skip/2 ............................................ 67
sleep/1 .......................................... 136
socket/2 ......................................... 69
socket/4 ......................................... 69
socket_accept/2 .................................. 70
socket_accept/3 .................................. 70
socket_bind/2 .................................... 69
socket_buffering/4 ................................ 70
socket_close/1 ................................... 69
socket_connect/3 .................................. 70
socket_listen/2 ................................... 70
socket_select/5 ................................... 70
sort/2 ............................................ 47
source/0 ........................................... 23
source_file/1 ..................................... 104
source_file/2 ..................................... 104
source_location/2 ................................ 104
source_mode/2 .................................... 23
splay_access/5 .................................... 130
splay_delete/4 .................................... 130
splay_init/3 ...................................... 130
splay_insert/4 .................................... 130
splay_join/3 ...................................... 131
splay_split/5 ...................................... 131
split/2 ............................................ 112
split/3 ............................................ 112
spy/1 ............................................. 199
spy_write/2 ...................................... 200
srandom/1 ....................................... 53
stack_dump_on_error (yap_flag/2 option) ... 99
stack_shifts (statistics/2 option) .......... 94
start_low_level_trace/0 ....................... 195
start_matlab/1 .................................. 118
static_array/3 ................................... 90, 91
static_array_location/4 ....................... 90
static_array_properties/3 .................... 90
static_array_to_term/3 ....................... 91
static_code (statistics/2 option) .......... 94
statistics/0 ..................................... 92
statistics/2 ..................................... 93
stream_position/2 .............................. 58
stream_position_data/3 ....................... 58
stream_property/2 .............................. 57
stream_select/3 .................................. 56
strict_iso (yap_flag/2 option) ............... 98
style_check/1 ................................... 24
sub_atom/5 ....................................... 44
sublist/2 ........................................ 111
subsumes/2 ...................................... 137
subsumes chk/2 .................................. 137
succ/3 ............................................ 53
suffix/2 ......................................... 111
sum_list/2 ....................................... 111
sum_list/3 ....................................... 111
sumargs/4 ....................................... 114
sumlist/2 ....................................... 112
sumlist/4 ....................................... 114
sumnodes/4 ..................................... 114
syntax_errors (yap_flag/2 option) .......... 99
syntax_errors/1 (read_term/2 option) ...... 60
system/0 ......................................... 136
system/1 ......................................... 81
system/2 ......................................... 136
system_options (yap_flag/2 option) ....... 99
system_predicate/2 ............................. 73

T

tab/1 ............................................. 66
tab/2 ............................................. 67
table/1 .......................................... 193
table_statistics/1 ............................... 194
tabling_mode (yap_flag/2 option) .......... 194
tabling_mode/2 .................................. 193
tabling_statistics/0 ............................ 194
tell/1 ............................................. 58
telling/1 ......................................... 59
variant/2 .................................................. 137
verbose (yap_flag/2 option) .................. 101
verbose_load (yap_flag/2 option) ...... 102
verify_attributes/3 ................. 159
version (yap_flag/2 option) ........... 102
version/0 ........................................... 103
version/1 ........................................... 103
version_data (yap_flag/2 option) ...... 102
vertices/2 ........................................ 141
vertices_edges_to_ugraph/3 .......... 141

\textbf{W}

wait/2 .................................................. 136
walltime (statistics/2 option) ..... 94
when/2 ................................................. 154
windows (yap_flag/2 option) ........ 102
with_mutex/2 .................................... 188
with_output_to/2 ............................... 64
with_output_to_chars/2 ............... 132
with_output_to_chars/3 ............... 132
with_output_to_chars/4 ............... 132
working_directory/2 ................... 135, 150
write/1 ................................................. 60
write/2 ................................................. 66
write_canonical/1 ...................... 66
write_canonical/2 ...................... 66
write_depth/2 ..................................... 68
write_depth/3 ..................................... 68
write_strings (yap_flag/2 option) ... 102
write_term/2 ......................................... 60
write_term/3 ......................................... 66
write_to_chars/2 ......................... 131
write_to_chars/3 ......................... 131
writeln/1 ........................................ 60
writeq/1 ........................................ 61
writeq/2 ........................................ 66

\textbf{Y}

YAP_AddressFromSlot (C-Interface function) .... 215
YAP_AGCHook (C-Interface function) .......... 210
YAP_AllocSpaceFromYAP (C-Interface function) ........ 212
YAP_ArgOfTerm (C-Interface function) ........ 211
YAP_ArgsOfTerm (C-Interface function) ........ 211
YAP_ArgsToSlots (C-Interface function) .... 215
YAP_ArityOfFunctor (C-Interface function) .......... 211
YAP_AtomGetHold (C-Interface function) .... 210
YAP_AtomGetName (C-Interface function) ... 210
YAP_AtomGetLength (C-Interface function) .... 210
YAP_AtomGetTerm (C-Interface function) .... 210
YAP_AtomReleaseHold (C-Interface function) .......... 210
YAP_AtomToPred (C-Interface function) .... 215
YAP_BigNumOfTerm (C-Interface function) .... 209
YAP_BufferToAtomList (C-Interface function) ........ 212
YAP_BufferToDiffList (C-Interface function) ........ 212
YAP_BufferToString (C-Interface function) .... 212
YAP_CallProlog (C-Interface function) .... 216
YAP_ClearExceptions (C-Interface function) .......... 215
YAP_CloseAllOpenStreams (C-Interface function) ........ 213
YAP_CompileClause/1 ...................... 222
YAP_ContinueGoal/0 ....................... 222
YAP_CopyTerm/1 .................................. 223
YAP_CurrentSlot (C-Interface function) ........ 214
YAP_cuts fail (C-Interface function) ........ 217
YAP_cutsucceed (C-Interface function) ........ 217
YAP_EndConsult/0 ......................... 223
YAP_EnterGoal (C-Interface function) ........ 215
YAP_Exit/1 ........................................ 222
YAP_FloatOfTerm (C-Interface function) ........ 209
YAP_FlushAllStreams (C-Interface function) ........ 213
YAP_FloatOfTerm (C-Interface function) ........ 209
YAP_FunctorOfTerm (C-Interface function) ........ 210
YAP_ContinueGoal/0 ................. 216
YAP_FunctorOfTerm (C-Interface function) ........ 211
YAP_FunctorToPred (C-Interface function) ........ 215
YAP_GetValue/1 .................................. 222
YAP_IdentifyTerm (C-Interface function) ........ 211
YAP_ContinueGoal/0 ...................... 222
YAP_ContinueGoal/2 ...................... 223
YAP_InitSlot (C-Interface function) ........ 214
YAP_InitOfTerm (C-Interface function) ........ 210
YAP_IsApplTerm (C-Interface function) ........ 209
YAP_IsAtomTerm (C-Interface function) ........ 209
YAP_IsBigNumTerm (C-Interface function) ........ 209
YAP_IsDBRefTerm (C-Interface function) ........ 209
YAP_IsFloatTerm (C-Interface function) ........ 209
YAP_IsIntTerm (C-Interface function) ........ 209
YAP_IsNonVarTerm (C-Interface function) ........ 209
YAP_IsVarTerm (C-Interface function) ........ 209
YAP_IsWideAtom (C-Interface function) ........ 210
YAP_LeaveGoal (C-Interface function) ........ 210
YAP_LookupAtom (C-Interface function) ........ 210
YAP_LookupWideAtom (C-Interface function) ........ 210
<table>
<thead>
<tr>
<th>Predicate Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAP_MkApplTerm (C-Interface function)</td>
<td>211</td>
</tr>
<tr>
<td>YAP_MkAtomTerm (C-Interface function)</td>
<td>210</td>
</tr>
<tr>
<td>YAP_MkBigNumTerm (C-Interface function)</td>
<td>209</td>
</tr>
<tr>
<td>YAP_MkFloatTerm (C-Interface function)</td>
<td>209</td>
</tr>
<tr>
<td>YAP_MkFunctor (C-Interface function)</td>
<td>211</td>
</tr>
<tr>
<td>YAP_MkIntTerm (C-Interface function)</td>
<td>209</td>
</tr>
<tr>
<td>YAP_MkNewApplTerm (C-Interface function)</td>
<td>211</td>
</tr>
<tr>
<td>YAP_MkNewPairTerm (C-Interface function)</td>
<td>211</td>
</tr>
<tr>
<td>YAP_NameOfFunctor (C-Interface function)</td>
<td>211</td>
</tr>
<tr>
<td>YAP_NBufferToAtomList (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_NBufferToDiffList (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_NBufferToString (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_NewSlots (C-Interface function)</td>
<td>214</td>
</tr>
<tr>
<td>YAP_NWideBufferToAtomList (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_NWideBufferToDiffList (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_NWideBufferToString (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_NameOfFunctor (C-Interface function)</td>
<td>211</td>
</tr>
<tr>
<td>YAP_OpenStream (C-Interface function)</td>
<td>213</td>
</tr>
<tr>
<td>YAP_PRESERVE_DATA (C-Interface function)</td>
<td>217</td>
</tr>
<tr>
<td>YAP_PRESERVED_DATA (C-Interface function)</td>
<td>217</td>
</tr>
<tr>
<td>YAP_PutInSlot (C-Interface function)</td>
<td>217</td>
</tr>
<tr>
<td>YAP_PutValue/2 (C-Interface function)</td>
<td>215</td>
</tr>
<tr>
<td>YAP_Read/1 (C-Interface function)</td>
<td>223</td>
</tr>
<tr>
<td>YAP_ReadBuffer (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_Record (C-Interface function)</td>
<td>213</td>
</tr>
<tr>
<td>YAP_Recorded (C-Interface function)</td>
<td>213</td>
</tr>
<tr>
<td>YAP_RecoverSlots (C-Interface function)</td>
<td>215</td>
</tr>
<tr>
<td>YAP_Reset (C-Interface function)</td>
<td>215</td>
</tr>
<tr>
<td>YAP_RestartGoal (C-Interface function)</td>
<td>215</td>
</tr>
<tr>
<td>YAP_RetryGoal (C-Interface function)</td>
<td>216</td>
</tr>
<tr>
<td>YAP_RunGoal (C-Interface function)</td>
<td>214</td>
</tr>
<tr>
<td>YAP_ShutdownGoal (C-Interface function)</td>
<td>215</td>
</tr>
<tr>
<td>YAP_SlotsToArgs (C-Interface function)</td>
<td>215</td>
</tr>
<tr>
<td>YAP_StreamToFileNo (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_StringToBuffer (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_TailOfTerm (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_Unify (C-Interface function)</td>
<td>211</td>
</tr>
<tr>
<td>YAP_UserBackCPredicate (C-Interface function)</td>
<td>217</td>
</tr>
<tr>
<td>YAP_UserBackCutCPredicate (C-Interface function)</td>
<td>217</td>
</tr>
<tr>
<td>YAP_UserCPredicate (C-Interface function)</td>
<td>217</td>
</tr>
<tr>
<td>YAP_WideAtomName (C-Interface function)</td>
<td>210</td>
</tr>
<tr>
<td>YAP_WideBufferToAtomList (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_WideBufferToDiffList (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_WideBufferToString (C-Interface function)</td>
<td>212</td>
</tr>
<tr>
<td>YAP_WRITE_DATA</td>
<td>212</td>
</tr>
<tr>
<td>YAP_Write/3</td>
<td>223</td>
</tr>
<tr>
<td>YAP_WriteBuffer/4</td>
<td>223</td>
</tr>
<tr>
<td>YAPBINDIR</td>
<td>10</td>
</tr>
<tr>
<td>YAPLIBDIR</td>
<td>10</td>
</tr>
<tr>
<td>YAPSHAREDIR</td>
<td>10</td>
</tr>
</tbody>
</table>
## Concept Index

**A**
- aggregate ........................................ 105
- anonymous variable ............................... 17
- apply ........................................... 107
- association list ................................ 107
- atom ........................................... 16
- attribute declaration ......................... 158
- attributed variables ......................... 155
- AVL trees .................................... 108

**B**
- BOM ........................................... 19
- booting ........................................ 9
- Byte Order Mark ............................... 19

**C**
- cleanup ....................................... 138
- CLPQ ......................................... 163
- CLPR ......................................... 163
- comment ...................................... 17
- Counting Calls ................................ 88

**D**
- declaration, attribute ......................... 158

**E**
- Efficient Directed Graphs ..................... 144
- encodings .................................... 17
- end of term ................................... 14
- environment variables ....................... 10

**F**
- floating-point number ......................... 15

**G**
- global variables .............................. 84

**H**
- heap ........................................... 109

**I**
- installation ................................... 3
- integer ....................................... 14
- internationalization ......................... 17

**L**
- lam ........................................... 147
- Line Utilities Library ......................... 112
- list manipulation ............................. 110
- Logtalk .................................... 179

**M**
- machine optimizations ....................... 4
- macros ....................................... 113
- Matlab Interface ............................. 118
- Matrix Library ................................ 115
- mutable variables ............................ 83

**N**
- number ....................................... 14

**O**
- Operating System Utilities .................. 132
- or-parallelism ................................. 191
- ordered set .................................. 121

**P**
- parallelism .................................. 191
- profiling ...................................... 86
- pseudo random ............................... 122
- punctuation token ............................ 17

**Q**
- queue ........................................ 123, 124

**R**
- Red-Black Trees ............................... 125
- regular expressions ......................... 127

**S**
- splay trees ................................. 130
- string ....................................... 15
- string I/O .................................. 131
- SWI-Compatible foreign file loading ........ 129
- SWI-Prolog .................................. 148
- syntax ...................................... 3, 13

**T**
- tabling ...................................... 193
- timeout ...................................... 140
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>token</td>
<td>14</td>
</tr>
<tr>
<td>tries</td>
<td>137</td>
</tr>
<tr>
<td><strong>U</strong></td>
<td></td>
</tr>
<tr>
<td>UCS</td>
<td>17</td>
</tr>
<tr>
<td>undirected graphs</td>
<td>146</td>
</tr>
<tr>
<td>Unicode</td>
<td>17</td>
</tr>
<tr>
<td>unweighted graphs</td>
<td>141</td>
</tr>
<tr>
<td>updatable tree</td>
<td>140</td>
</tr>
<tr>
<td>update semantics</td>
<td>236</td>
</tr>
<tr>
<td>updating terms</td>
<td>83</td>
</tr>
<tr>
<td>UTF-8</td>
<td>17</td>
</tr>
<tr>
<td>utilities on terms</td>
<td>136</td>
</tr>
<tr>
<td><strong>V</strong></td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>16</td>
</tr>
</tbody>
</table>