YAP User's Manual

Version 4.5.5

Vítor Santos Costa,
Luís Damas,
Rogério Reis, and
Rúben Azevedo

Table of Contents

Introduction

This document provides User information on version 4.5.5 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.
- 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.

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 yappers@ncc.up.pt. To subscribe to the mailing list, send a request to majordomo@ncc.up.pt with body "subscribe yappers".

Online 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 Srinivasian.

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 at TUM by Ludwig-Maximilians-Universitaet Muenchen (LMU) by Dr. Fruehwirth Thom and by Dr. Christian Holzbaur. The package is distributed under license from LMU (Ludwig-Maximilians-University), Munich, Germany:

  Permission is granted to copy and distribute modified versions of this chapter 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 chapter into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by LMU.

  Copyright © 1996-98 LMU (Ludwig-Maximilians-University)
  Munich, Germany

- The CLP(Q,R) package developed at OFAI Austrian Research Institute for Artificial Intelligence by Christian Holzbaur. The package is distributed under the OFAI license. Documentation on this package is a chapter of this manual, which is covered by the OFAI license:

  Copyright © 1992,1993,1994,1995 OFAI Austrian Research Institute for Artificial Intelligence (OFAI) Schottengasse 3 A-1010 Vienna, Austria

  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, except that this permission notice may be stated in a translation approved by the OFAI.

- The Logtalk Object-Oriented system is developed at the University of Beira Interior, Portugal, by Paulo Moura. The package is distributed under the Perl Artistic License. Instructions about loading this package are included in this document. The documentation on this package is distributed separately from yap.tex.

  Copyright © 1998-2001 Paulo Moura

- 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 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 coroutining 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={local,batched}` allows one of the two forms of tabling. 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-gmp[=DIR]` give a path to where one can find the `GMP` library if not installed in the default path.

Next follow 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-regs -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, eg, 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.*:

```
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 `--enable-cygwin=no` 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 `/PROGRA~1/Yap` in the standard Makefile. This string will usually be expanded into `c:\Program Files\Yap` by Windows.

The cygwin environment does not provide `gmp`. 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-coroutining \\
                  --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.

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
   ```

   in `Project.Project Settings.Debug.Program Arguments`.
6. You need the sockets and yap libraries. Add

   ```
   ws2_32.lib yapdll.lib to
   ```

   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).
   ^Z
   ```

   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` *n* allocate *n* K bytes for local and global stacks

`-h` *n* allocate *n* K bytes for heap and auxiliary stacks

`-t` *n* allocate *n* K bytes for the trail stack

`-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.

`-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

`--` separator for arguments to Prolog code. These arguments are visible through the unix/1 built-in.

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`' 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:

```
#!/usr/local/bin/yap -L
#
# Hello World script file using Yap
#

:- 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:

```
#!/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 this 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` builtin. 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, eg:

```
+ (a,b)         [the same as '+'(',',(a,b)) of arity one]
```

  versus

```
+(a,b)          [the same as '+'(a,b) 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 `{...}` stands for optionality, `+` optional repetition (one or more times), `<`*digit*`>` denotes one of the characters 0 ... 9, `|` 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:

```
0'd  100
```

YAP (version 4.5.5) 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:

```
<float> := <digit>+{<dot><digit>+}
             <exponent-marker>{<sign>}<digit>+
          |<digit>+<dot><digit>+
             {<exponent-marker>{<sign>}<digit>+}
```

where <*dot*> denotes the decimal-point character '.', <*exponent-marker*> denotes one of 'e' or 'E', and <*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:

```
string --> '"' string_quoted_characters '"'

string_quoted_characters --> '"' '"' string_quoted_characters
string_quoted_characters --> '\'
                         escape_sequence string_quoted_characters
string_quoted_characters -->
                         string_character string_quoted_characters

escape_sequence --> 'a' | 'b' | 'r' | 'f' | 't' | 'n' | 'v'
escape_sequence --> '\' | '"' | ''' | '`'
escape_sequence --> at_most_3_octal_digit_seq_char '\'
escape_sequence --> 'x' at_most_2_hexa_digit_seq_char '\'
```

where `string_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 Yap4.3.0 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), and `v` (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:

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


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:

```
<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.

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`.

`ensure_loaded(+F) [ISO]`
: When the files specified by *F* are module files, `ensure_loaded/1` loads them if they have note 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.

`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.

```
?- 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 discontigous procedures may be separated by clauses from other procedures.

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

`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]`
: 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 Saving and Loading Prolog States

`save(+F)` Saves an image of the current state of YAP in file *F*. From **Yap4.1.3** onwards, YAP saved states are executable files in the Unix ports.

`save(+F,-OUT)`
: Saves an image of the current state of YAP in file *F*. From **Yap4.1.3** onwards, YAP saved states are executable files in the Unix ports.

Unify *OUT* with 1 when saving the file and *OUT* with 0 when restoring the saved state.

`save_program(+F)`
: Saves an image of the current state of the YAP database in file *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 an 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 both a way to access predicates private to a module and that 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: only meta-calls that cross module boundaries are slowed down by the presence of modules.

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 mode for a clause by prefixing a clause with its module, say:

```
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:

```
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 into, say:

```
nasa:launch(apollo,13).
```

will execute the goal `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 predicate 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 predicates `consult/1`, `reconsult/1`, `ensure_loaded/1` or `use_module/2`. The non-public predicates of a module file are not visible by other files; they can, however, be accessed if the module name is prefixed to the file name through the `:/2` operator.

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

`module(+M,+L, +Options)`
: Similar to `module/2`, this predicate 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 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 binded 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:

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

...

a(G) :- call(G)

...
```

The expected behavior for this procedure is to execute goal G within the current module, that is, within `example`. On the other hand, when executing `call/1` the system only knows where `call/1` was defined, that is, it only knows of `primitives`. A similar problem arises for `assert/1` and friends.

The `meta_call/1` declaration informs the system that some arguments of a procedure are goals, clauses or clauses heads, and that these arguments must be expanded to receive the current source module:

`meta_predicate` *G1,....,Gn*
: Each *Gi* is a mode specification. For example, a declaration for `call/1` and `setof/3` would be of the form:

```
:- meta_predicate call(:), setof(?,:,?).
```

If the argument is `:` or an integer, the argument is a call and must be expanded. Otherwise, the argument should not be expanded. Note that the system already includes declarations for all built-ins.

In the previous example, the only argument to `call/1` must be expanded, resulting in the following code:

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

...

a(G) :- call(example:G)
```

...

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) :- 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) :- q(X); r(X).
```

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

`true [ISO]`

Succeeds once.

`fail [ISO]`

Fails always.

`false` The same as fail

`! [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:

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

With the above definition

```
?- member(X,[1,2,3]).
```

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

```
member(X,[X|_]) :- !.
member(X,[_|L]) :- member(X,L).
```

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:

```
\+(P) :- P, !, fail.
\+(_).
```

if *P* did not include "cuts".

`not +P` Goal *P* is not provable. The same as `'\+ P'`.

This predicate is kept for compatibility with C-Prolog and previous versions of YAP. Uses of `not/1` should be replace by `(\+)/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:

`+P -> +Q` "if P then Q".

`+P -> +Q; +R`
"if P then Q else R".

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 `(->)/2` does not affect the scope of cuts in its arguments.

`repeat [ISO]`

Succeeds repeatedly.

In the next example, `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_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) [ISO]`

Call goal *H* once per each solution of goal *H*. If goal *H* has no solutions, call goal *I*.

The builtin `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,
Y = b ? ;

X = 2,
Y = a ? ;

X = 2,
Y = b ? ;

no
```

The system will backtrack over the two solutions for `a/1` and the two solutions for `b/1`, generating four solutions.

Cuts are allowed inside the first goal *G*, but they will only prune over *G*.

If you want *G* to be deterministic you should use if-then-else, as it is both more efficient and more portable.

`once(:G) [ISO]`
Execute the goal *G* only once. The predicate is defined by:

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

Note that cuts inside `once/1` can only cut the other goals inside `once/1`.

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

`break` Suspends the execution of the current goal and creates a new execution level similar to the top level, displaying the following message:

```
[ Break (level <number>) ]
```

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.

`halt [ISO]`
Halts Prolog, and exits to the calling application. In YAP, `halt/0` returns the exit code `0`.

`halt(+ 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 through 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:

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

  and executing the goal:

  ```
  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.

`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.

`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`.

6.3 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.

`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 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.

`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.

```
name(yap,L).
```

will return:

```
L = [121,97,112].
```

and

```
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 atoms, 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.

`atom_concat(+A1,+A2,?A)`

The predicate holds when the first argument and second argument are atoms, and the third unifies with the atom obtained by concatenating the first two arguments.

`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.

`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.

`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.

`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 substring 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 substrings of A.

`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, of if N is not a positive integer. Note that previous versions of YAP would fail silently under these errors.

`functor(T,F,N)`
: 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:n

```
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:

```
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)`.

`copy_term(?TI,-TF) [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*.

6.4 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, ie: 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;
- Integers are sorted in increasing order;
- atoms are sorted in lexicographic order;
- compound terms are ordered first by name, then by arity of the main functor, and finally by their arguments in left-to-right order.

`compare(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;

`X == Y [ISO]`
Succeeds if terms X and Y are strictly identical. The difference between this predicate and `=/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 \== Y [ISO]`
Terms X and Y are not strictly identical.

`X @< Y [ISO]`
Term X precedes term Y in the standard order.

`X @=< Y [ISO]`
Term X does not follow term Y in the standard order.

`X @> Y [ISO]`
Term X follows term Y in the standard order.

`X @>= Y [ISO]`
Term X does not precede term Y in the standard order.

`sort(+L,-S)`
Unifies S with the list obtained by sorting L and merging identical (in the sense of `==`) elements.

`keysort(+L,S)`
Assuming L is a list of the form `Key-Value`, `keysort(+L,S)` unifies S with the list obtained from L, by sorting its elements according to the value of *Key*.

```
?- 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]
```

`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.

6.5 Arithmetic

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 rem Y` Integer remainder, the same as `mod`.

`exp(X) [ISO]`
Natural exponential.

`log(X) [ISO]`
Natural logarithm.

`log10(X)` Decimal logarithm.

`sqrt(X) [ISO]`
Square root.

`sin(X) [ISO]`
Sine.

`cos(X) [ISO]`
Cosine.

`tan(X)` Tangent.

`asin(X)` Arc sine.

`acos(X)` Arc cosine.

`atan(X) [ISO]`
Arc tangent.

`atan2(X)` Four-quadrant arc tangent.

`sinh(X)` Hyperbolic sine.

`cosh(X)` Hyperbolic cosine.

`tanh(X)` Hyperbolic tangent.

`asinh(X)` Hyperbolic arc sine.

`acosh(X)` Hyperbolic arc cosine.

`atanh(X)` Hyperbolic arc tangent.

`integer(X) [ISO]`
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 float 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 float 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)`
The float that is the integral value between X and 0 closest to X.

`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 [ISO]`
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 integer X.

`[X]` Evaluates to X for expression X. Useful because character strings in Prolog are lists of character codes.

```
X is Y*10+C-"0"
```

is the same as

```
X is Y*10+C-[48].
```

which would be evaluated as:

```
X is Y*10+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 circumrefence to its diameter.

`e` The base of the natural logarithms.

`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 iff the result of evaluating the expression Y unifies with X. This is the predicate normally used to perform evaluation of arithmetic expressions:

```
X is 2+3*4
```

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 =\= +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:

- In contrast to previous versions of Yap, Yap4 *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.

6.6 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.

6.6.1 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.

`open(+F,+M,-S,+Opts) [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(+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(+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(+Name)`
: Specify an alias to the stream. The alias `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`

`close(+S) [ISO]`

Closes the stream S. If S does not stand for a stream currently opened an error is reported. The streams `user_input`, `user_output`, and `user_error` can never be closed.

By default, give a file name, `close/1` will also try to close a corresponding open stream. This feature is not available in ISO or SICStus languages mode and is deprecated.

`close(+S,+O) [ISO]`
: Closes the stream *S*, following options *O*.

  The only valid options are `force(true)` and `force(false)`. YAP currently ignores these options.

`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`.

`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*.

`flush_output [ISO]`
: Send all data in the output buffer to current output stream.

`flush_output(+S) [ISO]`
: Send all data in the output buffer to 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* openeded 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.

6.6.2 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.).

6.6.3 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 operators (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:

`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 conversion 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.

`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 necessary, and operators are ignored, that is, the term is written in standard parenthesized prefix notation.

`write_term(+T, +Opts) [ISO]`
: Displays term *T* on the current output stream, according to the following options:

  `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`.

  `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.

`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).

`'~Ne'`
`'~NE'`
`'~Nf'`
`'~Ng'`
`'~NG'` 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:

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

As an example:

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

`'~Nd'` 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 $N = 0$.

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

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

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

`'~i'` Ignore the next argument in the list of arguments:

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

'~k' Print the next argument with `write_canonical`:

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

'~Nn' Print N newlines (where N defaults to 1).

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

'~Nr' The next argument must be an integer, and N is interpreted as a radix, such that `2 <= N <= 36` (the default is 8).

```
?- format("~2r, 0x~16r, ~r",
          [150000, 150000, 150000]).
100100100111110000, 0x249f0, 444760
```

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

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

```
?- format("~2r, 0x~16r, ~r",
          [150000, 150000, 150000]).
100100100111110000, 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:

```
   ?- 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(+S,+T,+L)`
Print formatted output to stream *S*.

6.6.4 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.

6.6.5 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_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.

6.6.6 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`.

6.6.7 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)`
: Unifies *T* and L, respectively, with the values of the maximum depth of a term and the maximum length of a list, that will be used by `write/1` or `write/2`. The default value for both arguments is 0, meaning unlimited depth and length.

```
?- write_depth(3,5).
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.

6.6.8 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 builtins 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.

`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.

`hostname_address(?HOSTNAME,?IP_ADDRESS)`
: *HOSTNAME* is an host name and *IP_ADDRESS* its IP address in number and dots notation.

6.7 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.

`dynamic +P`
: Declares predicate *P* or list of predicates [*P1*,...,*Pn*] as a dynamic predicate. *P* must be written in form: *name/arity*.

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

  a more convenient form can be used:

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

  or, equivalently,

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

Note:

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

`dynamic_predicate(+P,+Semantics)`
: Declares predicate *P* or list of predicates [*P1*,...,*Pn*] as a dynamic predicate following either `logical` or `immediate` semantics.

6.7.1 Modification of the Data Base

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

`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 builtin 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 19).

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

6.7.2 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.

6.7.3 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 6.7.1 [Modifying the Database], page 55) 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.

6.8 Internal Data Base

Some programs need global information for, eg., 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).

`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 preceeding 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:
  - K may be given, in this case the built-in will return all elements of the internal data-base that match the key.
  - R may be given, if so returning the key and element that match the reference.

`nth_instance(?K,?Index,T,?R)`
: Fetches the *Index*nth entry in the internal database under the key K. Entries are numbered from one. If the key K are the *Index* are bound, a reference is unified with R. Otherwise, the reference R must be given, and the term the system will find the matching key and index.

`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`. 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.

`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.

```
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.

6.9 The Blackboard

YAP implements a blackboard in the style of the SICStus Prolog blackboard. The blackboard uses the same underlying mechanism as the internal data-base but has several important 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.

`bb_put(+`*Key*`,?`*Term*`)`
: Store term table *Term* in the blackboard under key *Key*. If a previous term was stored under key *Key* it is simply forgotten.

`bb_get(+`*Key*`,?`*Term*`)`
: Unify *Term* with a term stored in the blackboard under key *Key*, or fail silently if no such term exists.

`bb_delete(+`*Key*`,?`*Term*`)`
: Delete any term stored in the blackboard under key *Key* and unify it with *Term*. Fail silently if no such term exists.

`bb_update(+`*Key*`,?`*Term*`,?`*New*`)`
: Atomically unify a term stored in the blackboard under key *Key* with *Term*, and if the unification succeeds replace it by *New*. Fail silently if no such term exists or if unification fails.

6.10 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. `findall/3` gives you the fastest, but crudest solution. The other built-in predicates postprocess the result of the query in several different ways:

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

Unifies L with a list that contains all the instantiations of the term T satisfying the goal G.

With the following program:

```
a(2,1).
a(1,1).
a(2,2).
```

the answer to the query

```
findall(X,a(X,Y),L).
```

would be:

```
X = _32
Y = _33
L = [2,1,2];
no
```

`findall(T,+G,+L,-L0)`

Similar to `findall/3`, but appends all answers to list *L0*.

`all(T,+G,-L)`

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

```
all(X,a(X,Y),L).
```

would be:

```
X = _32
Y = _33
L = [2,1];
no
```

`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

```
bagof(X,a(X,Y),L).

would be:
X = _32
Y = 1
L = [2,1];
X = _32
Y = 2
L = [2];
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

```
setof(X,a(X,Y),L).
```

would be:

```
X = _32
Y = 1
L = [1,2];
X = _32
Y = 2
L = [2];
no
```

6.11 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:

```
head --> 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:

`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 to use the user defined predicate `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.

`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)`.

6.12 Access to Operating System Functionality

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

`cd(+D)` Changes the current directory (on UNIX environments).

`environ(+E,-S)`
Given an enviromment variable *E* this predicate unifies the second argument *S* with its value.

`getcwd(-D)`
Unify the current directory, represented as an atom, with the argument *D*.

`putenv(+E,+S)`
Set environment variable *E* to the value *S*. If the environment variable *E* does not exist, create a new one. Both the environment variable and the value must be atoms.

`rename(+F,+G)`
Renames file *F* to *G*.

`sh` Creates a new shell interaction.

`system(+S)`
Passes command *S* to the Bourne shell (on UNIX environments) or the current command interpreter in WIN32 environments.

`unix(+S)` Access to Unix-like functionality:

- `argv/1` Return a list of arguments to the program. These are the arguments that follow a `--`, as in the usual Unix convention.
- `cd/0` Change to home directory.
- `cd/1` Change to given directory. Acceptable directory names are strings or atoms.

`environ/2`
: If the first argument is an atom, unify the second argument with the value of the corresponding environment variable.

`getcwd/1` Unify the first argument with an atom representing the current directory.

`putenv/2` Set environment variable *E* to the value *S*. If the environment variable *E* does not exist, create a new one. Both the environment variable and the value must be atoms.

`shell/1` Execute command under current shell. Acceptable commands are strings or atoms.

`system/1` Execute command with `/bin/sh`. Acceptable commands are strings or atoms.

`shell/0` Execute a new shell.

`alarm(+`*`Seconds`*`,+`*`Callable`*`,+`*`OldAlarm`*`)`
: Arranges for YAP to be interrupted in *Seconds* seconds. 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:

```
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:

```
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:

```
once_with_alarm(Time,Goal,DoOnAlarm) :-
   catch(execute_once_with_alarm(Time, Goal), alarm, DoOnAlarm).

execute_once_with_alarm(Time, Goal) :-
        alarm(Time, alarm, _),
        ( call(Goal) -> alarm(0, alarm, _) ; alarm(0, alarm, _)).
```

The procedure has three arguments: the *Time* 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_signal(+`*Signal*`,?`*OldAction*`,+`*Callable*`)`

Set the interrupt handler for soft interrupt *Signal* to be *Callable*. *OldAction* is unified with the previous handler.

Only a subset of the software interrupts (signals) can have their handlers manipulated through `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 `on_signal/3`. It is chosen so that it matches the signal's POSIX name.

`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_up (Hangup)`

SIGHUP in Unix/Linux; Reconsult the initialization files ~/.yaprc, ~/.prologrc and ~/prolog.ini.

`sig_usr1 and sig_usr2 (User signals)`

SIGUSR1 and SIGUSR2 in Unix/Linux; Print a message and halt.

A special case is made, where if *Callable* is bound to `default`, then the default handler is restored for that signal.

A call in the form `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.

6.13 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 `setarg/3` primitive allows updating any argument of a Prolog compound terms. The `mutable` family of predicates provides *mutable variables*. They should be used instead of `setarg/3`, as they allow the encapsulation of accesses to updatable variables. Their implementation can also be more efficient for long deterministic computations.

`setarg(+I,+S,?T)`
: Set the value of the Ith argument of term S to term T.

`create_mutable(+D,-M)`
: Create new mutable variable M with initial value 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.

6.14 Profiling Prolog Programs

Predicates compiled with YAP's flag `profiling` set to `on`, keep information on the number of times the predicate was called. This information can be used to detect what are the most commonly called predicates in the program.

The YAP profiling sub-system is currently under-development. Functionality for this sub-system will increase with newer implementation.

Notes:

- 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.

```
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
        % 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:
        % 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.

6.15 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 the *Retries* counter, and last the *CallsAndRetries* counter.

`call_count_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:

```
?- yap_flag(call_counting,on), [-user]. l :- l. end_of_file. yap_flag(call_counting

yes

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.

6.16 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 builtin `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.
- `dbref`: 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] is Y[J]+Z[I].
```

should give the same results as:

```
G(X,Y,Z,I,J) :-
        array_element(X,I,E1),
        array_element(Y,J,E2),
        array_element(Z,I,E3),
        E1 is E2+E3.
```

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.

`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 the 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`, `dbref`, `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 dynamic 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 elements 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.

6.17 Predicate Information

Built-ins that return information on the current predicates and modules:

`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 typein module.

`current_module(M,F)`
: Succeeds if M are current modules associated to the file F.

6.18 Miscellaneous

`statistics/0`
: Send to the current user error stream general information on space used and time spent by the system.

```
?- statistics.
memory (total)        4784124 bytes
   program space      3055616 bytes:    1392224 in use,      1663392 free
                                                             2228132  max
   stack space        1531904 bytes:        464 in use,      1531440 free
     global stack:                          96 in use,       616684  max
      local stack:                         368 in use,       546208  max
   trail stack         196604 bytes:          8 in use,       196596 free

    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(?Param,-Info)`

Gives statistical information on the system parameter given by first argument:

`cputime` `[Time since Boot,Time From Last Call to Cputime]`
This gives the total cputime in milliseconds spent executing Prolog code, garbage collection and stack shifts time included.

`garbage_collection`
`[Number of GCs,Total Global Recovered,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 `yap_flag(gc_trace,verbose)`.

`global_stack`
`[Global Stack Used,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`
`[Local Stack Used,Execution Stack Free]`
Space in kbytes currently used in the local stack, and space available for expansion by the local and global stacks.

`heap` `[Heap Used,Heap Free]`
Total space in kbytes not recoverable in backtracking. It includes the program code, internal data base, and, atom symbol table.

`program` `[Program Space Used,Program Space Free]`
Equivalent to `heap`.

`runtime` `[Time since Boot,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)`.

`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 Runtime*]
This gives the clock time in milliseconds since starting Prolog.

`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 `--`.

`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 enables, `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.

`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`.

`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.

`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.

`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` multiprecision library. If `bounded` is false, requests for `max_integer` will fail.

`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` multiprecision library. If `bounded` is false, requests for `min_integer` will fail.

`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.

`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.

`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.

`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 not 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 `coroutining`, `depth_limit`, the `low_level_tracer`, `or-parallelism`, `rational_trees`, `tabling`, `threads`, or the `wam_profiler`.

`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.

`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.

`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_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.

`version`

Read-only flag that giving the current version of Yap.

`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`.

`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.

`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 filed 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.

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 Apply Macros

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.

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*.

`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 collect a result in *Accumulator* (depth-first and left-to-right order)

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')]).
```

7.2 Association Lists

The following association list manipulation predicates are available once included with the `use_module(library(assoc))` command.

`assoc_to_list(+`*Assoc*`,?`*List*`)`
: Given an association list *Assoc* unify *List* with a list of the form *Key*-*Val*, where the elements *Key* are in ascending order.

`empty_assoc(+`*Assoc*`)`
: Succeeds if association list *Assoc* is empty.

`gen_assoc(+`*Assoc*`,?`*Key*`,?`*Value*`)`
: Given the association list *Assoc*, unify *Key* and *Value* with two associated elements. It can be used to enumerate all elements in the association list.

`get_assoc(+`*Key*`,+`*Assoc*`,?`*Value*`)`
: If *Key* is one of the elements in the association list *Assoc*, return the associated value.

`get_assoc(+Key,+Assoc,?Value,+NAssoc,?NValue)`
: If *Key* is one of the elements in the association list *Assoc*, return the associated value *Value* and a new association list *NAssoc* where *Key* is associated with *NValue*.

`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,?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.

`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*.

7.3 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_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*.

7.4 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(NlgN). 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.

7.5 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.

`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*.

```
?- flatten([[1],[2,3],[4,[5,6],7,8]],L).

L = [1,2,3,4,5,6,7,8] ? ;

no
```

`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.

`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 Nth 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`

`nth(?N, ?List, ?Elem)`
: The same as `nth0/3`, except that it counts from 1, that is `nth(1, [H|_], H)`.

`nth0(?N, ?List, ?Elem, ?Rest)`
: Unifies *Elem* with the Nth element of *List*, counting from 0, 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. `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*.

`nth(?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. `nth(1, 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*.

`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(-,+)` and `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, ?Set, ?Residue)`
: True when *Set* is a list, *Element* occurs in *Set*, and *Residue* is everything in *Set* except *Element* (things stay in the same order).

`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 integers, and *Total* is their sum.

`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.

7.6 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_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.

7.7 Pseudo Random Number Integer Generator

The following routines produce random non-negative integers in the range 0 .. 2^(w-1) -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 random, 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.

7.8 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)`
Ads 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*.

7.9 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)`.

7.10 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.

`insert(+T0,+Key,?Value,+TF)`
: Add an element with key *Key* and *Value* to the tree *T0* creating a new AVL tree *TF*. Duplicated elements are not allowed.

`lookup(+Key,-Value,+T)`
: Lookup an element with key *Key* in the red-black tree *T*, returning the value *Value*.

`lookupall(+Key,-Value,+T)`
: Lookup all elements with key *Key* in the red-black tree *T*, returning the value *Value*.

`new(?T)` Create a new tree.

`delete(+T,+Key,-TN)`
: Delete element with key *Key* from the tree *T*, returning a new tree *TN*.

7.11 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 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 "`^`"). To include a literal "-", 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 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 *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

```
regexp("(a*)b*","aabaaabb", [], [X,Y])
```

Considering only the rules given so far, X and Y could end up with the values `"aabb"` and `"aa"`, `"aaab"` and `"aaa"`, `"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:

```
regexp("(ab|a)(b*)c",  "abc", [], [X,Y,Z])
```

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.

7.12 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*`,+`*RighTree*`,-`*NewTree*`)`
: Combine trees *LeftTree* and *RighTree* 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 *RighTree* and destroys both *LeftTree* and *RighTree*.

`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 operations destroys *Tree*.

7.13 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 builtin 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*`, -`*Result0*`, -`*Result*`)`
: 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*.

`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 builtins:

`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 builtins 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.

7.14 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.

`datime(datime(-`*Year*`, -`*Month*`, -`*DayOfTheMonth*`,`
-*Hour*, -*Minute*, -*Second*) The `datime/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) ?
```

`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 returned on local time. This function uses the WIN32 `GetLocalTime` function or the Unix `mktime` function.

```
?- mktime(datime(2001,5,28,15,29,46),X).

X = 991081786 ? ;
```

`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).
```

`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]).
```

`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,..,'.']
```

The predicates uses the `dirent` family of routines in Unix environments, and `findfirst` in WIN32.

`file_exists(+File)`
The atom *File* corresponds to an existing file.

`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(FileName)`, gives the file pointed to by a symbolic link. Properties can be obtained through backtracking:

```
?- file_property('Makefile',P).

P = type(regular) ? ;

P = size(2375) ? ;

P = mod_time(990826911) ? ;

no
```

`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` builtin 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.

```
?- 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* indepent 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.

`pid(-Id)`
: Unify *Id* with the process identifier for the current process. An interface to the `getpid` function.

`tmpnam(-File)`
: Interface with *tmpnam*: create an unique file and unify its name with *File*.

exec(+*Command*,[+*InputStream*,+*OutputStream*,+*ErrorStream*], -*Status*)
: Execute command *Command* with its streams connected to *InputStream*, *OutputStream*, and *ErrorStream*. The result for the command is returned in *Status*. 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`
: Start a new shell and leave Yap in background until the shell completes. 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 process for *Time* seconds. 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*.

7.15 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 structure 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 a list of all variables in term *Term*.

`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*.

7.16 Call With registered Cleanup Calls

`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.

`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.

7.17 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 `timeout`.

  This command is implemented by activating an alarm at procedure entry. If the timer expires before the goal completes, the alarm will through 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`. 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.

7.18 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.

7.19 Unweighted Graphs

The following graph manipulation routines are based from 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 builtins 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 vertice 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(+Vertices, +Graph, -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-[],6-[],
              7-[],8-[]],[1-6,2-3,3-2,5-7,3-2,4-5],NL).

NL = [1-[3,5,6],2-[3,4],3-[2],4-[5],5-[7],6-[],7-[],8-[]]
```

`sub_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 vertice 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]
```

`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]
```

8 Extensions

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`

9 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(a(...)))))`, or *rational tree*.

By default, rational trees are not supported in YAP, and these terms can easily lead to infinite computation. For example, `X = a(X), X = X` will 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), ground(X)` will succeed instead of looping. Other affected builtins 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 Coroutining

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:

`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 `predname(`*C1*`,...,`*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.

`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:

`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.

`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.

11 Attributed Variables

YAP now supports the attributed variables packaged 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 and Fruewirth and Holzbaur's CHR, but other applications have been proposed in the literature.

The command

```
| ?- use_module(library(atts)).
```

enables the 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.

11.1 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:

```
:- 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 builtins 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 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 builtins are rewritten by the system into internal builtins, and that the rewriting process *depends* on the module on which the builtins 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.

11.3 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.

11.4 Displaying Attributes

Attributes are usually presented as goals. The following routines are used by builtin predicates such as `call_residue/2` and by the Prolog top-level to display attributes:

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.

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*.

11.5 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.

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.

11.6 Attribute Examples

The following two examples example is 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.

```
:- 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" `Other=El` was deferred until after the completion of `verify_attribute/3`. Otherwise, there might be a danger of recursively invoking `verify_attribute/3`, which might bind `Var`, which is not allowed inside the scope of `verify_attribute/3`. Deferring unifications into the third argument of `verify_attribute/3` effectively serializes the calls to `verify_attribute/3`.

Assuming that the code resides in the file '`domain.yap`', we can use it via:

```
| ?- use_module(domain).
```

Let's test it:

```
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]).

domain(X,[1,5,6,7]),
domain(Y,[3,4,5,6]),
domain(Z,[1,6,7,8]) ?

yes
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
     X=Y.

Y = X,
domain(X,[5,6]),
domain(Z,[1,6,7,8]) ?

yes
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
     X=Y, Y=Z.

X = 6,
Y = 6,
Z = 6
```

To demonstrate the use of the *Goals* argument of `verify_attributes/3`, we give an implementation of `freeze/2`. We have to name it `myfreeze/2` in order to avoid a name clash with the built-in predicate of the same name.

```
:- module(myfreeze, [myfreeze/2]).

:- use_module(library(atts)).

:- attribute frozen/1.

verify_attributes(Var, Other, Goals) :-
        get_atts(Var, frozen(Fa)), !,       % are we involved?
```

```
        (   var(Other) ->                   % must be attributed then
            (   get_atts(Other, frozen(Fb)) % has a pending goal?
            ->  put_atts(Other, frozen((Fa,Fb))) % rescue conjunction
            ;   put_atts(Other, frozen(Fa)) % rescue the pending goal
            ),
            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 CLP(Q,R) Manual

This Manual documents a Prolog implementation of clp(Q,R), based on SICStus featuring extensible unification via attributed variables.

Edition 1.3.3 December 1995

Christian Holzbaur `christian@ai.univie.ac.at`

12.1 Introduction to CLP(Q,R)

The clp(Q,R) system described in this document is an instance of the general Constraint Logic Programming scheme introduced by [Jaffar & Michaylov 87].

The implementation is at least as complete as other existing clp(R) implementations: It solves linear equations over rational or real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization.

The full clp(Q,R) distribution, including a stand-alone manual and an examples directory that is possibly more up to date than the version in the SICStus Prolog distribution, is available from: http://www.ai.univie.ac.at/clpqr/.

12.2 Referencing CLP(Q,R)

When referring to this implementation of clp(Q,R) in publications, you should use the following reference:

Holzbaur C.: OFAI clp(q,r) Manual, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.

12.3 CLP(QR) Acknowledgments

Acknowledgments

The development of this software was supported by the Austrian Fonds zur Foerderung der Wissenschaftlichen Forschung under grant P9426-PHY. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry for Science and Research.

We include a collection of examples that has been distributed with the Monash University version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

12.4 Solver Interface

Rational numbers are not first class citizens in SICStus Prolog, so rational arithmetics has to be emulated. Because of the emulation it is too expensive to support arithmetics with automatic coercion between all sorts of numbers, like you find it in CommonLisp, for example.

You must choose whether you want to operate in the field of Q (Rationals) or R (Reals):

```
?- use_module(library(clpq)).
```

or

```
?- use_module(library(clpr)).
```

12.5 Notational Conventions

Throughout this chapter, the prompts `clp(q) ?-` and `clp(r) ?-` are used to differentiate between clp(Q) and clp(R) in exemplary interactions.

In general there are many ways to express the same linear relationship. This degree of freedom is manifest in the fact that the printed manual and an actual interaction with the current version of clp(Q,R) may show syntactically different answer constraints, despite the fact the same semantic relationship is being expressed. There are means to control the presentation, see see Section 12.14 [Variable Ordering], page 126. The approximative nature of floating point numbers may also produce numerical differences between the text in this manual and the actual results of clp(R), for a given edition of the software.

12.6 Solver Predicates

The solver interface for both Q and R consists of the following predicates which are exported from module(linear).

`{+Constraint}`

Constraint is a term accepted by the the grammar below. The corresponding constraint is added to the current constraint store and checked for satisfiability. If you want to overload `{}`/1 with other solvers, you can avoid its importation via: `use_module(clpq, []).`

```
Constraint --> C
             | C , C                  conjunction

C -->          Expr =:= Expr           equation
             | Expr = Expr             equation
             | Expr < Expr             strict inequation
             | Expr > Expr             strict inequation
```

```
                    | Expr =< Expr          nonstrict inequation
                    | Expr >= Expr          nonstrict inequation
                    | Expr =\= Expr         disequation

          Expr -->    variable              Prolog variable
                    | number                floating point or integer
                    | + Expr                unary plus
                    | - Expr                unary minus
                    | Expr + Expr           addition
                    | Expr - Expr           subtraction
                    | Expr * Expr           multiplication
                    | Expr / Expr           division
                    | abs(Expr)             absolute value
                    | sin(Expr)             trigonometric sine
                    | cos(Expr)             trigonometric cosine
                    | tan(Expr)             trigonometric tangent
                    | pow(Expr,Expr)        raise to the power
                    | exp(Expr,Expr)        raise to the power
                    | min(Expr,Expr)        minimum of the two
                                                      arguments
                    | max(Expr,Expr)        maximum of the two
                                                      arguments
                    | #(Const)              symbolic numerical
                                                      constant
```

Conjunctive constraints `{-C,C}` have been made part of the syntax in order to enable grouped submission of constraints, which could be exploited by future versions of this software. Symbolic numerical constants are provided for compatibility only, see see Section 12.19 [Monash Examples], page 132.

`entailed(+`*`Constraint`*`)`

Succeeds iff the linear *Constraint* is entailed by the current constraint store. This predicate does not change the state of the constraint store.

```
clp(q) ?- {A =< 4}, entailed(A=\=5).

{A=<4}
yes

clp(q) ?- {A =< 4}, entailed(A=\=3).
no
```

`inf(+`*`Expr`*`, -`*`Inf`*` )`

Computes the infimum of the linear expression *Expr* and unifies it with *Inf*. Failure indicates unboundedness.

`sup(+`*`Expr`*`, -`*`Sup`*`)`

Computes the supremum of the linear expression *Expr* and unifies it with *Sup*. Failure indicates unboundedness.

```
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11,
```

```
                    X+3*Y =< 15, Z = 30*X+50*Y
                }, sup(Z, Sup).

Sup = 310,
{Z=30*X+50*Y},
{X+1/2*Y=<8}

{X+3*Y=<15},
{X+2*Y=<11}
```

`minimize(+Expr)`
: Computes the infimum of the linear expression *Expr* and equates it with the expression, i.e. as if defined as:

```
minimize(Expr) :- inf(Expr, Expr).
```

`maximize(+Expr)`
: Computes the supremum of the linear expression *Expr* and equates it with the expression.

```
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11,
            X+3*Y =< 15, Z = 30*X+50*Y
          }, maximize(Z).

X = 7,
Y = 2,
Z = 310
```

`bb_inf(+Ints, +Expr, -Inf)`
: Computes the infimum of the linear expression *Expr* under the additional constraint that all of variables in the list *Ints* assume integral values at the infimum. This allows for the solution of mixed integer linear optimization problems, see see Section 12.21 [A Mixed Integer Linear Optimization Example], page 133.

`ordering(+Spec)`
: Provides a means to control one aspect of the presentation of the answer constraints, see see Section 12.14 [Variable Ordering], page 126.

12.7 Unification

Equality constraints are added to the store implicitly each time variables that have been mentioned in explicit constraints are bound - either to another such variable or to a number.

```
clp(r) ?- {2*A+3*B=C/2}, C=10.0, A=B.

A = 1.0,
B = 1.0,
C = 10.0
```

Is equivalent modulo rounding errors to

```
clp(r) ?- {2*A+3*B=C/2, C=10, A=B}.
```

```
A = 1.0,
B = 0.9999999999999999,
C = 10.0
```

The shortcut bypassing the use of {}q/1 is allowed and makes sense because the interpretation of this equality in Prolog and clp(R) coincides. In general, equations involving interpreted functors, +/2 in this case, must be fed to the solver explicitly:

```
clp(r) ?- X=3.0+1.0, X=4.0.

no
```

Further, variables known by clp(R) may be bound directly to floats only. Likewise, variables known by clp(Q) may be bound directly to rational numbers only, see see Section 12.12 [Numerical Precision and Rationals], page 122. Failing to do so is rewarded with an exception:

```
clp(q) ?- {2*A+3*B=C/2}, C=10.0, A=B.

[ERROR: not.normalized(10.0)]
```

This is because 10.0 is not a rational constant. To make clp(Q) happy you have to say:

```
clp(q) ?- {2*A+3*B=C/2}, C=rat(10,1), A=B.

A = 1,
B = 1,
C = 10
```

If you use {}/1, you don't have to worry about such details. Alternatively, you may use the automatic expansion facility, check see Section 12.18 [Syntactic Sugar], page 131.

12.8 Feedback and Bindings

What was covered so far was how the user populates the constraint store. The other direction of the information flow consists of the success and failure of the above predicates and the binding of variables to numerical values and the aliasing of variables. Example:

```
clp(r) ?- {A-B+C=10, C=5+5}.

B = A,
C = 10.0
```

The linear constraints imply A=B and the solver consequently exports this binding to the Prolog world, which is manifest in the fact that the test A==B will succeed. More about answer presentation in see Section 12.13 [Projection and Redundancy Elimination], page 125.

12.9 Linearity and Nonlinear Residues

The clp(Q,R) system is restricted to deal with linear constraints because the decision algorithms for general nonlinear constraints are prohibitively expensive to run. If you need

this functionality badly, you should look into symbolic algebra packages. Although the clp(Q,R) system cannot solve nonlinear constraints, it will collect them faithfully in the hope that through the addition of further (linear) constraints they might get simple enough to solve eventually. If an answer contains constraints, you have to be aware of the fact that success is qualified modulo the existence of a solution to the system of residual (nonlinear) constraints:

```
clp(r) ?- {sin(X) = cos(X)}.

nonlin:{sin(X)-cos(X)=0.0}
```

There are indeed infinitely many solutions to this constraint (`X = 0.785398 + n*Pi`), but clp(Q,R) has no direct means to find and represent them.

The systems goes through some lengths to recognize linear expressions as such. The method is based on a normal form for multivariate polynomials. In addition, some simple isolation axioms, that can be used in equality constraints, have been added. The current major limitation of the method is that full polynomial division has not been implemented.

This is an example where the isolation axioms are sufficient to determine the value of X.

```
clp(r) ?- {sin(cos(X)) = 1/2}.

X = 1.0197267436954502
```

If we change the equation into an inequation, clp(Q,R) gives up:

```
clp(r) ?- {sin(cos(X)) < 1/2}.

nonlin:{sin(cos(X))-0.5!0.0}
```

The following is easy again:

```
clp(r) ?- {sin(X+2+2)/sin(4+X) = Y}.

Y = 1.0
```

And so is this:

```
clp(r) ?- {(X+Y)*(Y+X)/X = Y*Y/X+99}.

{Y=49.5-0.5*X}
```

An ancient symbol manipulation benchmark consists in rising the expression `X+Y+Z+1` to the 15th power:

```
clp(q) ?- {exp(X+Y+Z+1,15)=0}.

nonlin:{Z^15+Z^14*15+Z^13*105+Z^12*455+Z^11*1365+Z^10*3003+...
         ... polynomial continues for a few pages ...
        =0}
```

Computing its roots is another story.

12.10 How Nonlinear Residues are made to disappear

Binding variables that appear in nonlinear residues will reduce the complexity of the nonlinear expressions and eventually results in linear expressions:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.

nonlin:{Y*2-X^2*2+Y*X*2+X*2+1=0}
```

Equating X and Y collapses the expression completely and even determines the values of the two variables:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.

X = -1/4,
Y = -1/4
```

12.11 Isolation Axioms

These axioms are used to rewrite equations such that the variable to be solved for is moved to the left hand side and the result of the evaluation of the right hand side can be assigned to the variable. This allows, for example, to use the exponentiation operator for the computation of roots and logarithms, see below.

`A = B * C` Residuates unless B or C is ground or A and B or C are ground.

`A = B / C` Residuates unless C is ground or A and B are ground.

`X = min(Y,Z)`
: Residuates unless Y and Z are ground.

`X = max(Y,Z)`
: Residuates unless Y and Z are ground.

`X = abs(Y)`
: Residuates unless Y is ground.

`X = pow(Y,Z)`, `X = exp(Y,Z)`
: Residuates unless any pair of two of the three variables is ground. Example:

```
clp(r) ?- { 12=pow(2,X) }.

X = 3.5849625007211565

clp(r) ?- { 12=pow(X,3.585) }.

X = 1.9999854993443926

clp(r) ?- { X=pow(2,3.585) }.

X = 12.000311914286545
```

`X = sin(Y)`
: Residuates unless X or Y is ground. Example:

```
clp(r) ?- { 1/2 = sin(X) }.

X = 0.5235987755982989
```

`X = cos(Y)`
: Residuates unless X or Y is ground.

`X = tan(Y)`
: Residuates unless X or Y is ground.

12.12 Numerical Precision and Rationals

The fact that you can switch between clp(R) and clp(Q) should solve most of your numerical problems regarding precision. Within clp(Q), floating point constants will be coerced into rational numbers automatically. Transcendental functions will be approximated with rationals. The precision of the approximation is limited by the floating point precision. These two provisions allow you to switch between clp(R) and clp(Q) without having to change your programs.

What is to be kept in mind however is the fact that it may take quite big rationals to accommodate the required precision. High levels of precision are for example required if your linear program is ill-conditioned, i.e., in a full rank system the determinant of the coefficient matrix is close to zero. Another situation that may call for elevated levels of precision is when a linear optimization problem requires exceedingly many pivot steps before the optimum is reached.

If your application approximates irrational numbers, you may be out of space particularly soon. The following program implements N steps of Newton's approximation for the square root function at point 2.

```
%
% from file: library('clpqr/examples/root')
%

root(N, R) :-
  root(N, 1, R).

root(0, S, R) :- !, S=R.
root(N, S, R) :-
  N1 is N-1,
  { S1 = S/2 + 1/S },
  root(N1, S1, R).
```

It is known that this approximation converges quadratically, which means that the number of correct digits in the decimal expansion roughly doubles with each iteration. Therefore the numerator and denominator of the rational approximation have to grow likewise:

```
clp(q) ?- use_module(library('clpqr/examples/root')).
clp(q) ?- root(3,R),print_decimal(R,70).
1.4142156862 7450980392 1568627450 9803921568 6274509803 9215686274
5098039215

R = 577/408

clp(q) ?- root(4,R),print_decimal(R,70).
```

```
1.4142135623 7468991062 6295578890 1349101165 5962211574 4044584905
0192000543

R = 665857/470832

clp(q) ?- root(5,R),print_decimal(R,70).

1.4142135623 7309504880 1689623502 5302436149 8192577619 7428498289
4986231958

R = 886731088897/627013566048

clp(q) ?- root(6,R),print_decimal(R,70).

1.4142135623 7309504880 1688724209 6980785696 7187537723 4001561013
1331132652

R = 157258404803291863335321 7/11119848443498681379381 12

clp(q) ?- root(7,R),print_decimal(R,70).

1.4142135623 7309504880 1688724209 6980785696 7187537694 8073176679
7379907324

R = 4946041176255201878775086487573351061418968498177 /
    3497379255757941172020851852070562919437964212608
```

Iterating for 8 steps produces no further change in the first 70 decimal digits of `sqrt(2)`. After 15 steps the approximating rational number has a numerator and a denominator with 12543 digits each, and the next step runs out of memory.

Another irrational number that is easily computed is `e`. The following program implements an alternating series for `1/e`, where the absolute value of last term is an upper bound on the error.

```
%
% from file: library('clpqr/examples/root')
%

e(N, E) :-
  { Err =:= exp(10,-(N+2)), Half =:= 1/2 },
  inv_e_series(Half, Half, 3, Err, Inv.E),
  { E =:= 1/Inv_E }.

inv_e_series(Term, S0, ., Err, Sum) :-
  { abs(Term) =< Err }, !,
  S0 = Sum.

inv_e_series(Term, S0, N, Err, Sum) :-
```

```
    N1 is N+1,
    { Term1 =:= -Term/N, S1 =:= Term1+S0 },
    inv_e_series(Term1, S1, N1, Err, Sum).
```

The computation of the rational number E that approximates `e` up to at least 1000 digits in its decimal expansion requires the evaluation of 450 terms of the series, i.e. 450 calls of inv.e. series/5.

```
clp(q) ?- e(1000,E).

E = 7149056228932760213666809592072842334290744221392610955845565494
    3708750229467761730471738895197792271346693089326102132000338192
    0131874187833985420922688804220167840319199699494193852403223700
    5853832741544191628747052136402176941963825543565900589161585723
    4023097417605004829991929283045372355639145644588174733401360176
    9953973706537274133283614740902771561159913069917833820285608440
    3104966899999651928637634656418969027076699082888742481392304807
    9484725489080844360397606199771786024695620205344042765860581379
    3538290451208322129898069978107971226873160872046731879753034549
    3130492167474809196348846916421782850086985668680640425192038155
    4902863298351349469211627292865440876581064873866786120098602898
    8799130098877372097360065934827751120659213470528793143805903554
    7928682131082164366007016698761961066948371407368962539467994627
    1374858249110795976398595034606994740186040425117101588480000000
    0000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000000000000000000000
    /
    2629990810403002651095959155503002285441272170673105334466808931
    6863103901346024240326549035084528682487048064823380723787110941
    6809235187356318780972302796570251102928552003708556939314795678
    1978390674393498540663747334079841518303636625888963910391440709
    0887345797303470959207883316838346973393937778363411195624313553
    8835644822353659840936818391050630360633734935381528275392050975
    7271468992840907541350345459011192466892177866882264242860412188
    0652112744642450404625763019639086944558899249788084559753723892
    1643188991444945360726899532023542969572584363761073528841147012
    2634218045463494055807073778490814692996517359952229262198396182
    1838930043528583109973872348193806830382584040536394640895148751
    0766256738740729894909630785260101721285704616818889741995949666
    6303289703199393801976334974240815397920213059799071915067856758
    6716458821062645562512745336709063396510021681900076680696945309
    3660590933279867736747926648678738515702777431353845466199680991
    73361873421152165477774911660108200059
```

The decimal expansion itself looks like this:

```
clp(q) ?- e(1000, E), print_decimal(E, 1000).
2.
7182818284 5904523536 0287471352 6624977572 4709369995 9574966967
6277240766 3035354759 4571382178 5251664274 2746639193 2003059921
```

```
8174135966 2904357290 0334295260 5956307381 3232862794 3490763233
8298807531 9525101901 1573834187 9307021540 8914993488 4167509244
7614606680 8226480016 8477411853 7423454424 3710753907 7744992069
5517027618 3860626133 1384583000 7520449338 2656029760 6737113200
7093287091 2744374704 7230696977 2093101416 9283681902 5515108657
4637721112 5238978442 5056953696 7707854499 6996794686 4454905987
9316368892 3009879312 7736178215 4249992295 7635148220 8269895193
6680331825 2886939849 6465105820 9392398294 8879332036 2509443117
3012381970 6841614039 7019837679 3206832823 7646480429 5311802328
7825098194 5581530175 6717361332 0698112509 9618188159 3041690351
5988885193 4580727386 6738589422 8792284998 9208680582 5749279610
4841984443 6346324496 8487560233 6248270419 7862320900 2160990235
3043699418 4914631409 3431738143 6405462531 5209618369 0888707016
7683964243 7814059271 4563549061 3031072085 1038375051 0115747704
1718986106 8739696552 1267154688 9570350354
```

12.13 Projection and Redundancy Elimination

Once a derivation succeeds, the Prolog system presents the bindings for the variables in the query. In a CLP system, the set of answer constraints is presented in analogy. A complication in the CLP context are variables and associated constraints that were not mentioned in the query. A motivating example is the familiar mortgage relation:

```
%
% from file: library('clpqr/examples/mg')
%

mg(P,T,I,B,MP):-
  {
    T = 1,
    B + MP = P * (1 + I)
  }.
mg(P,T,I,B,MP):-
  {
    T > 1,
    P1 = P * (1 + I) - MP,
    T1 = T - 1
  }, mg(P1, T1, I, B, MP).
```

A sample query yields:

```
clp(r) ?- use_module(library('clpqr/examples/mg')).

clp(r) ?- mg(P,12,0.01,B,Mp).

{B=1.1268250301319698*P-12.682503013196973*Mp}
```

Without projection of the answer constraints onto the query variables we would observe the following interaction:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=12.682503013196973*_A-11.682503013196971*P},
{Mp= -(_A)+1.01*P},
{_B=2.01*_A-1.01*P}
{_C=3.0301*_A-2.0301*P},
{_D=4.060401000000001*_A-3.0604009999999997*P},
{_E=5.101005010000001*_A-4.10100501*P},
{_F=6.152015060100001*_A-5.152015060099999*P},
{_G=7.213535210701001*_A-6.213535210700999*P},
{_H=8.285670562808011*_A-7.285670562808009*P},
{_I=9.368527268436091*_A-8.36852726843609*P},
{_J=10.462212541120453*_A-9.46221254112045*P},
{_K=11.566834666531657*_A-10.566834666531655*P}
```

The variables *_A* ... *_K* are not part of the query, they originate from the mortgage program proper. Although the latter answer is equivalent to the former in terms of linear algebra, most users would prefer the former.

12.14 Variable Ordering

In general, there are many ways to express the same linear relationship between variables. clp(Q,R) does not care to distinguish between them, but the user might. The predicate ordering(+*Spec*) gives you some control over the variable ordering. Suppose that instead of *B*, you want *Mp* to be the defined variable:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=1.1268250301319698*P-12.682503013196973*Mp}
```

This is achieved with:

```
clp(r) ?- mg(P,12,0.01,B,Mp), ordering([Mp]).

{Mp= -0.0788487886783417*B+0.08884878867834171*P}
```

One could go one step further and require *P* to appear before (to the left of) *B* in a addition:

```
clp(r) ?- mg(P,12,0.01,B,Mp), ordering([Mp,P]).

{Mp=0.08884878867834171*P-0.0788487886783417*B}
```

Spec in ordering(+*Spec*) is either a list of variables with the intended ordering, or of the form `A<B`. The latter form means that *A* goes to the left of *B*. In fact, `ordering([A,B,C,D])` is shorthand for:

```
ordering(A < B), ordering(A < C), ordering(A < D),
ordering(B < C), ordering(B < D),
ordering(C < D)
```

The ordering specification only affects the final presentation of the constraints. For all other operations of clp(Q,R), the ordering is immaterial. Note that ordering/1 acts like

a constraint: you can put it anywhere in the computation, and you can submit multiple specifications.

```
clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp).

{B= -12.682503013196973*Mp+1.1268250301319698*P}

yes
clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp), ordering(P < Mp).

{P=0.8874492252651537*B+11.255077473484631*Mp}
```

12.15 Turning Answers into Terms

In meta-programming applications one needs to get a grip on the results computed by the clp(Q,R) solver. The SISCtus Prolog predicate `call_residue/2` provides this functionality:

```
clp(r) ?- call_residue({2*A+B+C=10,C-D=E,A<10}, Constraints).

Constraints = [
               [A]-{A<10.0},
               [B]-{B=10.0-2.0*A-C},
               [D]-{D=C-E}
              ]
```

12.16 Projecting Inequalities

As soon as linear inequations are involved, projection gets more demanding complexity wise. The current clp(Q,R) version uses a Fourier-Motzkin algorithm for the projection of linear inequalities. The choice of a suitable algorithm is somewhat dependent on the number of variables to be eliminated, the total number of variables, and other factors. It is quite easy to produce problems of moderate size where the elimination step takes some time. For example, when the dimension of the projection is 1, you might be better off computing the supremum and the infimum of the remaining variable instead of eliminating n-1 variables via implicit projection.

In order to make answers as concise as possible, redundant constraints are removed by the system as well. In the following set of inequalities, half of them are redundant.

```
%
% from file: library('clpqr/examples/elimination')
%

example(2, [X0,X1,X2,X3,X4]) :-
  {
       +87*X0 +52*X1 +27*X2 -54*X3 +56*X4 =< -93,
       +33*X0 -10*X1 +61*X2 -28*X3 -29*X4 =<  63,
       -68*X0  +8*X1 +35*X2 +68*X3 +35*X4 =< -85,
       +90*X0 +60*X1 -76*X2 -53*X3 +24*X4 =< -68,
```

```
        -95*X0 -10*X1 +64*X2 +76*X3 -24*X4 =<  33,
        +43*X0 -22*X1 +67*X2 -68*X3 -92*X4 =< -97,
        +39*X0  +7*X1 +62*X2 +54*X3 -26*X4 =< -27,
        +48*X0 -13*X1  +7*X2 -61*X3 -59*X4 =<  -2,
        +49*X0 -23*X1 -31*X2 -76*X3 +27*X4 =<   3,
        -50*X0 +58*X1  -1*X2 +57*X3 +20*X4 =<   6,
        -13*X0 -63*X1 +81*X2  -3*X3 +70*X4 =<  64,
        +20*X0 +67*X1 -23*X2 -41*X3 -66*X4 =<  52,
        -81*X0 -44*X1 +19*X2 -22*X3 -73*X4 =< -17,
        -43*X0  -9*X1 +14*X2 +27*X3 +40*X4 =<  39,
        +16*X0 +83*X1 +89*X2 +25*X3 +55*X4 =<  36,
        +2*X0  +40*X1 +65*X2 +59*X3 -32*X4 =<  13,
        -65*X0 -11*X1 +10*X2 -13*X3 +91*X4 =<  49,
        +93*X0 -73*X1 +91*X2  -1*X3 +23*X4 =< -87
    }.
```

Consequently, the answer consists of the system of nine non-redundant inequalities only:

```
clp(q) ?- use_module(library('clpqr/examples/elimination')).
clp(q) ?- example(2, [X0,X1,X2,X3,X4]).

{X0-2/17*X1-35/68*X2-X3-35/68*X4?=5/4},
{X0-73/93*X1+91/93*X2-1/93*X3+23/93*X4=<-29/31},
{X0-29/25*X1+1/50*X2-57/50*X3-2/5*X4>=-3/25},
{X0+7/39*X1+62/39*X2+18/13*X3-2/3*X4=<-9/13},
{X0+2/19*X1-64/95*X2-4/5*X3+24/95*X4>=-33/95},
{X0+2/3*X1-38/45*X2-53/90*X3+4/15*X4=<-34/45},
{X0-23/49*X1-31/49*X2-76/49*X3+27/49*X4=<3/49},
{X0+44/81*X1-19/81*X2+22/81*X3+73/81*X4>=17/81},
{X0+9/43*X1-14/43*X2-27/43*X3-40/43*X4>=-39/43}
```

The projection (the shadow) of this polyhedral set into the *X0*,*X1* space can be computed via the implicit elimination of non-query variables:

```
clp(q) ?- example(2, [X0,X1--.]).

{X0+2619277/17854273*X1>=-851123/17854273},
{X0+6429953/16575801*X1=<-12749681/16575801},
{X0+19130/1213083*X1>=795400/404361},
{X0-1251619/3956679*X1?=21101146/3956679},
{X0+601502/4257189*X1>=220850/473021}
```

Projection is quite a powerful concept that leads to surprisingly terse executable specifications of nontrivial problems like the computation of the convex hull from a set of points in an n-dimensional space: Given the program

```
%
% from file: library('clpqr/examples/elimination')
%
conv.hull(Points, Xs) :-
  lin_comb(Points, Lambdas, Zero, Xs),
```

```
  zero(Zero),
  polytope(Lambdas).

polytope(Xs) :-
  positive_sum(Xs, 1).

  positive_sum([], Z) :- {Z=0}.
  positive_sum([X--Xs], SumX) :-
    {X >= 0, SumX = X+Sum },
    positive_sum(Xs, Sum).

zero([]).
zero([Z--Zs]) :- {Z=0}, zero(Zs).

lin_comb([], [], S1, S1).
lin_comb([Ps--Rest], [K--Ks], S1, S3) :-
  lin_comb_r(Ps, K, S1, S2),
  lin_comb(Rest, Ks, S2, S3).

  lin_comb_r([], ., [], []).
  lin_comb_r([P--Ps], K, [S--Ss], [Kps--Ss1]) :-
    { Kps = K*P+S },
    lin_comb_r(Ps, K, Ss, Ss1).
```

we can post the following query:

```
clp(q) ?- conv.hull([ [1,1], [2,0], [3,0], [1,2], [2,2] ], [X,Y]).

{Y=<2},
{X+1/2*Y=<3},
{X>=1},
{Y>=0},
{X+Y>=2}
```

This answer is easily verified graphically:

```
 |
2-   *   *
 |
 |
1|   *
 |
 |
0 ---|---*---*----
     1   2   3
```

The convex hull program directly corresponds to the mathematical definition of the convex hull. What does the trick in operational terms is the implicit elimination of the Lambdas from the program formulation. Please note that this program does not limit the number of points or the dimension of the space they are from. Please note further that

quantifier elimination is a computationally expensive operation and therefore this program is only useful as a benchmark for the projector and not so for the intended purpose.

12.17 Why Disequations

A beautiful example of disequations at work is due to [Colmerauer 90]. It addresses the task of tiling a rectangle with squares of all-different, a priori unknown sizes. Here is a translation of the original Prolog-III program to clp(Q,R):

```
%
% from file: library('clpqr/examples/squares')
filled_rectangle( A, C) :-
  { A >= 1 },
  distinct_squares( C),
  filled_zone( [-1,A,1], _, C, []).

distinct_squares( []).
distinct_squares( [B|C]) :-
  { B > 0 },
  outof( C, B),
  distinct_squares( C).

outof( [],     _).
outof( [B1|C], B) :-
  { B =\= B1 },       % *** note disequation ***
  outof( C, B).

filled_zone( [V|L], [V|L], C0, C0) :-
  { V >= 0 }.
filled_zone( [V|L], L3, [B|C], C2) :-
  { V < 0 },
  placed_square( B, L, L1),
  filled_zone( L1, L2, C, C1),
  { Vb=V+B },
  filled_zone( [Vb,B|L2], L3, C1, C2).

placed_square( B, [H,H0,H1|L], L1) :-
  { B > H, H0=0, H2=H+H1 },
  placed_square( B, [H2|L], L1).
placed_square( B, [B,V|L], [X|L]) :-
  { X=V-B }.
placed_square( B, [H|L], [X,Y|L]) :-
  { B < H, X= -B, Y=H-B }.
```

There are no tilings with less than nine squares except the trivial one where the rectangle equals the only square. There are eight solutions for nine squares. Six further solutions are rotations of the first two.

```
clp(q) ?- use_module(library('clpqr/examples/squares')).
```

```
clp(q) ?- filled_rectangle(A, Squares).

A = 1,f
Squares = [1] ? ;

A = 33/32,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61]
```

Depending on your hardware, the above query may take a few minutes. Supplying the knowledge about the minimal number of squares beforehand cuts the computation time by a factor of roughly four:

```
clp(q) ?- length(Squares, 9), filled.rectangle(A, Squares).

A = 33/32,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61]
```

12.18 Syntactic Sugar

There is a package that transforms programs and queries from a eval-quote variant of clp(Q,R) into corresponding programs and queries in a quote-eval variant. Before you use it, you need to know that in an eval-quote language, all symbols are interpreted unless explicitly quoted. This means that interpreted terms cannot be manipulated syntactically directly. Meta-programming in a CLP context by definition manipulates interpreted terms, therefore you need `quote/1` (just as in LISP) and some means to put syntactical terms back to their interpreted life: `{}/1`.

In a quote-eval language, meta-programming is (pragmatically) simpler because everything is implicitly quoted until explicitly evaluated. On the other hand, now object programming suffers from the dual inconvenience.

We chose to make our version of clp(Q,R) of the quote-eval type because this matches the intended use of the already existing boolean solver of SICStus. In order to keep the users of the eval-quote variant happy, we provide a source transformation package. It is activated via:

```
| ?- use_module(library('clpqr/expand')).
```

Loading the package puts you in a mode where the arithmetic functors like `+/2`, `*/2` and all numbers (functors of arity 0) are interpreted semantically.

```
clp(r) ?- 2+2=X. X = 4.0
```

The package works by purifying programs and queries in the sense that all references to interpreted terms are made explicit. The above query is expanded prior to evaluation into:

```
linear:{2.0+2.0=X}
```

The same mechanism applies when interpreted terms are nested deeper:

```
some_predicate(10, f(A+B/2), 2*cos(A))
```

Expands into:

```
linear:{Xc=2.0*cos(A)},
linear:{Xb=A+B/2},
linear:{Xa=10.0},
some_predicate(Xa, f(Xb), Xc)
```

This process also applies when files are consulted or compiled. In fact, this is the only situation where expansion can be applied with relative safety. To see this, consider what happens when the toplevel evaluates the expansion, namely some calls to the clp(Q,R) solver, followed by the call of the purified query. As we learned in see Section 12.8 [Feedback and Bindings], page 119, the solver may bind variables, which produces a goal with interpreted functors in it (numbers), which leads to another stage of expansion, and so on.

We recommend that you only turn on expansion temporarily while consulting or compiling files needing expansion with `expand/0` and `noexpand/0`.

12.19 Monash Examples

This collection of examples has been distributed with the Monash University Version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

In order to execute the examples, a small compatibility package has to be loaded first:

```
clp(r) ?- use_module(library('clpqr/monash')).
```

Then, assuming you are using clp(R):

```
clp(r) ?- expand, [library('clpqr/examples/monash/rkf45')],
          noexpand.

clp(r) ?- go.
Point    0.00000 :    0.75000    0.00000
Point    0.50000 :    0.61969    0.47793
Point    1.00000 :    0.29417    0.81233
Point    1.50000 :   -0.10556    0.95809
Point    2.00000 :   -0.49076    0.93977
Point    2.50000 :   -0.81440    0.79929
Point    3.00000 :   -1.05440    0.57522

Iteration finished
------------------
 439 derivative evaluations
```

12.20 Compatibility Notes

The Monash examples have been written for clp(R). Nevertheless, all but rkf45 complete nicely in clp(Q). With `rkf45`, clp(Q) runs out of memory. This is an instance of the problem discussed in see Section 12.12 [Numerical Precision and Rationals], page 122.

The Monash University clp(R) interpreter features a `dump/n` predicate. It is used to print the target variables according to the given ordering. Within this version of clp(Q,R), the corresponding functionality is provided via `ordering/1`. The difference is that `ordering/1` does only specify the ordering of the variables and no printing is performed. We think Prolog has enough predicates to perform output already. You can still run the examples referring to `dump/n` from the Prolog toplevel:

```
clp(r) ?- expand, [library('clpqr/examples/monash/mortgage')], noexpand.

% go2
%
clp(r) ?- mg(P,120,0.01,0,MP), dump([P,MP]).

{P=69.7005220313972*MP}

% go3
%
clp(r) ?- mg(P,120,0.01,B,MP), dump([P,B,MP]).

{P=0.30299477968602706*B+69.7005220313972*MP}

% go4
%
clp(r) ?- mg(999, 3, Int, 0, 400), dump.

nonlin:{_B-_B*Int+.A+400.0=0.0},
nonlin:{_A-_A*Int+400.0=0.0},
{_B=599.0+999.0*Int}
```

12.21 A Mixed Integer Linear Optimization Example

In this section we are going to exercise our solver a little by the computation of a small mixed integer optimization problem (MIP) from miplib, a collection of MIP models, housed at Rice University. Here are the original comments on the example:

```
NAME:          flugpl
ROWS:          18
COLUMNS:       18
INTEGER:       11
NONZERO:       46
BEST SOLN:     1201500 (opt)
LP SOLN:       1167185.73
SOURCE:        Harvey M. Wagner
               John W. Gregory (Cray Research)
               E. Andrew Boyd (Rice University)
APPLICATION:   airline model
COMMENTS:      no integer variables are binary
%
```

```
% from file: library('clpqr/examples/mip')
%
example(flugpl, Obj, Vs, Ints, []) :-
  Vs = [ Anm1,Anm2,Anm3,Anm4,Anm5,Anm6,
         Stm1,Stm2,Stm3,Stm4,Stm5,Stm6,
         UE1,UE2,UE3,UE4,UE5,UE6],
  Ints = [Stm6, Stm5, Stm4, Stm3, Stm2,
          Anm6, Anm5, Anm4, Anm3, Anm2, Anm1],

  Obj =    2700*Stm1 + 1500*Anm1 + 30*UE1
         + 2700*Stm2 + 1500*Anm2 + 30*UE2
         + 2700*Stm3 + 1500*Anm3 + 30*UE3
         + 2700*Stm4 + 1500*Anm4 + 30*UE4
         + 2700*Stm5 + 1500*Anm5 + 30*UE5
         + 2700*Stm6 + 1500*Anm6 + 30*UE6,

  allpos(Vs),
  {  Stm1 = 60, 0.9*Stm1 +1*Anm1 -1*Stm2 = 0,
     0.9*Stm2 +1*Anm2 -1*Stm3 = 0, 0.9*Stm3 +1*Anm3 -1*Stm4 = 0,
     0.9*Stm4 +1*Anm4 -1*Stm5 = 0, 0.9*Stm5 +1*Anm5 -1*Stm6 = 0,
     150*Stm1 -100*Anm1 +1*UE1 >= 8000,
     150*Stm2 -100*Anm2 +1*UE2 >= 9000,
     150*Stm3 -100*Anm3 +1*UE3 >= 8000,
     150*Stm4 -100*Anm4 +1*UE4 >= 10000,
     150*Stm5 -100*Anm5 +1*UE5 >= 9000,
     150*Stm6 -100*Anm6 +1*UE6 >= 12000,
     -20*Stm1 +1*UE1 =< 0, -20*Stm2 +1*UE2 =< 0, -20*Stm3 +1*UE3 =< 0,
     -20*Stm4 +1*UE4 =< 0, -20*Stm5 +1*UE5 =< 0, -20*Stm6 +1*UE6 =< 0,
     Anm1 =< 18, 57 =< Stm2, Stm2 =< 75, Anm2 =< 18,
     57 =< Stm3, Stm3 =< 75, Anm3 =< 18, 57 =< Stm4,
     Stm4 =< 75, Anm4 =< 18, 57 =< Stm5, Stm5 =< 75,
     Anm5 =< 18, 57 =< Stm6, Stm6 =< 75, Anm6 =< 18
   }.

  allpos([]).
  allpos([X|Xs]) :- {X >= 0}, allpos(Xs).
```

We can first check whether the relaxed problem has indeed the quoted infimum:

```
clp(r) ?- example(flugpl, Obj, _, _, _), inf(Obj, Inf).

Inf = 1167185.7255923203
```

Computing the infimum under the additional constraints that *Stm6*, *Stm5*, *Stm4*, *Stm3*, *Stm2*, *Anm6*, *Anm5*, *Anm4*, *Anm3*, *Anm2*, *Anm1* assume integer values at the infimum is computationally harder, but the query does not change much:

```
clp(r) ?- example(flugpl, Obj, _, Ints, _), bb_inf(Ints, Obj, Inf).

Inf = 1201500.0000000005
```

12.22 Implementation Architecture

The system consists roughly of the following components:

- A polynomial normal form expression simplification mechanism.
- A solver for linear equations [Holzbaur 92].
- A simplex algorithm to decide linear inequalities [Holzbaur 94].

12.23 Fragments and Bits

The internal data structure for rational numbers is `rat(Num,Den)`. *Den* is always positive, i.e. the sign of the rational number is the sign of *Num*. Further, *Num* and *Den* are relative prime. Note that integer *N* looks like `rat(N,1)` in this representation. You can control printing of terms with `portray/1`.

Partial Evaluation

Once one has a working solver, it is obvious and attractive to run the constraints in a clause definition at read time or compile time and proceed with the answer constraints in place of the original constraints. This gets you constant folding and in fact the full algebraic power of the solver applied to the avoidance of computations at runtime. The mechanism to realize this idea is to use `call_residue/2` for the expansion of `{}/1`.

Asserting with Constraints

If you use the dynamic data base, the clauses you assert might have constraints on the variables occurring in the clause. This should work as follows:

```
clp(r) ?- {A < 10}, assert(p(A)).

{A < 10.0}
yes

clp(r) ?- p(X).

{X<10.0}
```

YAP currently does not implement this feature.

12.24 CLPQR bugs

- The fuzzy comparison of floats is the source for all sorts of weirdness. If a result in R surprises you, try to run the program in Q before you send me a bug report.
- The projector for floundered nonlinear relations keeps too many variables. Its output is rather unreadable.
- Disequations are not projected properly.
- This list is probably incomplete.

Please send bug reports to `christian@ai.univie.ac.at`.

12.25 CLPQR References

[Colmerauer 90] Colmerauer A.: An Introduction to Prolog III, Communications of the ACM, 33(7), 69-90, 1990.

[Heintze et al. 87] Heintze N., Jaffar J., Michaylov S., Stuckey P., Yap R.: The CLP(R) Programmers Manual, Monash University, Clayton, Victoria, Australia, Department of Computer Science, 1987.

[Holzbaur 92] Holzbaur C.: A High-Level Approach to the Realization of CLP Languages, in Proceedings of the JICSLP92 Post-Conference Workshop on Constraint Logic Programming Systems, Washington D.C., 1992.

[Holzbaur 92] Holzbaur C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification, in Bruynooghe M. & Wirsing M.(eds.), Programming Language Implementation and Logic Programming, Springer, LNCS 631, pp.260- 268, 1992.

[Holzbaur 94] Holzbaur C.: A Specialized, Incremental Solved Form Algorithm for Systems of Linear Inequalities, Austrian Research Institute for Artificial Intelligence, Vienna, TR-94-07, 1994.

[Jaffar & Michaylov 87] Jaffar J., Michaylov S.: Methodology and Implementation of a CLP System, in Lassez J.L.(ed.), Logic Programming - Proceedings of the 4th International Conference - Volume 1, MIT Press, Cambridge, MA, 1987.

13 Constraint Handling Rules

Copyright

13.1 Introduction

Experience from real-life applications using constraint-based programming has shown that typically, one is confronted with a heterogeneous mix of different types of constraints. To be able to express constraints as they appear in the application and to write and combine constraint systems, a special purpose language for writing constraint systems called *constraint handling rules* (CHR) was developed. CHR have been used to encode a wide range of constraint handlers (solvers), including new domains such as terminological and temporal reasoning. Several CHR libraries exist in declarative languages such as Prolog and LISP, worldwide more than 20 projects use CHR. You can find more information about CHR at URL: `http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/chr-intro.html`

The high-level CHR are an excellent tool for rapid prototyping and implementation of constraint handlers. The usual abstract formalism to describe a constraint system, i.e. inference rules, rewrite rules, sequents, formulas expressing axioms and theorems, can be written as CHR in a straightforward way. Starting from this executable specification, the rules can be refined and adapted to the specifics of the application.

The CHR library includes a compiler, which translates CHR programs into Prolog programs on the fly, and a runtime system, which includes a stepper for debugging. Many constraint handlers are provided in the example directory of the library.

CHR are essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved. CHR define both *simplification* of and *propagation* over constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence (e.g. `X>Y,Y>X <=> fail`). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`). Repeatedly applying CHR incrementally simplifies and finally solves constraints (e.g. `A>B,B>C,C>A` leads to `fail`.

With multiple heads and propagation rules, CHR provide two features which are essential for non-trivial constraint handling. The declarative reading of CHR as formulas of first order logic allows one to reason about their correctness. On the other hand, regarding CHR as a rewrite system on logical formulas allows one to reason about their termination and confluence.

In case the implementation of CHR disagrees with your expectations based on this chapter, drop a line to the current maintainer: `christian@ai.univie.ac.at` (Christian Holzbaur).

13.2 Introductory Examples

We define a CHR constraint for less-than-or-equal, `leq`, that can handle variable arguments. This handler can be found in the library as the file `leq.pl`. (The code works regardless of options switched on or off.)

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

handler leq.
constraints leq/2.
:- op(500, xfx, leq).

reflexivity  @ X leq Y <=> X=Y | true.
antisymmetry @ X leq Y , Y leq X <=> X=Y.
idempotence  @ X leq Y \ X leq Y <=> true.
transitivity @ X leq Y , Y leq Z ==> X leq Z.
```

The CHR specify how `leq` simplifies and propagates as a constraint. They implement reflexivity, idempotence, antisymmetry and transitivity in a straightforward way. CHR `reflexivity` states that `X leq Y` simplifies to `true`, provided it is the case that `X=Y`. This test forms the (optional) guard of a rule, a precondition on the applicability of the rule. Hence, whenever we see a constraint of the form `A leq A` we can simplify it to `true`.

The rule `antisymmetry` means that if we find `X leq Y` as well as `Y leq X` in the constraint store, we can replace it by the logically equivalent `X=Y`. Note the different use of `X=Y` in the two rules: In the `reflexivity` rule the equality is a precondition (test) on the rule, while in the `antisymmetry` rule it is enforced when the rule fires. (The reflexivity rule could also have been written as `reflexivity  X leq X <=> true`.)

The rules `reflexivity` and `antisymmetry` are *simplification CHR*. In such rules, the constraints found are removed when the rule applies and fires. The rule `idempotence` is a *simpagation CHR*, only the constraints right of `'\'` will be removed. The rule says that if we find `X leq Y` and another `X leq Y` in the constraint store, we can remove one.

Finally, the rule `transitivity` states that the conjunction `X leq Y, Y leq Z` implies `X leq Z`. Operationally, we add `X leq Z` as (redundant) constraint, without removing the constraints `X leq Y, Y leq Z`. This kind of CHR is called *propagation CHR*.

Propagation CHR are useful, as the query `A leq B,C leq A,B leq C` illustrates: The first two constraints cause CHR `transitivity` to fire and add `C leq B` to the query. This new constraint together with `B leq C` matches the head of CHR `antisymmetry`, `X leq Y, Y leq X`. So the two constraints are replaced by `B=C`. Since `B=C` makes `B` and `C` equivalent, CHR

`antisymmetry` applies to the constraints `A leq B, C leq A`, resulting in `A=B`. The query contains no more CHR constraints, the simplification stops. The constraint handler we built has solved `A leq B, C leq A, B leq C` and produced the answer `A=B, B=C`:

```
A leq B,C leq A,B leq C.
% C leq A, A leq B propagates C leq B by transitivity.
% C leq B, B leq C simplifies to B=C by antisymmetry.
% A leq B, C leq A simplifies to A=B by antisymmetry since B=C.
A=B,B=C.
```

Note that multiple heads of rules are essential in solving these constraints. Also note that this handler implements a (partial) order constraint over any constraint domain, this generality is only possible with CHR.

As another example, we can implement the sieve of Eratosthenes to compute primes simply as (for variations see the handler '`primes.pl`'):

```
:- use_module(library(chr)).
handler eratosthenes.
constraints primes/1,prime/1.

primes(1) <=> true.
primes(N) <=> N>1 | M is N-1,prime(N),primes(M). % generate candidates

absorb(J) @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

The constraint `primes(N)` generates candidates for prime numbers, `prime(M)`, where `M` is between `1` and `N`. The candidates react with each other such that each number absorbs multiples of itself. In the end, only prime numbers remain.

Looking at the two rules defining `primes/1`, note that head matching is used in CHR, so the first rule will only apply to `primes(1)`. The test `N>1` is a guard (precondition) on the second rule. A call with a free variable, like `primes(X)`, will delay (suspend). The third, multi-headed rule `absorb(J)` reads as follows: If there is a constraint `prime(I)` and some other constraint `prime(J)` such that `J mod I =:= 0` holds, i.e. `J` is a multiple of `I`, then keep `prime(I)` but remove `prime(J)` and execute the body of the rule, `true`.

13.3 CHR Library

CHR extend the Prolog syntax by a few constructs introduced in the next sections. Technically, the extension is achieved through the `user:term_expansion/2` mechanism. A file that contains a constraint handler may also contain arbitrary Prolog code. Constraint handling rules can be scattered across a file. Declarations and options should precede rules. There can only be at most one constraint handler per module.

13.3.1 Loading the Library

Before you can load or compile any file containing a constraint handler (solver) written in CHR, the `chr` library module has to be imported:

```
| ?- use_module(library(chr)).
```

It is recommended to include the corresponding directive at the start of your files containing handlers:

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

13.3.2 Declarations

Declarations in files containing CHR affect the compilation and thus the behavior of the rules at runtime.

The mandatory handler declaration precedes any other CHR specific code. Example:

```
handler minmax.
```

A handler name must be a valid Prolog `atom`. Per module, only one constraint handler can be defined.

The constraints must be declared before they are used by rules. With this mandatory declaration one lists the constraints the rules will later talk about. The declaration can be used more than once per handler. Example:

```
constraints  leq/2, minimum/3, maximum/3.
```

The following optional declaration allows for conditional rule compilation. Only the rules mentioned get compiled. Rules are referred to by their names (see Section 13.3.3 [CHR Syntax], page 140). The latest occurrence takes precedence if used more than once per handler. Although it can be put anywhere in the handler file, it makes sense, as with other declarations, to use it early. Example:

```
rules antisymmetry, transitivity.
```

To simplify the handling of operator declarations, in particular during `fcompile/1`, `operator/3` declarations with the same denotation as `op/3`, but taking effect during compilation and loading, are helpful. Example:

```
operator(700, xfx, ::).
operator(600, xfx, :).
```

13.3.3 Constraint Handling Rules, Syntax

A constraint handling rule has one or more heads, an optional guard, a body and an optional name. A *Head* is a *Constraint*. A constraint is a callable Prolog term, whose functor is a declared constraint. The *Guard* is a Prolog goal. The *Body* of a rule is a Prolog goal (including constraints). A rule can be named with a *Name* which can be any Prolog term (including variables from the rule).

There are three kinds of constraint handling rules:

```
Rule           --> [Name @]
                     (Simplification | Propagation | Simpagation)
                        [pragma Pragma].

Simplification --> Heads         <=> [Guard '|'] Body
Propagation    --> Heads         ==> [Guard '|'] Body
Simpagation    --> Heads \ Heads <=> [Guard '|'] Body
```

```
Heads          --> Head | Head, Heads
Head           --> Constraint | Constraint # Id
Constraint     --> a callable term declared as constraint
Id             --> a unique variable

Guard          --> Ask | Ask & Tell
Ask            --> Goal
Tell           --> Goal
Goal           --> a callable term,
                   including conjunction and disjunction etc.

Body           --> Goal

Pragma         --> a conjunction of terms usually referring to
                   one or more heads identified via #/2
```

The symbol '`|`' separates the guard (if present) from the body of a rule. Since '`|`' is read as '`;`' (disjunction) by the reader, care has to be taken when using disjunction in the guard or body of the rule. The top level disjunction will always be interpreted as guard-body separator '`|`', so proper bracketing has to be used, e.g. `a <=> (b;c) | (d;e)` instead of `a <=> b;c | d;e` and `a <=> true | (d;e)` instead of `a <=> (d;e)`.

In simpagation rules, '`\`' separates the heads of the rule into two parts.

Individual head constraints may be tagged with variables via '`#`', which may be used as identifiers in pragma declarations, for example. Constraint identifiers must be distinct variables, not occurring elsewhere in the heads.

Guards test the applicability of a rule. Guards come in two parts, tell and ask, separated by '`&`'. If the '`&`' operator is not present, the whole guard is assumed to be of the ask type.

Declaratively, a rule relates heads and body *provided the guard is true*. A simplification rule means that the heads are true if and only if the body is true. A propagation rule means that the body is true if the heads are true. A simpagation rule combines a simplification and a propagation rule. The rule `Heads1 \ Heads2 <=> Body` is equivalent to the simplification rule `Heads1, Heads2 <=> Heads1, Body`. However, the simpagation rule is more compact to write, more efficient to execute and has better termination behavior than the corresponding simplification rule, since the constraints comprising `Heads1` will not be removed and inserted again.

13.3.4 How CHR work

Each CHR constraint is associated with all rules in whose heads it occurs by the CHR compiler. Every time a CHR constraint is executed (called) or woken and reconsidered, it checks itself the applicability of its associated CHR by *trying* each CHR. By default, the rules are tried in textual order, i.e. in the order they occur in the defining file. To try a CHR, one of its heads is matched against the constraint. Matching succeeds if the constraint is an instance of the head. If a CHR has more than one head, the constraint store is searched for *partner* constraints that match the other heads. Heads are tried from left to right, except that in simpagation rules, the heads to be removed are tried before the

head constraints to be kept (this is done for efficiency reasons). If the matching succeeds, the guard is executed. Otherwise the next rule is tried.

The guard either succeeds or fails. A guard succeeds if the execution of its Ask and Tell parts succeeds and in the ask part no variable that occurs also in the heads was touched or the cause of an instantiation error. The ask guard will fail otherwise. A variable is *touched* if it is unified with a term (including other variables from other constraints) different from itself. Tell guards, on the contrary, are trusted and not checked for that property. If the guard succeeds, the rule applies. Otherwise the next rule is tried.

If the firing CHR is a simplification rule, the matched constraints are removed from the store and the body of the CHR is executed. Similarly for a firing simpagation rule, except that the constraints that matched the heads preceding '\' are kept. If the firing CHR is a propagation rule the body of the CHR is executed without removing any constraints. It is remembered that the propagation rule fired, so it will not fire again with the same constraints if the constraint is woken and reconsidered. If the currently active constraint has not been removed, the next rule is tried.

If the current constraint has not been removed and all rules have been tried, it delays until a variable occurring in the constraint is touched. Delaying means that the constraint is inserted into the constraint store. When a constraint is woken, all its rules are tried again. (This process can be watched and inspected with the CHR debugger, see below.)

13.3.5 Pragmas

Pragmas are annotations to rules and constraints that enable the compiler to generate more specific, more optimized code. A pragma can be a conjunction of the following terms:

`already_in_heads`
: The intention of simplification and simpagation rules is often to combine the heads into a stronger version of one of them. Depending on the strength of the guard, the new constraint may be identical to one of the heads to removed by the rule. This removal followed by addition is inefficient and may even cause termination problems. If the pragma is used, this situation is detected and the corresponding problems are avoided. The pragma applies to all constraints removed by the rule.

`already_in_head(Id)`
: Shares the intention of the previous pragma, but affects only the constraint indicated via *Id*. Note that one can use more than one pragma per rule.

`passive(Id)`
: No code will be generated for the specified constraint in the particular head position. This means that the constraint will not see the rule, it is passive in that rule. This changes the behavior of the CHR system, because normally, a rule can be entered starting from each head constraint. Usually this pragma will improve the efficiency of the constraint handler, but care has to be taken in order not to lose completeness.

  For example, in the handler `leq`, any pair of constraints, say `A leq B, B leq A`, that matches the head `X leq Y , Y leq X` of the `antisymmetry` rule, will also

match it when the constraints are exchanged, `B leq A, A leq B`. Therefore it is enough if a currently active constraint enters this rule in the first head only, the second head can be declared to be passive. Similarly for the `idempotence` rule. For this rule, it is more efficient to declare the first head passive, so that the currently active constraint will be removed when the rule fires (instead of removing the older constraint and redoing all the propagation with the currently active constraint). Note that the compiler itself detects the symmetry of the two head constraints in the simplification rule `antisymmetry`, thus it is automatically declared passive and the compiler outputs `CHR eliminated code for head 2 in antisymmetry`.

```
antisymmetry  X leq Y, Y leq X # Id <=> X=Y
              pragma passive(Id).
idempotence   X leq Y # Id \ X leq Y <=> true
              pragma passive(Id).
transitivity  X leq Y # Id, Y leq Z ==> X leq Z
              pragma passive(Id).
```

Declaring the first head of rule `transitivity` passive changes the behavior of the handler. It will propagate less depending on the order in which the constraints arrive:

```
?- X leq Y, Y leq Z.
X leq Y,
Y leq Z,
X leq Z ?

?- Y leq Z, X leq Y.
Y leq Z,
X leq Y ?

?- Y leq Z, X leq Y, Z leq X.
Y = X,
Z = X ?
```

The last query shows that the handler is still complete in the sense that all circular chains of leq-relations are collapsed into equalities.

13.3.6 Options

Options parametrise the rule compilation process. Thus they should precede the rule definitions. Example:

```
option(check_guard_bindings, off).
```

The format below lists the names of the recognized options together with the acceptable values. The first entry in the lists is the default value.

`option(debug_compile, [off,on]).`

Instruments the generated code such that the execution of the rules may be traced (see Section 13.4 [CHR Debugging], page 148).

`option(check_guard_bindings, [on,off]).`
: Per default, for guards of type ask the CHR runtime system makes sure that no variables are touched or the cause of an instantiation error. These checks may be turned off with this option, i.e. all guards are treated as if they were of the tell variety. The option was kept for backward compatibility. Tell and ask guards offer better granularity.

`option(already_in_store, [off,on]).`
: If this option is on, the CHR runtime system checks for the presence of an identical constraint upon the insertion into the store. If present, the attempted insertion has no effect. Since checking for duplicates for all constraints costs, duplicate removal specific to individual constraints, using a few simpagation rules of the following form instead, may be a better solution.

```
Constraint \ Constraint <=> true.
```

`option(already_in_heads, [off,on]).`
: The intention of simplification and simpagation rules is often to combine the heads into a stronger version of one of them. Depending on the strength of the guard, the new constraint may be identical to one of the heads removed by the rule. This removal followed by addition is inefficient and may even cause termination problems. If the option is enabled, this situation is detected and the corresponding problems are avoided. This option applies to all constraints and is provided mainly for backward compatibility. Better grained control can be achieved with corresponding pragmas. (see Section 13.3.5 [CHR Pragmas], page 142).

The remaining options are meant for CHR implementors only:

```
option(flatten, [on,off]).
option(rule_ordering, [canonical,heuristic]).
option(simpagation_scheme, [single,multi]).
option(revive_scheme, [new,old]).
option(dead_code_elimination, [on,off]).
```

13.3.7 Built-In Predicates

This table lists the predicates made available by the CHR library. They are meant for advanced users, who want to tailor the CHR system towards their specific needs.

`current_handler(?Handler, ?Module)`
: Nondeterministically enumerates the defined handlers with the module they are defined in.

`current_constraint(?Handler, ?Constraint)`
: Nondeterministically enumerates the defined constraints in the form *Functor/Arity* and the handlers they are defined in.

`insert_constraint(+Constraint, -Id)`
: Inserts *Constraint* into the constraint store without executing any rules. The constraint will be woken and reconsidered when one of the variables in *Con-*

straint is touched. *Id* is unified with an internal object representing the constraint. This predicate only gets defined when a handler and constraints are declared (see Section 13.3.2 [CHR Declarations], page 140).

`insert_constraint(+Constraint, -Id, ?Term)`
: Inserts *Constraint* into the constraint store without executing any rules. The constraint will be woken and reconsidered when one of the variables in *Term* is touched. *Id* is unified with an internal object representing the constraint. This predicate only gets defined when a handler and constraints are declared (see Section 13.3.2 [CHR Declarations], page 140).

`find_constraint(?Pattern, -Id)`
: Nondeterministically enumerates constraints from the constraint store that match *Pattern*, i.e. which are instances of *Pattern*. *Id* is unified with an internal object representing the constraint.

`find_constraint(-Var, ?Pattern, -Id)`
: Nondeterministically enumerates constraints from the constraint store that delay on *Var* and match *Pattern*, i.e. which are instances of *Pattern*. The identifier *Id* can be used to refer to the constraint later, e.g. for removal.

`findall_constraints(?Pattern, ?List)`
: Unifies *List* with a list of `Constraint # Id` pairs from the constraint store that match *Pattern*.

`findall_constraints(-Var, ?Pattern, ?List)`
: Unifies *List* with a list of `Constraint # Id` pairs from the constraint store that delay on *Var* and match *Pattern*.

`remove_constraint(+Id)`
: Removes the constraint *Id*, obtained with one of the previous predicates, from the constraint store.

`unconstrained(?Var)`
: Succeeds if no CHR constraint delays on *Var*. Defined as:

```
unconstrained(X) :-
    find_constraint(X, _, _), !, fail.
unconstrained(_).
```

`notify_constrained(?Var)`
: Leads to the reconsideration of the constraints associated with *Var*. This mechanism allows solvers to communicate reductions on the set of possible values of variables prior to making bindings.

13.3.8 Consulting and Compiling Constraint Handlers

The CHR compilation process has been made as transparent as possible. The user deals with files containing CHR just as with files containing ordinary Prolog predicates. Thus CHR may be consulted, compiled with various compilation modes, and compiled to file.

13.3.9 Compiler-generated Predicates

Besides predicates for the defined constraints, the CHR compiler generates some support predicates in the module containing the handler. To avoid naming conflicts, the following predicates must not be defined or referred to by user code in the same module:

`verify_attributes/3`
`attribute_goal/2`
`attach_increment/2`
`'attach_F/A'/2`
: for every defined constraint F/A.

`'F/A_N_M_...'/Arity`
: for every defined constraint F/A. N,M is are integers, Arity > A.

For the prime number example that is:

```
attach_increment/2
attach_prime/1/2
attach_primes/1/2
attribute_goal/2
goal_expansion/3
prime/1
prime/1_1/2
prime/1_1_0/3
prime/1_2/2
primes/1
primes/1_1/2
verify_attributes/3
```

If an author of a handler wants to avoid naming conflicts with the code that uses the handler, it is easy to encapsulate the handler. The module declaration below puts the handler into module `primes`, which exports only selected predicates - the constraints in our example.

```
:- module(primes, [primes/1,prime/1]).

:- use_module(library(chr)).

handler eratosthenes.
constraints primes/1,prime/1.
...
```

13.3.10 Operator Declarations

This table lists the operators as used by the CHR library:

```
:- op(1200, xfx, @).
:- op(1190, xfx, pragma).
:- op(1180, xfx, [==>,<=>]).
:- op(1180, fy,  chr_spy).
:- op(1180, fy,  chr_nospy).
:- op(1150, fx,  handler).
:- op(1150, fx,  constraints).
:- op(1150, fx,  rules).
:- op(1100, xfx, '|').
:- op(1100, xfx, \ ).
:- op(1050, xfx, &).
:- op( 500, yfx, #).
```

13.3.11 Exceptions

The CHR runtime system reports instantiation and type errors for the predicates:

```
find_constraint/2
findall_constraints/3
insert_constraint/2
remove_constraint/1
notify_constrained/1
```

The only other CHR specific runtime error is:

`{CHR ERROR: registering <New>, module <Module> already hosts <Old>}`
: An attempt to load a second handler New into module <Module> already hosting handler <Old> was made.

The following exceptional conditions are detected by the CHR compiler:

`{CHR Compiler ERROR: syntax rule <N>: <Term>}`
: If the N-th <Term> in the file being loaded violates the CHR syntax (see Section 13.3.3 [CHR Syntax], page 140).

`{CHR Compiler ERROR: too many general heads in <Name>}`
: Unspecific heads in definitions like `C \ C <=> true` must not be combined with other heads in rule <Name>.

`{CHR Compiler ERROR: bad pragma <Pragma> in <Name>}`
: The pragma <Pragma> used in rule <Name> does not qualify. Currently this only happens if <Pragma> is unbound.

`{CHR Compiler ERROR: found head <F/A> in <Name>, expected one of: <F/A list>}`
: Rule <Name> has a head of given F/A which is not among the defined constraints.

`{CHR Compiler ERROR: head identifiers in <Name> are not unique variables}`
: The identifiers to refer to individual constraints (heads) via '#' in rule <Name> do not meet the indicated requirements.

`{CHR Compiler ERROR: no handler defined}`
: CHR specific language elements, declarations or rules for example, are used before a handler was defined. This error is usually reported a couple of times, i.e. as often as there are CHR forms in the file expecting the missing definition.

`{CHR Compiler ERROR: compilation failed}`
: Not your fault. Send us a bug report.

13.4 Debugging CHR Programs

Use `option(debug_compile,on)` preceding any rules in the file containing the handler to enable CHR debugging. The CHR debugging mechanism works by instrumenting the code generated by the CHR compiler. Basically, the CHR debugger works like the Prolog debugger. The main differences are: there are extra ports specific to CHR, and the CHR debugger provides no means for the user to change the flow of control, i.e. there are currently no *retry* and *fail* options available.

13.4.1 Control Flow Model

The entities reflected by the CHR debugger are constraints and rules. Constraints are treated like ordinary Prolog goals with the usual ports: `[call,exit,redo,fail]`. In addition, constraints may get inserted into or removed from the constraint store (ports: `insert,remove`), and stored constraints containing variables will be woken and reconsidered (port: `wake`) when variables are touched.

The execution of a constraint consists of trying to apply the rules mentioning the constraint in their heads. Two ports for rules reflect this process: At a `try` port the active constraint matches one of the heads of the rule, and matching constraints for the remaining heads of the rule, if any, have been found as well. The transition from a `try` port to an `apply` port takes place when the guard has been successfully evaluated, i.e. when the rule commits. At the `apply` port, the body of the rule is just about to be executed. The body is a Prolog goal transparent to the CHR debugger. If the rule body contains CHR constraints, the CHR debugger will track them again. If the rules were consulted, the Prolog debugger can be used to study the evaluations of the other predicates in the body.

13.4.2 CHR Debugging Predicates

The following predicates control the operation of the CHR debugger:

`chr_trace`
: Switches the CHR debugger on and ensures that the next time control enters a CHR port, a message will be produced and you will be asked to interact.

  At this point you have a number of options. See Section 13.4.5 [CHR Debugging Options], page 151. In particular, you can just type ⟨cr⟩ (Return) to *creep* (or single-step) into your program. You will notice that the CHR debugger stops at many ports. If this is not what you want, the predicate `chr_leash` gives full control over the ports at which you are prompted.

`chr_debug`
: Switches the CHR debugger on and ensures that the next time control enters a CHR port with a spy-point set, a message will be produced and you will be asked to interact.

`chr_nodebug`
: Switches the CHR debugger off. If there are any spy-points set then they will be kept.

`chr_notrace`
: Equivalent to `chr_nodebug`.

`chr_debugging`
: Prints onto the standard error stream information about the current CHR debugging state. This will show:

  1. Whether the CHR debugger is switched on.
  2. What spy-points have been set (see below).
  3. What mode of leashing is in force (see below).

`chr_leash(+`*Mode*`)`
: The leashing mode is set to *Mode*. It determines the CHR ports at which you are to be prompted when you *creep* through your program. At unleashed ports a tracing message is still output, but program execution does not stop to allow user interaction. Note that the ports of spy-points are always leashed (and cannot be unleashed). *Mode* is a list containing none, one or more of the following port names:

  `call` Prompt when a constraint is executed for the first time.

  `exit` Prompt when the constraint is successfully processed, i.e. the applicable rules have applied.

  `redo` Prompt at subsequent exits generated by non-deterministic rule bodies.

  `fail` Prompt when a constraint fails.

  `wake` Prompt when a constraint from the constraint store is woken and reconsidered because one of its variables has been touched.

  `try` Prompt just before the guard evaluation of a rule, after constraints matching the heads have been found.

  `apply` Prompt upon the application of a rule, after the successful guard evaluation, when the rule commits and fires, just before evaluating the body.

  `insert` Prompt when a constraint gets inserted into the constraint store, i.e. after all rules have been tried.

  `remove` Prompt when a constraint gets removed from the constraint store, e.g. when a simplification rule applies.

  The initial value of the CHR leashing mode is `[call,exit,fail,wake,apply]`. Predefined shortcuts are:

`chr_leash(none), chr_leash(off)`
To turn leashing off.

`chr_leash(all)`
To prompt at every port.

`chr_leash(default)`
Same as `chr_leash([call,exit,fail,wake,apply])`.

`chr_leash(call)`
No need to use a list if only a singular port is to be leashed.

13.4.3 CHR Spy-points

For CHR programs of any size, it is clearly impractical to creep through the entire program. *Spy-points* make it possible to stop the program upon an event of interest. Once there, one can set further spy-points in order to catch the control flow a bit further on, or one can start creeping.

Setting a spy-point on a constraint or a rule indicates that you wish to see all control flow through the various ports involved, except during skips. When control passes through any port with a spy-point set on it, a message is output and the user is asked to interact. Note that the current mode of leashing does not affect spy-points: user interaction is requested at *every* port.

Spy-points are set and removed by the following predicates, which are declared as prefix operators:

`chr_spy` *Spec*
Sets spy-points on constraints and rules given by *Spec*, which is is of the form:

_ (variable)
denoting all constraints and rules, or:

constraints Cs
where *Cs* is one of

_ (variable)
denoting all constraints

C,...,*C* denoting a list of constraints *C*

Name denoting all constraints with this functor, regardless of arity

Name/Arity
denoting the constraint of that name and arity

rules Rs where *Rs* is one of:

_ (variable)
denoting all rules

R,...,*R* denoting a list of rules *R*

Name where *Name* is the name of a rule in any handler.

already_in_store
: The name of a rule implicitly defined by the system when the option `already_in_store` is in effect.

already_in_heads
: The name of a rule implicitly defined by the system when the option `already_in_heads` or the corresponding pragmas are in effect.

Handler:Name
: where *Handler* is the name of a constraint handler and *Name* is the name of a rule in that handler

Examples:

```
| ?- chr_spy rules rule(3), transitivity, already_in_store.
| ?- chr_spy constraints prime/1.
```

If you set spy-points, the CHR debugger will be switched on.

`chr_nospy` *Spec*
: Removes spy-points on constraints and rules given by *Spec*, where *Spec* is of the form as described for `chr_spy` *Spec*. There is no `chr_nospyall/0`. To remove all CHR spy-points use `chr_nospy _`.

The options available when you arrive at a spy-point are described later. See Section 13.4.5 [CHR Debugging Options], page 151.

13.4.4 CHR Debugging Messages

All trace messages are output to the standard error stream. This allows you to trace programs while they are performing file I/O. The basic format is as follows:

```
S 3 1 try eratosthenes:absorb(10) @ prime(9)#<c4>, prime(10)#<c2> ?
```

S is a spy-point indicator. It is printed as ' ' if there is no spy-point, as '`r`', indicating that there is a spy-point on this rule, or as '`c`' if one of the involved constraints has a spy-point.

The first number indicates the current depth of the execution; i.e. the number of direct *ancestors* the currently active constraint has.

The second number indicates the head position of the currently active constraint at rule ports.

The next item tells you which port is currently traced.

A constraint or a matching rule are printed next. Constraints print as `Term#Id`, where *Id* is a unique identifier pointing into the constraint store. Rules are printed as `Handler:Name @`, followed by the constraints matching the heads.

The final '`?`' is the prompt indicating that you should type in one of the debug options (see Section 13.4.5 [CHR Debugging Options], page 151).

13.4.5 CHR Debugging Options

This section describes the options available when the system prompts you after printing out a debugging message. Most of them you know from the standard Prolog debugger. All

the options are one letter mnemonics, some of which can be optionally followed by a decimal integer. They are read from the standard input stream up to the end of the line (Return, ⟨cr⟩). Blanks will be ignored.

The only option which you really have to remember is 'h'. This provides help in the form of the following list of available options.

```
CHR debugging options:
   <cr>   creep            c      creep
    l     leap
    s     skip             s <i>  skip (ancestor i)
    g     ancestors
    &     constraints      & <i>  constraints (details)
    n     nodebug          =      debugging
    +     spy this
    -     nospy this       .      show rule
    <     reset printdepth < <n>  set printdepth
    a     abort            b      break
    ?     help             h      help
```

c
⟨cr⟩ *creep* causes the debugger to single-step to the very next port and print a message. Then if the port is leashed, the user is prompted for further interaction. Otherwise, it continues creeping. If leashing is off, creep is the same as *leap* (see below) except that a complete trace is printed on the standard error stream.

l *leap* causes the debugger to resume running your program, only stopping when a spy-point is reached (or when the program terminates). Leaping can thus be used to follow the execution at a higher level than exhaustive tracing.

s
s i *skip* over the entire execution of the constraint. That is, you will not see anything until control comes back to this constraint (at either the `exit` port or the `fail` port). This includes ports with spy-points set; they will be masked out during the skip. The command can be used with a numeric argument to skip the execution up to and including the ancestor indicated by the argument. Example:

```
           ...
           4   - exit    prime(8)#<c6> ? g
     Ancestors:
           1   1 apply   eratosthenes:rule(2) @ primes(10)#<c1>
           2   1 apply   eratosthenes:rule(2) @ primes(9)#<c3>
           3   1 apply   eratosthenes:rule(2) @ primes(8)#<c5>
           4   - call    prime(8)#<c6>

           4   - exit    prime(8)#<c6> ? s 2
           2   - exit    primes(9)#<c3> ?
```

g *print ancestors* provides you with a list of ancestors to the currently active constraint, i.e. all constraints not yet exited that led to the current constraint in the derivation sequence. The format is the same as with trace messages.

Constraints start with `call` entries in the stack. The subsequent application of a rule replaces the call entry in the stack with an `apply` entry. Later the constraint shows again as `redo` or `fail` entry. Example:

```
          0   - call    primes(10)#<c1> ?
          1   1 try     eratosthenes:rule(2) @ primes(10)#<c1> ? g

     Ancestors:
          1   - call    primes(10)#<c1>

          1   1 try     eratosthenes:rule(2) @ primes(10)#<c1> ?
          1   1 apply   eratosthenes:rule(2) @ primes(10)#<c1> ?
          1   - call    prime(10)#<c2> ?
          2   - insert  prime(10)#<c2>
          2   - exit    prime(10)#<c2> ? g

     Ancestors:
          1   1 apply   eratosthenes:rule(2) @ primes(10)#<c1>
          2   - call    prime(10)#<c2>
```

`&` *print constraints* prints a list of the constraints in the constraint store. With a numeric argument, details relevant primarily to CHR implementors are shown.

`n` *nodebug* switches the CHR debugger off.

`=` *debugging* outputs information concerning the status of the CHR debugger as via `chr_debugging/0`

`+` *spy this* sets a spy-point on the current constraint or rule.

`-` *nospy this* removes the spy-point from the current constraint or rule, if it exists.

`.` *show rule* prints the current rule instantiated by the matched constraints. Example:

```
     8 1 apply  era:absorb(8) @ prime(4)#<c14> \ prime(8)#<c6> ? .

     absorb(8) @
      prime(4)#<c14> \
        prime(8)#<c6> <=>

        8 mod 4=:=0
        |
        true.
```

`<`

`< n` While in the debugger, a *printdepth* is in effect for limiting the subterm nesting level when printing rules and constraints. The limit is initially 10. This command, without arguments, resets the limit to 10. With an argument of *n*, the limit is set to *n*. An argument *n* of 0 disables depth limit in the debugger.

`a` *abort* calls the built-in predicate `abort/0`.

`b` *break* calls the built-in predicate `break/0`, thus putting you at a recursive top-level. When you end the break (entering ^D) you will be re-prompted at the port at which you broke. The CHR debugger is temporarily switched off as you call the break and will be switched on again when you finish the break and go back to the old execution. Any changes to the CHR leashing or to spy-points during the break will remain in effect.

`?`
`h` *help* displays the table of options given above.

13.5 Programming Hints

This section gives you some programming hints for CHR. For maximum efficiency of your constraint handler, see also the previous subsections on declarations and options.

Constraint handling rules for a given constraint system can often be derived from its definition in formalisms such as inference rules, rewrite rules, sequents, formulas expressing axioms and theorems. CHR can also be found by first considering special cases of each constraint and then looking at interactions of pairs of constraints sharing a variable. Cases that do not occur in the application can be ignored.

It is important to find the right *granularity* of the constraints. Assume one wants to express that *n* variables are different from each other. It is more efficient to have a single constraint `all_different(List_of_n_Vars)` than *n*n* inequality constraints between each pair of different variables. However, the extreme case of having a single constraint modeling the whole constraint store will usually be inefficient.

Starting from an executable specification, the rules can then be refined and adapted to the specifics of the application. Efficiency can be improved by weakening the guards to perform simplification as early as needed and by strengthening the guards to do the *just right* amount of propagation. Propagation rules can be expensive, because no constraints are removed.

The more heads a rule has, the more expensive it is. *Rules with several heads* are more efficient, if the heads of the rule share a variable (which is usually the case). Then the search for a partner constraint has to consider less candidates. In the current implementation, constraints are indexed by their functors, so that the search is only performed among the constraints containing the shared variable. Moreover, two rules with identical (or sufficiently similar) heads can be merged into one rule so that the search for a partner constraint is only performed once instead of twice.

As *guards* are tried frequently, they should be simple *tests* not involving side-effects. Head matching is more efficient than explicitly checking equalities in the ask-part of the guard. In the tell part of a guard, it should be made sure that variables from the head are never touched (e.g. by using `nonvar` or `ground` if necessary). For efficiency and clarity reasons, one should also avoid using constraints in guards. Besides conjunctions, disjunctions are allowed in the guard, but they should be used with care. The use of other control built-in predicates in the guard is discouraged. Negation and if-then-else in the ask part of a guard can give wrong results, since e.g. failure of the negated goal may be due to touching its variables.

Several handlers can be used simultaneously if they do not share constraints with the same name. The implementation will not work correctly if the same constraint is defined in rules of different handlers that have been compiled separately. In such a case, the handlers must be merged *by hand*. This means that the source code has to be edited so that the rules for the shared constraint are together (in one module). Changes may be necessary (like strengthening guards) to avoid divergence or loops in the computation.

13.6 Constraint Handlers

The CHR library comes with plenty of constraint handlers written in CHR. The most recent versions of these are maintained at:

`http://www.pst.informatik.uni-muenchen.de/~fruehwir/chr-solver.html`

'`arc.pl`' classical arc-consistency over finite domains

'`bool.pl`' simple Boolean constraints

'`cft.pl`' feature term constraints according to the CFT theory

'`domain.pl`'
finite domains over arbitrary ground terms and interval domains over integers and reals, but without arithmetic functions

'`gcd.pl`' elegant two-liner for the greatest common divisor

'`interval.pl`'
straightforward interval domains over integers and reals, with arithmetic functions

'`kl-one.pl`'
terminological reasoning similar to KL-ONE or feature trees

'`leq.pl`' standard introductory CHR example handler for less-than-or-equal

'`list.pl`' equality constraints over concatenations of lists (or strings)

'`listdom.pl`'
a straightforward finite enumeration list domains over integers, similar to '`interval.pl`'

'`math-elim.pl`'
solves linear polynomial equations and inequations using variable elimination, several variations possible

'`math-fougau.pl`'
solves linear polynomial equations and inequations by combining variable elimination for equations with Fourier's algorithm for inequations, several variations possible

'`math-fourier.pl`'
a straightforward Fouriers algorithm to solve polynomial inequations over the real or rational numbers

'`math-gauss.pl`'
: a straightforward, elegant implementation of variable elimination for equations in one rule

'`minmax.pl`'
: simple less-than and less-than-or-equal ordering constraints together with minimum and maximum constraints

'`modelgenerator.pl`'
: example of how to use CHR for model generation in theorem proving

'`monkey.pl`'
: classical monkey and banana problem, illustrates how CHR can be used as a fairly efficient production rule system

'`osf.pl`' constraints over order sorted feature terms according to the OSF theory

'`oztype.pl`'
: rational trees with disequality and OZ type constraint with intersection

'`pathc.pl`'
: the most simple example of a handler for path consistency - two rules

'`primes.pl`'
: elegant implementations of the sieve of Eratosthenes reminiscent of the chemical abstract machine model, also illustrates use of CHR as a general purpose concurrent constraint language

'`scheduling.pl`'
: simple classical constraint logic programming scheduling example on building a house

'`tarski.pl`'
: most of Tarski's axiomatization of geometry as constraint system

'`term.pl`' Prolog term manipulation built-in predicates `functor/3, arg/3, =../2` as constraints

'`time-pc.pl`'
: grand generic handler for path-consistency over arbitrary constraints, load via '`time.pl`' to get a powerful solver for temporal constraints based on Meiri's unifying framework. '`time-rnd.pl`' contains a generator for random test problems.

'`time-point.pl`'
: quantitative temporal constraints over time points using path-consistency

'`tree.pl`' equality and disequality over finite and infinite trees (terms)

'`type.pl`' equalities and type constraints over finite and infinite trees (terms)

You can consult or compile a constraint handler from the CHR library using e.g.:

```
?- [library('chr/examples/gcd')].
?- compile(library('chr/examples/gcd')).
```

If you want to learn more about the handlers, look at their documented source code.

In addition, there are files with example queries for some handlers, their file name starts with '`examples-`' and the file extension indicates the handler, e.g. '`.bool`':

```
examples-adder.bool
examples-benchmark.math
examples-deussen.bool
examples-diaz.bool
examples-fourier.math
examples-holzbaur.math
examples-lim1.math
examples-lim2.math
examples-lim3.math
examples-puzzle.bool
examples-queens.bool
examples-queens.domain
examples-stuckey.math
examples-thom.math
```

13.7 Backward Compatibility

In this section, we discuss backward compatibility with the CHR library of Eclipse Prolog.

1. The restriction on at most two heads in a rule has been abandoned. A rule can have as many heads as you like. Note however, that searching for partner constraints can be expensive.
2. By default, rules are compiled in textual order. This gives the programmer more control over the constraint handling process. In the Eclipse library of CHR, the compiler was optimizing the order of rules. Therefore, when porting a handler, rules may have to be reordered. A good heuristic is to prefer simplification to simpagation and propagation and to prefer rules with single heads to rules with several heads. Instead of manually rearranging an old handler one may also use the following combination of options to get the corresponding effect:

   ```
   option(rule_ordering,heuristic).
   option(revive_scheme,old).
   ```

3. For backward compatibility, the `already_in_store`, `already_in_head` and `guard_bindings` options are still around, but there are CHR syntax extensions: Section 13.3.3 [CHR Syntax], page 140 and pragmas Section 13.3.5 [CHR Pragmas], page 142 offering better grained control.
4. The Eclipse library of CHR provided automatic built-in labeling through the `label_with` declaration. Since it was not widely used and can be easily simulated, built-in labeling was dropped. The same effect can be achieved by replacing the declaration `label_with Constraint if Guard` by the simplification rule `chr_labeling, Constraint <=> Guard | Constraint', chr_labeling` and by renaming the head in each clause `Constraint :- Body` into `Constraint' :- Body` where `Constraint'` is a new predicate. Efficiency can be improved by declaring `Constraint` to be passive: `chr_labeling, Constraint#Id <=> Guard | Constraint', chr_labeling pragma`

`passive(Id)`. This translation will not work if `option(already_in_heads,on)`. In that case use e.g. `chr_labeling(_), Constraint <=> Guard | Constraint', chr_labeling(_)` to make the new call to `chr_labeling` differ from the head occurrence.

5. The set of built-in predicates for advanced CHR users is now larger and better designed. Also the debugger has been improved. The Opium debugging environment is not available in SICStus Prolog.

14 Logtalk

The Logtalk object-oriented extension is available once included with the `use_module(library(logtalk))` command. Note that, although we load Logtalk using the `use_module/1` built-in predicate, the system is not packaged as a module not does it use modules in its implementation.

Logtalk documentation is included in the Logtalk directory. For the latest news, please see the URL `http://www.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 succesfully, 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 omited, 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 omited, 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`).

`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_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_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 `current_thread/2` using the result of `thread_self/1` as thread-identifier.

`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.

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.

`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 collistion-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.

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(+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 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`.

```
<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`.

`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_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.

`thread_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 anynymous 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.

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). 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 synchronysation 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 *MUT*ual *EX*clusive 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:

```
:- 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`, wich 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 te 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

An initial cut for an implementation of tabling in the style of XSB-Prolog is now available. Tabling was implemented by Ricardo Rocha. To experiment with tabling use `-DTABLING` to `YAP_EXTRAS` in the system's `Makefile`.

You can use the directive `table` to force calls for the argument predicate to be tabled. Tabling information is stored in a trie, as for XSB-Prolog.

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 builtins 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 =\= 0` prompt on fail
- if `N/\ 2 =\= 0` prompt on redo
- if `N/\ 4 =\= 0` prompt on exit
- if `N/\ 8 =\= 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)`.

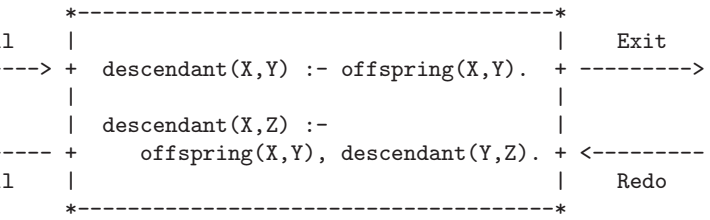
`spy_write(+`*Stream*`,Term)`
If defined by the user, this predicate will be used to print goals by the debugger instead of `write/2`.

`trace` Switches on the debugger and starts tracing.

20.2 Interacting with the debugger

Debugging with YAP is similar to debugging with C-Prolog. Both systems include a procedural debugger, based in the 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:



`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.

`Exit` This port is activated if the procedure succeeds. Control will now leave the procedure and return to its ancestor.

`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.

`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 usually 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 like:

```
* (1)  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 two characters are used to point out special states of the debugger. If the first character is a `*`, execution is at a spy-point. If the second 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/1`.

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 - creep`
the same as c

`l - leap` YAP will continue execution until a port of a spied predicate is found;

`k - quasi-leap`
similar to leap but faster since the computation history is not kept; useful when leap becomes too slow.

`s - skip` YAP will continue execution without showing any messages until returning to the current activation. Spy-points will be ignored in this mode. This command is meaningless, and therefore illegal, in the fail and exit ports.

`t - fast-skip`
similar to skip but faster since the computation history is not kept; useful when skip becomes too slow.

`q - quasi-leap`
YAP will continue execution until a port of a spied predicate is found or until returning to the current activation.

`f - fail` forces YAP to fail the goal proceeding directly to the fail port. The command is not available in the fail port.

`r - retry` after this command, YAP will retry the present goal, and so go back to the call port. Note that any side effects of the goal will not be undone. This command is not available at the call port.

`a - abort` execution will be aborted, and the interpreter will return to the top-level.

`n - nodebug`
stop debugging but continue execution. The command will clear all active spy-points, leave debugging mode and continue execution.

`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 6.6.7 [I/O Control], page 52).

`< - full term`
resets to the default of ten the debugger's maximum write depth. For more information about `write_depth/2` (see Section 6.6.7 [I/O Control], page 52).

The debugging information, when fast-skip `quasi-leap` is used, will be lost.

21 Indexing

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 a clauses from all the clauses forming the procedure.

As an example, the two clauses for concatenate:

```
concatenate([],L,L).
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 programmers 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(aristhoteles,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:

```
type(n(N),T) :- type_of_noun(N,T).
type(v(V),T) :- type_of_verb(V,T).

type_of_noun(mary,person).
type_of_noun(john,person).
type_of_noun(chair,object).

type_of_verb(eat,active).
type_of_verb(rest,passive).
```

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.

```
#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 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.

Yap4.3.3 now supports loading WIN/NT DLLs. Currently you must compile YAP under cygwin to create a library yap.dll first. You can then use this dll to create your own dlls. Have a look at the code in library/regex to see how to create a dll under the cygwin/mingw32 environment.

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

```
YAP_Bool YAP_IsVarTerm(YAP_Term t)
```

returns true iff its argument is an uninstantiated variable. Conversely the primitive

```
YAP_Bool 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

```
Term  YAP_MkVarTerm()
```

The following primitives can be used to discriminate among the different types of non-variable terms:

```
YAP_Bool YAP_IsIntTerm(YAP_Term t)
YAP_Bool YAP_IsFloatTerm(YAP_Term t)
YAP_Bool YAP_IsDbRefTerm(YAP_Term t)
YAP_Bool YAP_IsAtomTerm(YAP_Term t)
YAP_Bool YAP_IsPairTerm(YAP_Term t)
YAP_Bool 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.

```
YAP_Term YAP_MkIntTerm(YAP_Int  i)
YAP_Int  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

```
YAP_Term YAP_MkFloatTerm(YAP_flt double)
YAP_flt  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.

```
YAP_Term YAP_MkBigNumTerm(YAP_Int  i)
YAP_Int  YAP_BigNumOfTerm(YAP_Term t)
```

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 the `MP_INT *` type.

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

```
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

```
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.

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 of 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_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 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

```
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 or to a list of character atoms

```
YAP_Term YAP_BufferToString(char *buf)
YAP_Term YAP_BufferToAtomList(char *buf)
```

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.

```
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:

```
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:

```
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()`.

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 From C back to Prolog

Newer versions of YAP allow for calling the Prolog interpreter from `C`. One must first construct a goal `G`, and then it is sufficient to perform:

```
YAP_Bool      YapCallProlog(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.

22.7 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

```
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 two C functions. The first one which is called when the predicate is first activated, and the second one to be called on backtracking to provide (possibly) other solutions. 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

```
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.

```
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:

```
static int start_n100()
{
      YAP_Term t = 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

```
static int continue_n100()
{
      int n;
      Term t;
      Term sol = 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 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 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 YAP_unify to bind the argument of the call to the value saved in n100_state->next_solution.

Note also that the only correct way to signal failure in a backtrackable predicate is to use the 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

```
void YAP_UserBackCPredicate(char *name,
          int *init(), int *cont(),
          unsigned long int arity, unsigned int sizeof);
```

where *name* is a string with the name of the predicate, *init* and *cont* are the C functions used to start and continue the execution of the predicate, *arity* is the predicate arity, and *sizeof* is the size of the data to be preserved in the stack.

22.8 Loading Object Files

The primitive predicate

```
load_foreign_files(Files,Libs,InitRoutine)
```

should be used, from inside YAP, to load object files produced by the C compiler. The argument *ObjectFiles* should be a list of atoms specifying the object files to load, *Libs* is a list (possibly empty) of libraries to be passed to the unix loader (`ld`) and 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 *ObjectFiles* in the current directory first. If it cannot find them it will search for the files using the environment variable `YAPLIBDIR`, if defined, or in the default library.

In a.out systems YAP by default only reserves a fixed amount of memory for object code (64 Kbytes in the current version). Should this size prove inadequate the flag `-c n` can be passed to YAP (in the command line invoking YAP) to force the allocation of `n` Kbytes.

22.9 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.

22.10 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:

```
{
   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 dereferenciate 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`:

```
#include <stdio.h>
#include "Yap/YapInterface.h"


int
main(int argc, char *argv[]) {
  if (YAP_FastInit("saved_state") == YAP_BOOT_FROM_SAVED_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:

```
cc -o exem -I../Yap4.3.0 test.c -lYap -lreadline -lm
```

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(`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(`char *` *error_description*) Generate an YAP System Error with description given by the string *error_description*.
- `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.
- `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(`char *` *SavedState*, `int` *HeapSize*, `int` *StackSize*, `int` *TrailSize*, `int` *NumberofWorkers*, `int` *SchedulerLoop*, `int` *DelayedReleaseLoad*, `int` *argc*, `char **` *argv*) Initialize YAP. In the future the arguments as a single `C` structure.

  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.
- `void` YAP_PutValue(`Atom` *at*, `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.
- `Term` YAP_Read(`int (*)(void)` *GetC*) Parse a Term using the function *GetC* to input characters.
- `Term` YAP_RunGoal(`Term` *Goal*) Execute query *Goal* and return 1 if the query succeeds, and 0 otherwise. The predicate returns 0 if failure, otherwise it will return *Term*. Note that *Term* may change due to garbage collection, so you should use something like:

```
t = YAP_RunGoal(t);
if (t == 0) return FALSE;
```

If the execution fails, garbage collection might still have changed the term, so you should not use the input argument again.

- `int` YAP_RestartGoal(`void`) Look for the next solution to the current query by forcing YAP to backtrack.
- `int` YAP_Reset(`void`) Reset execution environment (similar to the `abort/0` builtin). This is useful when you want to start a new query before asking all solutions to the previous query.
- `void` YAP_Write(`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(`Term` *t*, `char *` *buff*, `unsigned int` *size*, `int` *flags*) Write a 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:

>

@

[

\

A

B

C

D

E

F

G

H

I

K

L

N

O

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.

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

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

```
:- 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 builtins `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): `call_cleanup/1`, `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 introduced in future releases of SICStus Prolog.
- YAP allows asserting and abolishing static code during execution through the `assert_static/1` and `abolish/1` builtins. This is not allowed in Quintus Prolog or SICStus Prolog.
- YAP implements rational trees and co-routining but they are not included by default in the system. You must enable these extensions when compiling the system.
- YAP does not currently implement constraints.
- 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 backtracking. 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:

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

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:

24.2.4 Yap predicates not available in SICStus Prolog

These are YAP built-ins not available in SICStus Prolog.

N

O

P

R

S

T

U

V

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 204). 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 6.3 [Testing Terms], page 32), and `number_chars/2`, (see Section 6.3 [Testing Terms], page 32), 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:

  ```
  :- 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, 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, and a postfix operator, xf, 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(1000,xfy,',').
:-op(999,xfy,'.').
:-op(900,fy,['\+', not]).
:-op(900,fx,[nospy, spy]).
:-op(700,xfx,[@>=,@=<,@<,@>,<,=,>,=:=,=\=,\==,>=,=<,==,\=,=..,is]).
:-op(500,yfx,['\/','/\','+','-']).
```

```
:-op(500,fx,['+','-']).
:-op(400,yfx,['<<','>>','//','*','/']).
:-op(300,xfx,mod).
:-op(200,xfy,['^','**']).
:-op(50,xfx,same).
```

Predicate Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Concept Index