Abstract

Qu-Prolog is an extension of Prolog designed primarily as an implementation and tactic language for interactive theorem provers, particularly those that carry out schematic proofs. Qu-Prolog has built-in support for the kinds of data structures typically encountered in theorem proving activities such as object variables, substitutions and quantified terms. Qu-Prolog also supports inter-process communication (IPC) that allows messages to be passed between Qu-Prolog threads, whether they are executing on the same Qu-Prolog process or on different processes or even on different machines.

This document is the reference manual for Qu-Prolog version 10.3
## Contents

1 Getting Started  
   1.1 Environment Variables  
   1.2 Data Areas  
   1.3 Running the Interpreter  
   1.4 Running the Compiler  
   1.5 Online Manuals  
   1.6 User Guide  
   1.7 Windows Users  

2 Syntax  
   2.1 Constants  
      2.1.1 Atoms  
      2.1.2 Numbers  
   2.2 Variables  
      2.2.1 Meta Variables  
      2.2.2 Object Variables  
   2.3 Compound Terms  
      2.3.1 Functional Notation  
      2.3.2 Expressions  
      2.3.3 Lists  
      2.3.4 Strings  
      2.3.5 {}-Lists  
   2.4 Quantified Terms  
   2.5 Substitutions  
   2.6 Programs  

3 Built-in Predicates  
   3.1 Introduction  
   3.2 Control  
   3.3 Input / Output  

4  

10  

15  

16  

17  

22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1</td>
<td>File and Stream Handling</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Term Input/Output</td>
<td>31</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Character Input/Output</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Terms</td>
<td>49</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison of Terms</td>
<td>49</td>
</tr>
<tr>
<td>3.6</td>
<td>Testing of Terms</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>Term Manipulation</td>
<td>60</td>
</tr>
<tr>
<td>3.8</td>
<td>List Processing</td>
<td>72</td>
</tr>
<tr>
<td>3.9</td>
<td>All Solutions</td>
<td>80</td>
</tr>
<tr>
<td>3.10</td>
<td>Arithmetic</td>
<td>82</td>
</tr>
<tr>
<td>3.11</td>
<td>Term Expansion</td>
<td>86</td>
</tr>
<tr>
<td>3.12</td>
<td>Database</td>
<td>92</td>
</tr>
<tr>
<td>3.12.1</td>
<td>Dynamic Database</td>
<td>92</td>
</tr>
<tr>
<td>3.12.2</td>
<td>Record Database</td>
<td>97</td>
</tr>
<tr>
<td>3.12.3</td>
<td>Global State Database</td>
<td>98</td>
</tr>
<tr>
<td>3.12.4</td>
<td>Hash Table</td>
<td>100</td>
</tr>
<tr>
<td>3.13</td>
<td>Loading Programs</td>
<td>101</td>
</tr>
<tr>
<td>3.14</td>
<td>Debugging</td>
<td>103</td>
</tr>
<tr>
<td>3.15</td>
<td>Foreign Language Interface</td>
<td>109</td>
</tr>
<tr>
<td>3.15.1</td>
<td>High Level Foreign Language Interface</td>
<td>109</td>
</tr>
<tr>
<td>3.15.2</td>
<td>Low Level Foreign Language Interface</td>
<td>113</td>
</tr>
<tr>
<td>3.16</td>
<td>Macros</td>
<td>114</td>
</tr>
<tr>
<td>3.17</td>
<td>Higher-Order Predicates</td>
<td>116</td>
</tr>
<tr>
<td>3.18</td>
<td>Implicit Parameters</td>
<td>120</td>
</tr>
<tr>
<td>3.19</td>
<td>Unification Control</td>
<td>123</td>
</tr>
<tr>
<td>3.20</td>
<td>Delayed Problem Handling</td>
<td>126</td>
</tr>
<tr>
<td>3.21</td>
<td>Program State</td>
<td>129</td>
</tr>
<tr>
<td>3.22</td>
<td>Exception Handling</td>
<td>133</td>
</tr>
<tr>
<td>3.23</td>
<td>Multiple Threads</td>
<td>139</td>
</tr>
<tr>
<td>3.24</td>
<td>Interprocess Communication</td>
<td>144</td>
</tr>
<tr>
<td>3.25</td>
<td>Graphical User Interface</td>
<td>150</td>
</tr>
</tbody>
</table>
1 Getting Started

This section describes how to set up the required environment variables and briefly describes how to run the interpreter and compiler.

1.1 Environment Variables

The root directory of the Qu-Prolog tree contains the files `PROFILE_CMDS` and `LOGIN_CMDS` that can be used to define the required environment variables.

1.2 Data Areas

Qu-Prolog contains several data areas that store execution state information. The sizes of these areas can be set at runtime or when new threads are created.

The data areas are described below.

The global stack (sometimes called the heap).

The global stack stores the Qu-Prolog terms build during forward execution.

The scratch pad.

The scratch pad is used for storing terms temporarily during the execution of `findall`, `setof` and `bagof` and for simplifying terms involving substitutions.

The local stack (environment stack).

The local stack contains all the environments for the current execution state.

The choice point stack.

The choice point stack contains all the choice points for the current execution state.

The binding trail.

This trail is used to determine which variables should be reset to unbound on backtracking.

The other trail.

Some Qu-Prolog data structures change values during computation and this trail is used to reset these data structures to their old values on backtracking. Such data structures include the delay list associated with a variable, the distinctness information associated with an object variable, the names of variables, and implicit parameters. It is also used to manage variable tags, reference counts for the dynamic database and `call_cleanup`. 


The code area.

The code area is used to store the static (compiled) code. It includes all the Qu-Prolog library code.

The string table.

The string table stores all the strings used as the names of atoms.

The name table.

The name table is a hash table used to associate variable names with variables.

The implicit parameter table.

The implicit parameter table is a hash table that stores (pointers to) the current value of the implicit parameters.

The atom table.

The atom table is a hash table that stores information about atoms.

The predicate table.

The predicate table is a hash table that associates predicates (name and arity) with the code for the predicate.

1.3 Running the Interpreter

 qp is the name of the Qu-Prolog interpreter. From a Unix shell, Qu-Prolog is started by typing:

```
qp
```

When the interpreter is ready to accept a query, it will prompt you with

```
| ?-
```

When the interpreter displays an answer it is accompanied with any delayed problems (constraints) relevant to the query. After the interpreter displays an answer to a query it expects input from the user. If the user enters a semicolon then the interpreter will attempt to find another solution to the query. If the user enters a RETURN then the interpreter will prompt for a new query. If the user enters a comma then the interpreter will enter a new level where the user can extend the current query with more goals. Any variables in the original query and in the displayed answer to that query may be referenced in the extended query. The user may return to the previous level by entering a CONTROL-D at the prompt. The interpreter is able to maintain references to variables by using the variants of read and write that remember and generate variable names.
The available switches for the interpreter (and any Qu-Prolog runtime system) are as follows.

-B size

Set the binding trail size to \texttt{size} K words. The default size is 32K.

-O size

Set the other trail size to \texttt{size} K words. The default size is 32K.

-i size

Set the implicit parameter table size to \texttt{size} entries. The system makes the size of the table the next power of two bigger than twice the supplied size. The default size is 10000.

-b size

Set the recode database to \texttt{size} K words. The default size is 64K.

-C size

Set the choice point stack to \texttt{size} K words. The default size is 64K.

-e size

Set the environment stack to \texttt{size} K words. The default size is 64K.

-h size

Set the heap (global stack) to \texttt{size} K words. The default size is 400K.

-H size

Set the scratch pad to \texttt{size} K words. The default size is 10K.

-n size

Set the name table to \texttt{size} entries. The system makes the size of the table the next power of two bigger than twice the supplied size. The default size is 10000.

-z size

Set the thread table to \texttt{size} entries. The default size is 100. This switch determines the maximum number of threads that can be running at any time.

-N server-name

Set the machine (IP address) on which the Pedro server is running to \texttt{server-name}. The default is the current machine (localhost).

-P server-port
Set the port on which the Pedro server is listening to `server-port`. The default is 4550.

`-A process-symbol`

Set the name of this process to `process-symbol`.

`-l initialization-file`

Consult `initialization-file` before the interpreter starts.

`-g initial-goal`

Execute `initial-goal` before the interpreter starts but after initialization file is loaded (if any).

An online manual `qp(1)` is available to explain the options available to the interpreter.

1.4 Running the Compiler

As well as running programs in interpreted mode, Qu-Prolog programs can be compiled for faster execution.

Declarations appearing in the source code of the form

`?- Decl.`

or

`:- Decl.`

are executed by the compiler and are also compiled for execution at load time. The exceptions to this are `index/3` and `compile_time_only/1` declarations which are executed by the compiler only.

`qc` is an interface to the Qu-Prolog compilation system. The system consists of a preprocessor, a term expander, a compiler, an assembler, and a linker. `qc` processes the supplied options and calls each component with the appropriate arguments in the sequence given above.

A common usage of `qc` is where the user supplies a Qu-Prolog source program. `qc` compiles the program and generates an executable. The executable is stored in two files. `exec_file` (e.g. `a.out`) contains the basic information about the executable and `exec_file.qx` (e.g. `a.out.qx`) has the essential data about the program. To run the executable, the user types in `exec_file`.

`qc` accepts several types of filename arguments.

Files ending with `.ql` are Qu-Prolog source programs.

Files ending with `.qi` indicates that the file has been preprocessed.

Files ending with `.qg` are taken to contain clauses after term expansion.
Files ending with .qs are Qu-Prolog assembly programs.

Object files have the suffix .qo.

The compiler also accepts byte-encoded files produced by the encoded write of Qu-Prolog. Encoded files must have .qle or .qge extensions and are encoded equivalents of .ql and .qg files.

The available switches for the compiler are as follows.

-D macro

Define macro as 1 (one). This is the same as if a #define macro 1 line appeared at the beginning of the .ql file being compiled.

-E

Stop after running the preprocessor. The output is placed in the corresponding file suffixed .qi.

-G

Stop after running the term expander. The output is placed in the corresponding file suffixed .qg.

-R file

Supply the term expansion rules (in file) to the term expander.

-S

Stop after running the compiler. The output is placed in the corresponding file suffixed .qs.

-c

Stop after running the assembler. The default output file is the corresponding file suffixed .qo.

-o exec_file

Name the object file if the -c switch is also supplied. Otherwise, name the executable. If exec_file is not supplied the default name for the executable is a.out.

-r

Add rlwrap to the runtime system to provide history and command line editing. This is used for the interpreter. The runtime system (shell script) may need to be edited if rlwrap is to behave in a different way.

The qp switches are used to alter the size of different data areas for the compiler.
The \texttt{ql} switches (see below) fix the size of some of the data areas in the executable being generated.

The Qu-Prolog linker (\texttt{ql}) links .\texttt{qo} files to produce an executable.

The available switches for the linker are as follows.

\begin{itemize}
\item[-a \texttt{size}]
\begin{itemize}
\item Set the size of the atom table to \texttt{size} entries. The system makes the size of the table the next power of two bigger than twice the supplied size.
\item The default size is 10000.
\end{itemize}

\item[-d \texttt{size}]
\begin{itemize}
\item Set the code area to \texttt{size} K bytes. The default size is 400K bytes.
\end{itemize}

\item[-p \texttt{size}]
\begin{itemize}
\item Set the size of the predicate table to \texttt{size} entries. The system makes the size of the table the next power of two bigger than twice the supplied size.
\item The default size is 10000.
\end{itemize}

\item[-s \texttt{size}]
\begin{itemize}
\item Set the string table to \texttt{size} K bytes. The default size is 64K bytes.
\end{itemize}
\end{itemize}

An online manuals \texttt{qc(1)} and \texttt{ql(1)} is available to explain the options available to the compiler and linker.

1.5 Online Manuals

Included with the Qu-Prolog release is a \texttt{HTML} version of this manual. To access this version open the file \$\texttt{QPHOME/doc/manual/MAIN.html} in a browser.

1.6 User Guide

The system is supplied with a User Guide (Technical Report 00-20) that introduces the basic concepts of Qu-Prolog and presents some example programs and sessions with the interpreter.

Example programs are supplied in the \texttt{examples} directory.

1.7 Windows Users

The Windows installer for Qu-Prolog updates the PATH variable to include the Qu-Prolog executables. All Qu-Prolog programs should be run from a Command Prompt window. In the manual \texttt{Control-D} is used for end-of-file. For Windows users, end-of-file is \texttt{Control-Z} followed by a Enter. \texttt{\textit{rip}}. If the foreign function interface is required then the mingw compiler needs to be installed to manage the compilation and linking of C code.
2 Syntax

This section describes the concrete syntax of Qu-Prolog 10.3

2.1 Constants

2.1.1 Atoms

There are four syntactic forms for atoms.

1. A lower case letter followed by any sequence consisting of "_" and alphanumeric characters.
   For example:
   true
   semester_1

2. Any combination of the following set of graphic characters.
   |-/++<=#\$\^&':?.
   For example:
   @<=
   ?=
   Note: This is a non-standard use of '|' which is usually reserved exclusively for list construction. A consequence is that extra spaces or brackets may be needed when constructing lists whose elements contain graphic characters.
   For example:
   [+ |X]

3. Any sequence of characters enclosed by "'" (single quote). Single quote can be included in the sequence by writing the quote twice. "\" indicates an escape sequence, where the escape characters are case insensitive. The possible escape characters are:
newline  Meaning: Continuation.
-        Meaning: Same as d.
^character Meaning: Control character.
dd       Meaning: A two digit octal number.
a        Meaning: Alarm (ASCII = 7).
b        Meaning: Backspace (ASCII = 8).
c        Meaning: Continuation.
d        Meaning: Delete (ASCII = 127).
e        Meaning: Escape (ASCII = 27).
f        Meaning: Formfeed (ASCII = 12).
n        Meaning: Newline (ASCII = 10).
odd      Meaning: A two digits octal number.
r        Meaning: Return (ASCII = 13)
s        Meaning: Space (ASCII = 32).
t        Meaning: Horizontal tab (ASCII = 9).
v        Meaning: Vertical tab (ASCII = 11).
xdx      Meaning: A two digit hexadecimal number.

Here are a few examples of quoted atoms.
'hi!'     
'they''re' 
'\n'

4. One of the following.
   !
   ;
   [
   ]

2.1.2 Numbers

The available range of integers is -(2^31-1) to 2^31-1 on a 32 bit machine and -(2^63-1) to 2^63-1 on a 64 bit machine. Integers can be represented in any of the following ways.

1. Any sequence of numeric characters. This method denotes the number in decimal, or base 10.
   For example:
   123

2. Base^Number, where Base ranges from 2 to 36 and Number can have any sequence of alphanumeric characters. Both upper and lower case alphabetic characters in Number are used to represent the appropriate digit when Base is greater than 10.
   For example, integer value 10 can be written as:
   2’1010
3. Binary numbers can also be represented in the form `0b` followed by binary digits. Similarly octal and hexadecimal numbers can be represented by `0o` or `0x` followed by digits.
   For example:
   - `0b1011`
   - `0o3170`
   - `0x3afd`

4. `0`'Character` gives the character code of `Character`.
   For example,
   - `0`'A`
   gives the ASCII character code 65.

Double precision floating point numbers are also available and are represented using either a decimal point or scientific notation.

### 2.2 Variables

#### 2.2.1 Meta Variables

Meta variables are available in three syntactic forms.

1. An upper case letter followed by any sequence consisting of "_" and alphanumeric characters.
   For example:
   - `VarList`
   - `Term1`

2. "_", followed by an upper case letter, and then any sequence consisting of "_" and alphanumeric characters.
   For example:
   - `_Dictionary`
   - `_X_1`

3. "_" alone denotes an anonymous variable.

#### 2.2.2 Object Variables

Object variable names adhere to the following EBNF grammar:

```plaintext
obvar-name  ---> ['!'|'][_''] obvar-prefix obvar-suffix
obvar-prefix  ---> (lower-case-letter)+
obvar-suffix  ---> (letter | digit | '_')*
```
Notes:

1. When the '!' is omitted, the object-variable-prefix must have been previously declared with the predicates obvar_prefix/[1,2].

2. When the '_' is used, the object variable is said to be anonymous.

Examples (where x is predeclared as an object-variable-prefix):

\begin{verbatim}
x
x0
!_y (anonymous)
!y_0_1
\end{verbatim}

2.3 Compound Terms

2.3.1 Functional Notation

The compound terms are represented in this notation. A compound term is composed of a functor and a sequence of one or more arguments, which are enclosed in a pair of parenthesis. The functor and each of the arguments can be any term. For example:

\begin{verbatim}
sibling(jack, jill)
sort(qsort)(InList, OutList)
Functor(X, Y, Z)
\end{verbatim}

In the first example, sibling is the functor and the arity, the number of arguments, of the term is 2. sort(qsort) is the functor of the second example and the functor itself is a compound term with sort as the functor.

A compound term has at least one argument.

2.3.2 Expressions

If the functor of a compound term is declared as an operator by op/[3,4], terms may be written in the style of an expression. The expression is parsed according to the precedence and associativity of the operators. For example, + and * are infix operators while \(+\) is a prefix operator.

\begin{verbatim}
Number + 2 * 3
\end{verbatim}

2.3.3 Lists

Lists are a special kind of compound term. Lists have "." as the functor and two arguments. A special list notation is provided where the elements of a list are enclosed by a pair of " [ " ] " (square brackets). The elements are separated
from each other by a comma. The tail of the list, which is a term, not a sequence of terms, can be separated from the rest of the list by a ”|”.

For example, the following represent the same list.

[apple, orange, banana]
[apple, orange|[banana]]
[apple|[orange|[banana]]]

The atom [] represents the empty list.

2.3.4 Strings

Any sequence of characters enclosed by ”"" is considered as a strings. Strings are semantically the same as lists of ASCII codes but are stored more efficiently. Consequently the empty string ("") is parsed as the empty list ([]) and strings can unify with lists.

Example:

?- X = "hello", X = [H|T].
X = "hello"
H = 104
?- X = "".
X = []

2.3.5 {}-Lists

A special syntactic form recognized by the Qu-Prolog parser is that of {}-lists. The syntax for a {}-list is a ‘{’ followed by a collection of terms each terminated by a full stop and white spaces followed by ‘}’. The full stop and white space after that last term is optional. A {}-list is represented in Qu-Prolog as a compound term whose functor is {} and whose only argument is a ”comma pair” representing the elements of the {}-list.

For example, the following are {}-lists together with their internal representation.

{a. b. } {}{(a , b))
{a. b. c} {}{(a , (b , c))}
{a} {}{(a)

One use of this notation is for grouping predicate definitions together, for example as class methods, and using term expansion rules to transform the resulting {}-lists into programs.
2.4 Quantified Terms

There are two syntactic forms for quantified terms.

1. Any quantified term can be represented by $Q \ BV \ Body$ provided its quantifier $Q$ has been declared by $op/[3,4]$. $BV$ is a (possibly open) list of bound object variables, and $Body$ is the body, which is another term. Each bound variable can, optionally, be followed by a ":" and a term. If there is only one bound variable, the list notation can be dropped.
   
   For example:
   
   $all \ x \ flower(x, red)$
   $exist [x:Type] \ all \ [lo:int, hi:int] \ lo < f(x) < hi$

2. The escape sequence "!!" may be used to introduce a quantified term whose quantifier might not have been previously declared or whose quantifier is not an atom.
   
   For example,
   
   !!q \ x \ A
   !!Q \ A \ B
   !!integral(A,B) \ x \ T

2.5 Substitutions

The general form of a substitution term is

$[t_1/x_1, \ldots, t_n/x_n]t_m$

where $t_i$ are terms and $x_i$ are object variables.

The substitution $[t_1/x_1, \ldots, t_n/x_n]$ is applied to the term $t_m$.

For example:

$[f(a)/x, t/y]g(X)$
$[[b/z]x/y][t/z]h(A)$

2.6 Programs

A program is composed of a number of predicate definitions. Each predicate definition is made up of a number of clauses. Each clause has a head and an optional body. The head is either an atom or a compound term whose functor is an atom, and the body may be an atom, a meta variable or a compound term. The head and the body are connected together by ":=".
3 Built-in Predicates

3.1 Introduction

This section contains descriptions of the library predicates of Qu-Prolog, grouped by predicate family. The meaning of the terms used may be found in Section 5.

Many of the predicates described in this section are accompanied with mode information and examples. The mode information of each argument of a predicate is represented by a pair consisting of a mode and a type.

The mode is one of

- `-`: must be a variable at the time of call
- `+`: must be supplied at the time of call
- `?`: may be a variable or supplied at the time of call
- `@`: unchanged by call – that is, no bindings to variables in the argument

The type is one of

- `integer`: an integer
- `float`: a double precision float
- `atom`: an atom
- `atomic`: an atom or integer
- `var`: a variable
- `obvar`: an object variable
- `anyvar`: a variable or object variable
- `compound`: a structure
- `gcomp`: a ground structure
- `ground`: a ground term
- `quant`: a quantified term
- `list(Type)`: a list whose elements have type Type
- `closed_list(Type)`: a closed list
- `open_list(Type)`: an open list
- `string`: a string
• **nonvar**: any term other than a variable  
• **term**: any term  
• **goal**: an atom or compound representing a goal.  
• **stream**: a term representing a stream  
• **handle**: a term representing the handle (address) for messages

As an example, the mode information for `=..` is  
mode +nonvar `=..` ?closed_list(term)  
mode -nonvar `=..` @closed_list(term)

The first mode deals with the case where the first argument is a nonvar at the time of call and the second argument will be a closed list of terms by the end of the call. The second mode deals with the case where the first argument is a variable at the time of call (and is instantiated to a nonvar by the end of the call) and the second argument is a closed list of terms that is unchanged by the call.

### 3.2 Control

This set of predicates provides control for the execution of Qu-Prolog.

**Predicates:**

**Goal1, Goal2**  
Conjunction. Goal1 and then Goal2.  
mode `,`,(+goal, +goal)  
Example:  
```plaintext
| ?- A = 10, B is 2 * A.
A = 10
B = 20;
no
```

**Goal1 ; Goal2**  
Disjunction. Goal1 or Goal2.  
mode +goal ; +goal  
Example:  
```plaintext
| ?- A = 10 ; B = 20.
A = 10
B = B;
A = A
B = 20;
no
```
true

Succeed.
Example:

| ?- true. |
| yes |

\+ Goal

Negation. If Goal then fail else succeed.
mode \+ +goal
Example:

| ?- \+ (10 = 20). |
| yes |
| ?- \+ true. |
| no |

Goal1 -> Goal2

If Goal1 then Goal2 else fail.
mode +goal -> +goal
Example:

| ?- true -> A = 10. |
| A = 10; |
| no |
| ?- \+ true -> A = 10. |
| no |

Goal1 -> Goal2 ; Goal3

If Goal1 then Goal2 else Goal3.
mode +goal -> +goal ; +goal
Example:

| ?- true -> A = 10 ; B = 20. |
| A = 10 |
| B = B; |
| no |
| ?- \+ true -> A = 10 ; B = 20. |
| A = A |
| B = 20; |
| no |
The 'cut' operator removes all choices from the parent goal and any goals before the cut in the clause.

Example:

| ?- A = 10 , ! ; B = 20.  
| A = 10  
| B = B;  
| no  

break

Start an invocation of the interpreter. The debugging state is unaffected. A control-D exits the break and returns to the previous level.

Example:

| ?- A = 10 , B = 10 , break, C = 15.  
| [b1] | ?- A = 15.  
| no  
| [b1] | ?- A = B.  
| A = 10  
| B = 10;  
| no  
| [b1] | ?- C = 20.  
| C = 20;  
| no  

fail

Fail.

Example:

| ?- A = 10, fail ; B = 20  
| A = A  
| B = 20;  
| no  

halt

halt(Integer)

Exit Qu-Prolog with exit code 0 or Integer.

Example:

| ?- A = 10, halt ; B = 20
Qu-Prolog exits and returns the user to the system prompt.

call(Goal)

Execute Goal. If Goal has an inline declaration then it will be expanded.
mode call(+goal)

Example:

| ?- call( (A = 10, B = 20) ).
  A = 10
  B = 20;
  no

call(F, A1)
call(F, A1, A2)
call(F, A1, A2, A3)
call(F, A1, A2, A3, A4)
call(F, A1, A2, A3, A5)
call(F, A1, A2, A3, A5, A6)
call(F, A1, A2, A3, A5, A6, A7)
call(F, A1, A2, A3, A5, A6, A7, A8)

Build a goal with functor F and the supplied arguments and call that goal.

callable(Goal)

The same as once((atom(Goal);compound(Goal))).

initialization(Goal)

The same as call(Goal). It is included for compatibility with the ISO standard.

call_predicate(Goal)
call_predicate(Goal,Arg1)
call_predicate(Goal,Arg1,Arg2)
call_predicate(Goal,Arg1,Arg2,Arg3)
call_predicate(Goal,Arg1,Arg2,Arg3,Arg4)

Execute Goal with the required arguments. This is more direct than using call(Goal) (page 20) and, from the point of view of the debugger, behaves like compiled code. Note that Goal can be higher-order, that is, Goal can include arguments.
mode call_predicate(+goal, +term, ...)

Example:

| ?- call_predicate(=, A, 10).
  A = 10;
  no

20
repeat

Succeeds repeatedly.

Example:

| ?- A = 10, repeat, B = 20.  
  A = 10  
  B = 20;  
  A = 10  
  B = 20;  
  A = 10  
  B = 20;  
...

once(Goal)

Execute the Goal and discard any generated alternatives.

mode once(+goal)

Example:

| ?- once( (A = 10, repeat, B = 20) ).  
  A = 10  
  B = 20;  
  no  
| ?- once( (A = 10 ; B = 20) ).  
  A = 10  
  B = B;  
  no

catch(Goal1, Template, Goal2)

Set a trap for Template during the execution of Goal1. Goal2 is executed when a term that unifies with Template is thrown.
catch/3 and throw/1 are typically used for error handling.

mode catch(+goal, +term, +goal)

throw(Template)

Throw Template to the innermost matching catch/3.

mode throw(+term)

Example:

Consider an application that carries out complex processing and may exit at various points within this processing. One way of programming this behaviour is by using catch at the top level of the application as given below.
main(Args) :-
catch(process(Args), exit_throw(Msg), write(Msg)).

When the application is started, process(Args) is executed with a trap set that will write a message when an exit_throw(Msg) is thrown.
The application can exit at a given point within the execution of process(Args) by making a call such as

throw(exit_throw('this is my exit message'))

unwind_protect(Goal1, Goal2)

Succeed if Goal1 succeeds. Goal2 is executed after Goal1, even if Goal1 fails or exits non-locally. Goal2 is called for its side effects only, and any bindings it makes are ignored. Note that it is not currently possible to protect against halt/0 (page 19) or against SIGKILL signal from the operating system.

mode unwind_protect(+goal, +goal)

Example:
Assume that a goal p is to be executed in an environment where the fact fact(a,b) is added to the dynamic database. Further assume that this fact is to be removed when the execution of p finishes with success or failure or because of a throw/1 (page 21) within p, assuming that fact(a,b) is the only clause for fact/2, this can be achieved with the call

unwind_protect((assert(fact(a,b)), p), retract(fact(a,b)))

setup_call_cleanup(Goal0, Goal1, Goal2)

First calls Goal0 deterministically then calls Goal1 and whenever this call fails or succeeds deterministically, then Goal2 is called. It is similar to unwind_protect. The main difference is that unwind_protect is called on success (even if choice points remain) whereas call_cleanup is called on success only when Goal1 has no more choices.

call_cleanup(Goal1, Goal2)

The same as setup_call_cleanup(true, Goal1, Goal2)

3.3 Input / Output

3.3.1 File and Stream Handling

Streams are the basic input/output management units. They provide a uniform interface to files and strings. Each stream can be opened for reading or writing, and data is sent through the stream using term or character input/output predicates. String streams behave in the same way as file streams.
Rather than having a file attached to the stream, a string stream is connected to an atom or a list of `CharCode`. If the string stream is opened for writing, any further writing to the stream is not possible after the successful execution of `stream_to_atom/2` (page 29), `stream_to_string/2` (page 30) or `stream_to_chars/2` (page 30) as these predicates close the stream.

If the process is registered with Pedro then it is possible to open a message stream for either reading or writing.

A term representing a stream is an integer (the stream ID) generated by a call to `open/3` (page 23) or one of the atoms `stdin`, `user_input`, `stdout`, `user_output`, `stderr` and `user_error`, or an atom declared by the user as an alias for a stream.

Predicates:

```
stat(File, Info)
```

- `Info` is a structure of the form `stat(LastModified, Size)` and represents the last time of modification and size of `File`.

```
mode stat(@atom, ?structure)
```

```
access(File, Permission, Result)
```

- `Integer` contains the result of `Permission` check for `File`. `Permission` is an integer made up of mask bits 0,1,2,4 and is used to test if the file exists, and has execute, write and read permissions. `Result` is 0 if the file passes the test and -1 if it fails. The usage is identical to the Unix system call `access(2)`.

So, for example, `access(File, 3, 0)` will succeed if the file is readable and writable.

```
mode access(@atom, @integer, ?integer)
```

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

- `Stream` is the stream resulting from opening `File` with the given `Mode` and `OptionList`.

Modes:

- `read` - Input
- `write` - Output
- `append` - Output

Options:

- `type(Value)`
  - `text` or `binary` stream.
  (Default: `text`.)
• reposition(Boolean)
  Reposition the stream.
  (Default: false.)

• eof_action(Value)
  Action when read past EOF: error, eof_code, reset.
  (Default: error.)

mode open(@atom, @atom, -stream)
mode open(@atom, @atom, -stream, @closed_list(gcomp))

Example:

\[
\begin{align*}
\text{Stream} &= 3 \\
\text{Term1} &= \text{Hello World} \\
\text{Term2} &= \text{Bye}
\end{align*}
\]

open_string(StringMode, Stream)
open_string(StringMode, Stream, OptionList)

Stream is the stream resulting from opening the string with the given StringMode and OptionList. Possible value for OptionList is explained in open/4 (page 23). The possible values for StringMode are

• write - Output
• read(Atom) - Input obtained from the Atom
• read(String) - Input obtained from the String
• read(CharCodeList) - Input obtained from the CharCodeList

mode open_string(@ground, -stream)
mode open_string(@ground, -stream, @closed_list(gcomp))

Example:

\[
\begin{align*}
\text{Stream} &= 3 \\
\text{ Atom} &= \text{stream}
\end{align*}
\]
Stream = 3
Atom = 'Hello World'.
?- open_string(read('p(X,Y).'), Stream),
   read(Stream, Term).
Stream = 3
Term = p(B, A)
?- open_string(read("p(X,Y)."), Stream),
   read(Stream, Term).
Stream = 3
Term = p(B, A)

open_msgstream(Handle, Mode, Stream)

This predicate opens a stream with Mode for an address given by the handle Handle. If the stream is opened for reading then messages from the sender should be of the form

p2pmsg('':myname@mymachine, Handle, msg)

where myname and mymachine are respectively the name of this process and the machine on which this process is running. Handle is the handle specified when opening the stream. The third argument, msg should be a string - it is appended to a stream buffer that can then be read from using the Qu-Prolog input predicates. Messages from this address are ignored by the predicates that look at the message buffer such as ipc_peek.

If the stream is opened for writing then when the stream is flushed, the contents of the stream buffer is wrapped into a message of the form

p2pmsg(Handle, MyHandle, msg)

For most applications the message sends and receives are more appropriate but for interacting with GUI's it is sometimes more convenient to use streams. The xqpdebug GUI uses this mechanism.

set_std_stream(StreamNo, Stream)

This predicate resets one of the standard streams given by StreamNo to Stream. StreamNo must be 0,1 or 2 (stdin,stdout,stderr). This is used in combination with open_msgstream in the xqpdebug GUI so that reads and writes in the debugger become reads and writes via the GUI. This allows the GUI to behave in a transparent way with respect to IO.

reset_std_stream(StreamNo)

Reset the supplied standard stream.

close(Stream)
close(Stream, OptionList)

Close the given Stream. The only possible kind of term in OptionList is
• force(Value)
  If Value is true then close the stream even when there is an error
  condition.
  (Default: false.)

mode close(@stream)
mode close(@stream, @closed_list(gcomp))

at_end_of_stream
at_end_of_stream(Stream)

Succeed if at the end of Stream or the current input stream.
mode at_end_of_stream(@stream)

Example:

| ?- open(filename, write, Stream),
  write_term_list(Stream, [wqa('Hello World'), pc(0'), nl]),
  write_term_list(Stream, [wqa('Bye'), pc(0'), nl]),
  close(Stream).
  Stream = 3
| ?- open(filename, read, Stream),
  at_end_of_stream(Stream),
  close(ReadStream).
  no
| ?- open(filename, read, Stream),
  read(Stream, Term1),
  read(Stream, Term2),
  at_end_of_stream(Stream).
  Stream = 3
  Term1 = Hello World
  Term2 = Bye

current_input(Stream)

The current input stream is Stream.
mode current_input(?stream)

set_input(Stream)

Change the current input stream to Stream. The initial stream is user_input.
mode set_input(@stream)

Examples (Continued from last):

| ?- open(filename, read, Stream),
  set_input(Stream),
  current_input(Input),
  read(Term1), read(Term2),
set_input(stdin).
Stream = 3
Input = 3
Term1 = Hello World
Term2 = Bye

see(File)
Open File for reading and change the current input stream to File.
mode see(@atom)
Examples (Continued from last):

| ?- see(filename),
seeing(File),
read(Term1), read(Term2),
seen.
File = filename
Term1 = Hello World
Term2 = Bye

seeing(File)
Return the file name of the current input stream.
mode seeing(?atom)

seen
Close the current input stream. Change the stream to user_input.

current_output(Stream)
The current output stream is Stream.
mode current_output(?stream)

set_output(Stream)
Change the current output stream to Stream. The initial stream is user_output.
mode set_output(@stream)
Example:

| ?- open_string(write, Stream),
set_output(Stream),
current_output(Output),
write_term_list([wqa('Hello World'), pc(0'), nl]),
stream_to_atom(Stream, Atom),
set_output(stdout).
Stream = 3
Output = 3
Atom = 'Hello World'.

27
tell(File)

Open File for writing and change the current output stream to File.
mode tell(@atom)

Example:

| ?- tell(filename),
telling(File),
write_term_list(Stream, [wqa('Hello World'), pc(0'.), nl]),
write_term_list(Stream, [wqa('Bye'), pc(0'.), nl]),
told.
File = filename
| ?- open(filename, read, Stream),
read(Stream, Term1),
read(Stream, Term2).
Stream = 3
Term1 = Hello World
Term2 = Bye

telling(File)

Return the file name of the current output stream.
mode telling(@atom)
told

Close the current output stream. Change the stream to user_output.

flush_output

flush_output(Stream)

Flush Stream or the current output stream. Forces a write on any remaining output to the stream.

set_autoflush(Stream)

Set Stream to automatically flush after each token is written. This is useful for streams that are to behave like standard error. (The default for streams is to flush only at a newline or at an explicit call to flush_output.)

set_stream_position(Stream, N)

Move Stream to the position N. The reposition flag must have been set when Stream was opened.
mode set_stream_position(@stream, @integer)

stream_property(Stream, Property)

A property of Stream is Property. The possible values for Property are given below.
• alias(Value)
  Alias name.
• end_of_stream(Value)
  no, at, past end of stream.
• eof_action(Value)
  EOF action defined in open/[3,4] (page 23).
• file_name(Value)
  File Name.
• input
  Input Stream.
• line_number(Value)
  Line number for input stream.
• mode(Value)
  open/[3,4] (page 23) mode.
• output
  Output Stream.
• position(Value)
  Position.
• reposition(Value)
  Reposition of stream.
• type(Value)
  Type of stream.

mode stream_property(@stream, ?compound)
Example:

| ?- open(filename, write, Stream),
  stream_property(Stream, file_name(Value)).
| Stream = 3
| Value = filename

stream_to_atom(Stream, Atom)

Stream specifies the output string stream where Atom can be obtained. When this predicate terminates successfully the stream is closed and any further writing to the stream is not possible.

mode stream_to_atom(@stream, -atom)
Example:

| ?- open_string(write, Stream),
  write(Stream, Hello),
  write(Stream, ', '),
  write(Stream, ' '),
write(Stream, World),
stream_to_atom(Stream, Atom).
Stream = 3
Hello = Hello
World = World
Atom = Hello World

stream_to_string(Stream, String)
Stream specifies the output string stream where String can be obtained. When this predicate terminates successfully the stream is closed and any further writing to the stream is not possible.
mode stream_to_string(@stream, -string)
Example:

| ?- open_string(write, Stream),
write(Stream, Hello),
write(Stream, ' '),
write(Stream, World),
stream_to_string(Stream, String).
Stream = 3
Hello = Hello
World = World
String = "Hello World"

stream_to_chars(Stream, CharCodeList)
Stream specifies the output string stream where CharCodeList can be obtained. When this predicate terminates successfully the stream is closed and any further writing to the stream is not possible.
mode stream_to_chars(@stream, -closed_list(integer))
Example:

| ?- open_string(write, Stream),
write(Stream, Hello),
write(Stream, ' '),
write(Stream, World),
stream_to_chars(Stream, Chars).
Stream = 3
Hello = Hello
World = World

An example of the use of string streams is for determining where information will appear in output to a user interface. The following definitions may be used
to retrieve the character positions of matching bracket pairs of a supplied term (assuming the character position of the first character of the term is in position one).

```
brackets(Term, Brackets) :-
    open_string(write, Stream),
    write(Stream, Term),
    stream_to_chars(Stream, CharList),
    match_brackets(CharList, 1, [], Brackets).
match_brackets([], _, _, []).
match_brackets([C|Rest], CurrPos, LeftBr, Brackets) :-
    NewCurrPos is CurrPos + 1,
    ( C = 0'( % open bracket
        ->
        match_brackets(Rest, NewCurrPos, [CurrPos|LeftBr], Brackets)
    ;
    C = 0') % close bracket
        ->
        LeftBr = [LBPos|LBRest],
        Brackets = [[LBPos - CurrPos|Brackets1],
                    match_brackets(Rest, NewCurrPos, LBRest, Brackets1)]
    ;
    match_brackets(Rest, NewCurrPos, LeftBr, Brackets)
).
```

Given the above definitions, the system will behave as follows.

```
| ?- brackets(f(XYZ, g(a, XYZ), h(b)), B).
  XYZ
  B = [9 - 16, 20 - 22, 2 - 23];
  no
| ?- write(f(XYZ, g(a, XYZ), h(b))).
  f(XYZ, g(a, XYZ), h(b))
  XYZ = XYZ;
  no
```

3.3.2 Term Input/Output

For input/output predicates, if the stream is specified explicitly, the stream appears as the first argument. Otherwise, the current input/output stream is assumed.
Qu-Prolog supports multiple operator tables and object variable prefix tables. At any time, there is one active operator table and one active object variable prefix table. Active tables can be changed and new tables can be created by op_table/1 (page 32) and obvar_prefix_table/1 (page 34). When a new operator table is created, the comma ‘,’ operator is automatically declared. An inheritance mechanism is also available for operator tables. The input/output predicates can use a table other than the current active table by specifying the name of the table in the OptionList.

The input/output predicates are enhanced with the ability to remember the association between the ASCII representation and the internal representation for both meta and object variables over multiple input/output operations. This enables the user to refer to the same variable over a number of input/output operations, and is useful in applications with interactive environments, such as interactive theorem provers. The association is not remembered for any variable that begins with an underscore (‘_’).

Qu-Prolog supports high-speed input/output using byte-encoded Qu-Prolog terms. Byte-encoded terms are written with encoded_write_term/3 (page 45) and read with encoded_read_term/3 (page 40). The compiler can accept byte-encoded files as input and generates some intermediate files in this form (see the online manual qc(1) for further details). consult/1 (page 101) also accepts such files as input. The system is supplied with a decoder for byte-encoded terms (see the online manual qecat(1) for further details).

Predicates:

\texttt{op_table(Table)}

Retrieve or change the current operator table. The initial table is user. Each new table is initialised with a comma ‘,’ when it is first created.

\texttt{mode op_table(?atom)}

Example:

| ?- op_table(Table).  
| Table = user;  
| no  
| ?- op_table(mine).  
| yes  
| ?- op_table(Table).  
| Table = mine;  
| no

\texttt{op_table_inherit(Table1, Table2)}

Table1 inherits all the operators declared in Table2. These operators do not include the ones which are inherited by Table2 from a third table. That is, inheritance is non-transitive.

\texttt{mode op_table_inherit(@atom, @atom)}
Examples (Continued from last):

?- op_table_inherit(mine, user).
yes

\[\text{current\_op(\text{Precedence, Associativity, Operator})}\]
\[\text{current\_op(\text{Table, Precedence, Associativity, Operator})}\]

\text{Operator} is an operator in Table (default: the current operator table) with Precedence and Associativity.
\text{mode current\_op(?integer, ?atom, ?atom)}
\text{mode current\_op(?atom, ?integer, ?atom, ?atom)}

Examples (Continued from last):

?- current\_op(700, Assoc, Op).
\text{Assoc} = xfx
\text{Op} = =;
\text{Assoc} = xfx
\text{Op} = \neq;
\text{Assoc} = xfx
\text{Op} = ?=;
\text{Assoc} = xfx
\text{Op} = ==

\[\text{op(\text{Precedence, Associativity, Operator})}\]
\[\text{op(\text{Table, Precedence, Associativity, Operator})}\]

Declare \text{Operator} with \text{Precedence} and \text{Associativity} as an operator in Table or in the current operator table. \text{Precedence} is any number between 0 and 1200, where higher numbers given lower precedence (loose binding).

\text{fx} Prefix. Argument has lower precedence than operator.
\text{fy} Prefix. Argument has the same precedence as operator.
\text{quant} Quantifier. Argument has the same precedence as operator.
\text{xfx} Infix. Not associative.
\text{xfy} Infix. Right associative.
\text{yfx} Infix. Left associative.
\text{xf} Postfix. Argument has lower precedence than operator.
\text{yf} Postfix. Argument has the same precedence as operator.

If \text{Precedence} is 0, any previous declaration is removed.
\text{mode op(?integer, @atom, @atom)}
\text{mode op(?atom, @integer, @atom, @atom)}

Example: (Continued from last):
obvar_prefix_table(Table)

Retrieve or change the current object variable prefix table. The initial table is user.

mode obvar_prefix_table(?atom)

Example:

?- obvar_prefix_table(Table).
Table = user;
no
?- obvar_prefix_table(mine).
yes
?- obvar_prefix_table(Table).
Table = mine;
no

current_obvar_prefix(Atom)
current_obvar_prefix(Table, Atom)

Atom is an object variable prefix in Table
(default: the current object variable prefix table).

mode current_obvar_prefix(?atom)

mode current_obvar_prefix(@atom, ?atom)

Examples (Continued from last):

?- current_obvar_prefix(Atom).
no

obvar_prefix(AtomList)
obvar_prefix(Table, AtomList)

Declares all the atoms in AtomList as object variable prefixes in Table
or the current object variable prefix table. This also implicitly declares
all variant names produced by extending the variable name with a com-
bination of underscores and sequences of digits and letters. The name of
the constant being declared as an object variable prefix must be made up
of lower case letters.
mode obvar_prefix(@closed_list(atom))
mode obvar_prefix(@atom)
mode obvar_prefix(@atom, @closed_list(atom))
mode obvar_prefix(@atom, @atom)

Examples (Continued from last):

| ?- obvar_prefix(a).
yes
| ?- obvar_prefix(user, [x , y , z]).
yes
| ?- current_obvar_prefix(Atom).
A = a;
no
| ?- current_obvar_prefix(user, Atom).
A = x;
A = y;
A = z;
no

remove_obvar_prefix(Atom)
remove_obvar_prefix(Table, Atom)

Remove the object variable prefix specified by Atom from Table or the current object variable prefix table.
mode remove_obvar_prefix(@atom)
mode remove_obvar_prefix(@atom, @atom)

Examples (Continued from last):

| ?- remove_obvar_prefix('a').
yes
| ?- remove_obvar_prefix(user, 'z').
yes
| ?- current_obvar_prefix(Atom).
no
| ?- current_obvar_prefix(user, Atom).
A = x;
A = y;
no

obvar_name_to_prefix(Atom, Atom)

Strips any trailing numbers or underscores from the first Atom and unifies the result with the second Atom. This predicate does not actually check whether the computed prefix is, in fact, a current object variable prefix. It is merely provided for convenience.
mode obvar_name_to_prefix(@atom, ?atom)

Examples (Continued from last):
?- obvar_name_to_prefix('!_a_1', A).
A = !_a;
no
?- obvar_name_to_prefix('y_28_1_3', A).
A = y;
no

set_obvar_name(ObVar, Name)
Set ObVar to have the base name Name. A suffix will be added to Name to complete the final name for ObVar. If ObVar has a name already, the predicate fails. See obvar_prefix/[1,2] (page 34).
mode set_obvar_name(+obvar, @atom)
Example:
?- set_obvar_name(!_y_3, x).
_y_3 = !x_0;
no

set_var_name(Var, Name)
Set Var to have the base name Name. A suffix will be added to Name to complete the final name for Var. If Var has a name already, the predicate fails.
mode set_var_name(+obvar, @atom)
Example:
?- set_var_name(_32, 'X').
_32 = X_0;
no

get_var_name(Variable, Name)
Variable has Name. Fails if Variable has no name.
mode get_var_name(@anyvar, ?atom)
Examples (Assuming x is still a current object variable prefix):
?- get_var_name(Max, Name), write(Name), nl, fail. Max
no
?- get_var_name(_32, Name), write(Name), nl, fail. no
?- get_var_name(!x_3, Name), write(Name), nl, fail. x_3
no

36
get_unnamed_vars(Term, VarList)

VarList is the list of unnamed (object) variables in Term.
mode get_unnamed_vars(@term, -closed_list(anyvar))

Examples (Assuming x is still a current object variable prefix):

| ?- get_unnamed_vars((A, _Y, !x, !_y), Vars), write(Vars), fail.  
 [_117, !_x0]  
  no

name_vars(Term)
name_vars(Term, VarList)

Name all unnamed (object) variables in Term returning them in VarList.
mode name_vars(+term)
mode name_vars(+term, -closed_list(anyvar))

Example:

| ?- Term = f(X, _Y, !x, !_x), write(Term) , nl, fail.  
f(X, _F5, !x, !_x0)  
  no  
?- Term = f(X, _Y, !x, !_x), name_vars(Term), write(Term), nl, fail.  
f(X, A, !x, !x0)  
  no

error(Term)

Write Term to user_error.
mode error(@term)

Example:

| ?- error('Hello'), error(' '), error('World').  
Hello World  
yes

errornl(Term)
errornl

Write Term and a newline to user_error, or write a newline to user_error.
mode errornl(@term)

Example:

| ?- errornl('Hello'), errornl('World').  
Hello World  
yes
read(Term)

read(Stream, Term)

Read Term from Stream or the current input stream.

mode read(?term)

mode read(@stream, ?term)

Example:

| ?- read(Term), write(Term), nl, fail.
| f(X, _Y, !x, !_x)
| f(_1A2, _19D, !_x0, !_x1)
| no

read_term(Term, OptionList)

read_term(Stream, Term, OptionList)

Read Term from Stream or the current input stream. OptionList is a list whose entries are chosen from the items given below.

1. Input Options
   Input options not mentioned in OptionList take on their default values.
   
   - **remember_name(Value)**
     Remember the names for all the variables other than those beginning with an underscore in Term.
     (Default: false.)
   
   - **op_table(Value)**
     Use the operator table given in Value.
     (Default: Current table.)
   
   - **obvar_prefix_table(Value)**
     Use the object variable prefix table given in Value.
     (Default: Current table.)

2. Output Options.
   The Value for each output option is instantiated during the reading of Term.
   
   - **variables(Value)**
     All variables, including anonymous variables, from left to right.
     (Default: []) Typically Value is a variable that is bound to the list of variables appearing in the term.
   
   - **variable_names(Value)**
     Variable=Name pairs, excluding anonymous variables.
     (Default: [])
• singletons(Value)
  Variable=Name pair for all singleton variables, excluding anonymous variables.
  (Default: [])

mode read_term(?term, +closed_list(compound))
mode read_term(@stream, ?term, +closed_list(compound))

Example:

| ?- read_term(Term, []), write(Term), nl.  
| f(X, _Y, !_x, !_x)  
| f(_3BE, _3B4, !_x0, !_x1)  
| Term = f(A, B, !_x0, !_x1)  
| ?- read_term(Term, [variables(Var)]),  
| write(Var), nl, fail.  
| f(X, _Y, !_x, !_x)  
| [_1F1, _1EC, !_x0, !_x1]  
no
| ?- read_term(Term, [variable_names(Var)]),  
| write(Var), nl, fail.  
| f(X, _Y, !_x, !_x)  
| [_1F1 = X, _1EC = _Y, !_x0 = x, !_x1 = _x]  
no
| ?- read_term(Term, [remember_name(true)]),  
| write(Term), nl, fail.  
| f(X, _Y, !_x, !_x)  
| f(X, _1DC, !_x, _-_x0)  
no

read_1_term(Term, VariableNames)
read_1_term(Stream, Term, VariableNames)

Read Term from Stream or the current input stream. Return the list of variables and their names in VariableNames.
It is the same as read_term(Term, [variable_names(VariableNames)]) (page 38).

mode read_1_term(?term, -closed_list(compound))
mode read_1_term(@stream, ?term, -closed_list(compound))

Example:

| ?- read_1_term(Term, Var), write(Var), nl, fail.  
| f(X, _Y, !_x, !_x)  
| [_1F1 = X, _1EC = _Y, !_x0 = x, !_x1 = _x]  
no

readR(Term)
readR(Stream, Term)
Read Term from Stream or the current input stream. Remember the names for all the variables other than those beginning with an underscore in Term.
It is the same as

\[ \text{read_term}(\text{Term}, \text{[remember_name(true)]}) \]

\text{mode readR(?term)}
\text{mode readR(@stream, ?term)}

Example:

\[ f(X, _Y, !x, !_x) \]
\[ f(X, _1DC, !x, !_x0) \]
no

\text{readR_1_term(Term, VariableNames)}
\text{readR_1_term(Stream, Term, VariableNames)}

Read Term from Stream or the current input stream. Remember the names for all the variables in Term, and return the list of variables and their names in VariableNames.
It is the same as

\[ \text{read_term}(\text{Term}, \text{[variable_names(VariableNames), remember_name(true)]}) \]

\text{mode readR_1_term(?term, -closed_list(compound))}
\text{mode readR_1_term(@stream, ?term, -closed_list(compound))}

Example:

\[ f(X, _Y, !x, !_x) \]
\[ f(X, _217, !x, !_x0) \]

\[ [X = X, _217 = _Y, !x = x, !_x0 = _x] \]
no

\text{encoded_read_term(Term, OptionList)}
\text{encoded_read_term(Stream, Term, OptionList)}

Read Term from the encoded stream Stream or the current input stream.
The OptionList is the same as for \text{read_term/[2,3]} (page 38).
\text{mode encoded_read_term(@term, +closed_list(compound))}
\text{mode encoded_read_term(@stream, @term, +closed_list(compound))}
write(Term)
write(Stream, Term)

Write Term to Stream or the current output stream.

mode write(?term)
mode write(@stream, ?term)

Example:

| ?- write(f(X, _Y, !x, !_x)), nl, fail.  
f(X, _D0, !x, !_x0)  
no

write_term(Term, OptionList)
write_term(Stream, Term, OptionList)

Write Term to Stream or the current output stream. OptionList is a list whose entries are chosen from the items given below. Items not mentioned in OptionList take on their default values.

- quoted(Value)
  Meaning: Quote all unsafe atoms.
  (Default: false.)

- ignore_ops(Value)
  Meaning: Output operators in structural format.
  (Default: false.)

- numbervars(Value)
  Meaning: Generate variable names (not including object variables).
  (Default: false.)

- remember_name(Value)
  Meaning: Generate and remember variable names (including object variables).
  (Default: false.)

- op_table(Value)
  Meaning: Operator table.
  (Default: Current table.)

- max_depth(Value)
  Meaning: Print depth limit.
  (Default: 0 (unlimited).)

- obvar_prefix_table(Value)
  Meaning: Object variable prefix table.
  (Default: Current table.)

mode write_term(@term, @closed_list(compound))
mode write_term(@stream, @term, @closed_list(compound))

Example:
Write Term to Stream or the current output stream. The trailing R, T, q determine if variable names are to be created and remembered, if an operator table is to be used, and if unsafe atoms are to be quoted. These variants are faster than using the option list method above.

\begin{verbatim}
mode write{R,q}(@term)
mode write{R,q}(@stream, @term)
mode write{R,T,q}(@term, @atom)
mode write{R,T,q}(@stream, @term, @atom)
\end{verbatim}

Example:

\begin{verbatim}
| ?- writeR(f(X, _Y, !x, !_y)), nl, fail.
f(X, A, !x, !x0)
no
\end{verbatim}

Write Atom to Stream or the current output stream. write_atom is faster than write (page 41) as it avoids many of the tests in write. Unsafe atoms are quoted if writeq_atom is used.

\begin{verbatim}
mode write_atom(@atom)
mode write_atom(@stream, @atom)
\end{verbatim}

Example:

\begin{verbatim}
42
\end{verbatim}
| ?- write_atom('Hello'), nl, fail. Hello no |
| ?- writeq_atom('Hello'), nl, fail. 'Hello' no |

write_canonical(Term)
write_canonical(Stream, Term)

Write Term to Stream or the current output stream. All the operators are written in structural format.
It is the same as write_term(Term, [ignore_ops(True)]) (page 41).
mode write_canonical(@term)
mode write_canonical(@stream, @term)

Example:
| ?- write_canonical((A = B + C * D)), nl, fail. (=)(A, (+)(B, (*)(C, D))) no |

write_integer(Stream, Integer)

Write Integer to Stream. write_integer is faster than write (page 41) as it avoids many of the tests in write.
mode write_integer(@integer)
mode write_integer(@stream, @integer)

Example:
| ?- write_integer(stdout, 42), nl, fail. 42 no |

write_string(Stream, String)

Write String to Stream but without the quotes.
mode write_string(@string)
mode write_string(@stream, @string)

Example:
| ?- write_string(stdout, "Hello World"). Hello World |

writeq_string(Stream, String)
Write String to Stream with quotes.
mode writeq_string(@string)
mode writeq_string(@stream, @string)

Example:

| :- writeq_string(stdout, "Hello World").
"Hello World"

write_term_list(Message)
write_term_list(Stream, Message)

Write terms to Stream or the current output stream according to the formatting information in the list Message described below. The Qu-Prolog compiler unfolds this predicate to low-level calls to carry out the writing and is therefore the most efficient way to output a sequence of terms. The call write_term_list(stdout, Message) is faster than write_term_list(Message) because the current output stream does not need to be looked up.

- nl (page 48)
  Meaning: New line.
- sp
  Meaning: Space.
- tab(N) (page 48)
  Meaning: Write N spaces.
- wa(Atom)
  Meaning: Use write_atom (page 42) for Atom.
- wqa(Atom)
  Meaning: Use writeq_atom (page 42) for Atom.
- wi(Integer)
  Meaning: Use write_integer (page 43) for Integer.
- pc(Code)
  Meaning: Use put_code (page 49) for Code.
- w(Term)
  Meaning: Use write (page 41) for Term.
- q(Term)
  Meaning: Use writeq (page 41) for Term.
- wR(Term)
  Meaning: Use writeR (page 41) for Term.
- wRq(Term)
  Meaning: Use writeRq (page 41) for Term.
- wl(List, Sep)
  Meaning: Write the elements of List using write (page 41) with Sep as a list separator.
mode write_term_list(@closed_list(term))
mode write_term_list(@stream, @closed_list(term))

Example:

| ?- write_term_list([wa('Hello'), sp, wa('World'), nl, wqa('Hello World'), pc(0')])), nl, fail.
Hello World
'Hello World'!
no

| ?- write_term_list([wi(42), tab(5), w(f(X, _Y, !x, !_x)), sp, pc(0'), sp, wRq(f(X, _Y, !x, !_x)), nl, w([31, 12, 1999], '/')]), nl, fail.
42 f(X, _197, !x, !_x0) - f(X, A, !x, !_x0)
31/12/1999
no

encoded_write_term(Term, OptionList)
encoded_write_term(Stream, Term, OptionList)

Write Term to the encoded stream Stream or the current output stream. The OptionList is the same as for write_term/[2,3] (page 41).
mode encoded_write_term(+term, +closed_list(compound))
mode encoded_write_term(@stream, +term, +closed_list(compound))

portray_clause(Clause)
portray_clause(Stream, Clause)

Pretty print Clause in Stream or the current output stream.
mode portray_clause(@compound)
mode portray_clause(@stream, @compound)

Example:

| ?- portray_clause((p(X,Y) :- q(X,Z), r(Z,Y), s(Y))), fail.
p(X, Y) :-
  q(X, Z),
  r(Z, Y),
  s(Y).
no

print(Term)
print(Stream, Term)

The same as writeq except that the dynamic predicate portray/1 is first tried. If that succeeds then writeq is not called. This provides a hook for user defined term writing.
mode print(@term)
mode print(@stream, @term)
portray(Term)

A dynamic predicate used as a hook for user defined term writing in print.

3.3.3 Character Input/Output

These predicates can be divided into two categories according to the representation of the character used. The \_char predicates deal with the character as a single letter atom. The \_code predicates handle the character in its character code representation, which is an integer. Any predicate without these suffixes are intended to be compatible with existing Prologs, which use character codes. Predicates without a stream argument use the current input/output stream.

Predicates:

<table>
<thead>
<tr>
<th>get(CharCode)</th>
<th>Get the next visible character from the current input stream, and unify it with CharCode.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode get(?integer)</td>
<td>mode get(@stream, ?integer)</td>
</tr>
<tr>
<td>Note: non-visible characters are character codes less than 33. This includes:</td>
<td></td>
</tr>
<tr>
<td>Tab Character code: 9</td>
<td>New line Character code: 10</td>
</tr>
</tbody>
</table>

Examples (In the second example, a 'tab' precedes the 'A'):

<table>
<thead>
<tr>
<th>?- get(CharCode).</th>
</tr>
</thead>
<tbody>
<tr>
<td>A CharCode = 65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?- get(CharCode).</th>
</tr>
</thead>
<tbody>
<tr>
<td>A CharCode = 65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>get0(CharCode)</th>
<th>Get the next character from the current input stream, and unify it with CharCode. The same as get Code/[1,2] (page 47).</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode get0(?integer)</td>
<td>mode get0(@stream, ?integer)</td>
</tr>
</tbody>
</table>

Examples (In the second example, a 'tab' precedes the 'A'):
get_char(Character)
get_char(Stream, Character)

Character is the next character from Stream or the current input stream.
mode get_char(?atom)
mode get_char(@stream, ?atom)

Example:

?- get_char(Char).
A
Char = A
no

get_code(CharCode)
get_code(Stream, CharCode)

CharCode is the next character code from Stream or the current input stream.
mode get_code(?integer)
mode get_code(@stream, ?integer)

Examples (In the second example, a 'tab' precedes the 'A'):

?- get_code(CharCode).
A
CharCode = 65
?- get_code(CharCode).
A
CharCode = 9

get_line(String)
get_line(Stream, String)

String is the next line from Stream or the current input stream. The newline is consumed but is not part of the returned list. String is instantiated to -1 at EOF.
mode get_line(?string)
mode get_line(@stream, ?string)

put_line(CodeList)
put_line(Stream, CodeList)
CodeList is written to Stream or the current output stream. A newline is added.

mode put_line(@list(integer))
mode put_line(@string)
mode put_line(@stream, @list(integer))
mode put_line(@stream, @string)

Example:

| ?- get_line(L), put_line(L).
  abc
  abc
  L = "abc"

skip(CharCode)
skip(Stream, CharCode)

Skip the input from Stream or the current input stream until after the first occurrence of CharCode. It is assumed that CharCode occurs in Stream.

mode skip(@integer)
mode skip(@stream, @integer)

Example:

| ?- skip(0',), get_char(Char).
  ABC*DEF*GHI
  Char = D;
  no

nl
nl(Stream)

Write a new line on Stream or the current output stream.

mode nl(@stream)

Example:

| ?- write(‘Hello’), nl, write(‘World’), nl.
  Hello
  World
  yes

tab(N)
tab(Stream, N)

Output N spaces to Stream or the current output stream.

mode tab(@integer)
mode tab(@stream, @integer)

Example:
```prolog
| ?- write('Hello'), tab(5), write('World'), nl.
Hello World
yes

put(CharCode)
Send CharCode to the current output stream.
mode put(@integer)
Example:
| ?- put(0'H), put(0'e), put(0'l), put(0'l), put(0'o), nl.
Hello
yes

put_code(CharCode)
put_code(Stream, CharCode)
Send the character code, CharCode, to Stream or the current output stream.
mode put_code(@integer)
mode put_code(@stream, @integer)
Example:
| ?- put_code(0'H), put_code(0'e), put_code(0'l), put_code(0'l), put_code(0'o), nl.
Hello
yes

put_char(Character)
put_char(Stream, Character)
Send Character to Stream or the current output stream.
mode put_char(@atom)
mode put_char(@stream, @atom)
Example:
| ?- put_char('H'), put_char('e'), put_char('l'), put_char('l'), put_char('o'), nl.
Hello
yes

3.4 Terms
3.5 Comparison of Terms
Two terms are compared according to the standard ordering, which is defined below. Items listed at the beginning come before the items listed at the end. For example, meta variables are less than object variables in the standard ordering.
1. Meta variables, in age ordering (older variables come before younger variables). If the variables are the same and there are substitutions, then the substitutions are compared as lists using the standard ordering.

2. Object variables, in age ordering (older variables come before younger variables). If the variables are the same and there are substitutions, then the substitutions are compared as lists using the standard ordering.

3. Integers, in numerical ordering.

4. Atoms, in character code (ASCII) ordering.

5. Compound terms (including lists) are compared in the following order:
   (a) Arity, in numerical ordering.
   (b) Functor, in standard ordering.
   (c) Arguments, in standard ordering, from left to right.
   If the terms are the same and there are substitutions, then the substitutions are compared as lists using the standard ordering.

6. Quantified terms are compared in the following order:
   (a) Quantifier, in standard ordering.
   (b) Bound variables list, in standard ordering.
   (c) Body, in standard ordering.
   If the terms are the same and there are substitutions, then the substitutions are compared as lists using the standard ordering.

Predicates:

```
Term1 == Term2
```

Term1 and Term2 are alpha-equivalent without instantiation.
```
 mode @term == @term
```

Example:
```
| ?- 10 * 2 == 10 + 10.  
   no                       
| ?- A == 10 * 2.        
   no                      
| ?- A = 10 * 2, A == 10 * 2.  
   A = 10 * 2;            
   no                      
```

```
Term1 \== Term2
```

Term1 and Term2 are not alpha-equivalent without instantiation.

mode @term \== @term

Example:

| ?- 10 * 2 \== 10 + 10. |
| yes                     |
| ?- A = 10 * 2, A \== 10 * 2. |
| no                      |

Term1 \= Term2

Term1 equals to Term2 in the standard order.
mode @term \= @term

Term1 \< Term2

Term1 precedes Term2 in the standard order.
mode @term \< @term

Term1 \=< Term2

Term1 precedes or equals to Term2 in the standard order.
mode @term \=< @term

Term1 \> Term2

Term1 follows Term2 in the standard order.
mode @term \> @term

Term1 \>= Term2

Term1 follows or equals to Term2 in the standard order.
mode @term \>= @term

compare(Operator, Term1, Term2)

The relation Operator holds between Term1 and Term2.
The possible choices of Operator are given below.

=  If \=/2 holds.
<  If \</2 holds.
>  If \>/2 holds.

mode compare(?atom, @term, @term)

3.6 Testing of Terms

These testing predicates are used to determine various properties of the data objects, or apply constraints to the data objects.

Predicates:

simple(Term)
Succeed if `Term` is atomic or any variable.
mode `simple(@term)`
Example:

```prolog
| ?- simple(atom). yes
| ?- simple(10). yes
| ?- simple(Var).
Var = Var
| ?- simple(!obvar).
obvar = !obvar
| ?- simple([a/!x]Var).
x = !x
Var = Var
| ?- simple(functor(arg1, arg2)). no
| ?- simple([list1, list2]). no

| ?- simple(!!quant !x Var). no
```

atomic(Term)
Succeed if `Term` is an atom or a number.
mode `atomic(@term)`
Example:

```prolog
| ?- atomic(atom). yes
| ?- atomic(10). yes
| ?- atomic(Var). no
```

atom(Term)
Succeed if `Term` is an atom.
mode `atom(@term)`
Example:

```prolog
| ?- atom(atom). yes
| ?- atom(10). no
```
number(Term)

Succeed if Term is a number.
mode number(@term)

Example:

| ?- number(10).      | yes |
| ?- number(3.4).     | yes |
| ?- number(atom).    | no  |

integer(Term)

Succeed if Term is an integer.
mode integer(@term)

Example:

| ?- integer(10).     | yes |
| ?- integer(3.4).    | no  |
| ?- integer(atom).   | no  |

float(Term)

Succeed if Term is a double.
mode float(@term)

Example:

| ?- float(10).       | no  |
| ?- float(3.4).      | yes |
| ?- float(atom).     | no  |

any_variable(Term)

Term is a meta or an object variable.
mode any_variable(@term)

Example:
| ?- any_variable(Var). Var = Var |
| ?- any_variable(!obvar). obvar = !obvar |
| ?- any_variable([a/!x]Var). x = !x Var = Var |
| ?- any_variable(atom). no |
| ?- any_variable(f(G)). no |

var(Term)

Succeed if Term is a meta variable.
mode var(@term)
Example:

| ?- var(Var). Var = Var |
| ?- var(!obvar). no |
| ?- var([a/!x]Var). x = !x Var = Var |

nonvar(Term)

Succeed if Term is not a meta variable.
mode nonvar(@term)
Example:

| ?- nonvar(Var). no |
| ?- nonvar(!obvar). obvar = !obvar |
| ?- nonvar([a/!x]Var). no |

| ?- nonvar(f(G)). G = G |

ground(Term)
Succeed if Term does not contain any meta variables (after simplifying substitutions).

mode ground(@term)

Example:

| ?- ground(Var).  |
| no              |
| ?- ground(!obvar).  |
| obvar = !obvar |
| ?- ground([a/!x]Var).  |
| no              |
| ?- ground(f(G)).  |
| no              |

obvar(Term)

Succeed if Term is an object variable.

mode obvar(@term)

Example:

| ?- obvar(Var).  |
| no              |
| ?- obvar(!obvar).  |
| obvar = !obvar |
| ?- obvar([a/!x]!y).  |
| x = !x           |
| y = !y           |
| no              |

compound(Term)

Succeed if Term is a structure or a list.

mode compound(@term)

Example:

| ?- compound(f(arg1, g(arg2))).  |
| yes                          |
| ?- compound([list1, list2]).  |
| yes                          |
| ?- compound(!!qant !x Var).  |
| no                           |
| ?- compound(atom).  |
| no                           |
| ?- compound(Var).  |
| 55                          |
no
| ?- compound([a/!x]Var).
  no

list(Term)
Term is a (possibly empty) list.
mode list(@term)
Example:
| ?- list(f(arg1, arg2)).
  no
| ?- list([list1, list2]).
  yes
| ?- list([a/!x]Var).
  no

string(Term)
Term is a string.
mode string(@term)
Example:
| ?- string([40,41]).
  no
| ?- string("ab").
  yes

quant(Term)
Succeed if Term is a quantified term.
mode quant(@term)
Example:
| ?- quant(!!quant !x Var).
  x = !x
  Var = Var
| ?- quant(f(!!quant !x Var)).
  no
| ?- quant([a/!x]Var).
  no

sub(Term)
Succeed if Term has a substitution at the outermost level.
mode sub(@term)
Example:
| ?- sub([a/x]Var).
x = !x
Var = Var
| ?- sub(f([a/x]Var)).
no

std_var(Term)

Equivalent to var(Term), \+ sub(Term) but faster.
mode std_var(@term)

Example:

| ?- std_var(Var).
Var = Var
| ?- std_var([A/a]Var).
no

std_nonvar(Term)

Equivalent to nonvar(Term), \+ sub(Term) but faster.
mode std_nonvar(@term)

std_compound(Term)

Term is compound with an atom as the functor.
mode std_compound(@term)

identical_or_apart(Term1, Term2)

True when the terms Term1 and Term2 are either identical or non-unifiable.
mode identical_or_apart(@term, @term)

Example:

| ?- identical_or_apart(A, B).
no
| ?- A is 10, identical_or_apart(A,B).
no
| ?- A is 10, B is 10, identical_or_apart(A,B).
A = 10
B = 10
| ?- A is 10, B is 20, identical_or_apart(A,B).
A = 10
B = 20

is_free_in(ObVar, Term)
is_not_free_in(ObVar, Term)
Succeed if ObVar is known to be (not) free in Term.

Both is_free_in and is_not_free_in are infix operators.

\[ \text{mode is_free_in(+obvar, +term)} \]
\[ \text{mode is_not_free_in(+obvar, +term)} \]

Example:

\[
\begin{align*}
\text{?- !x is_free_in f(!y, !x, !z).} \\
x & = !x \\
y & = !y \\
z & = !z \\
\text{?- !x is_free_in f(!y, g(!x), !z).} \\
x & = !x \\
y & = !y \\
z & = !z \\
\text{?- !x is_free_in f(X).} \\
\text{no} \\
\text{?- !x is_free_in atom.} \\
\text{no} \\
\text{?- !x is_free_in !!q !x B.} \\
\text{no} \\
\text{?- !x is_free_in [a/!x]B.} \\
\text{no} \\
\text{?- !x is_not_free_in f(!y, !x, !z).} \\
\text{no} \\
\text{?- !x is_not_free_in f(!y, g(!x), !z).} \\
\text{no} \\
\text{?- !x is_not_free_in f(X).} \\
\text{no} \\
\text{?- !x is_not_free_in atom.} \\
x & = !x \\
\text{?- !x is_not_free_in !!q !x B.} \\
x & = !x \\
B & = B \\
\text{?- !x is_not_free_in [a/!x]B.} \\
x & = !x \\
B & = B
\end{align*}
\]

\[ \text{not_free_in(ObVar, Term)} \]

Apply the not free in constraint. not_free_in is an infix operator.

\[ \text{mode not_free_in(+obvar, +term)} \]

Example:

\[
\begin{align*}
\text{?- !x not_free_in f(!y, !x, !z).} \\
\text{no}
\end{align*}
\]
is_distinct(ObVar1, ObVar2)
Succeed if ObVar1 and ObVar2 are known to represent different object-level variables.
mode is_distinct(@term, @term)
Example:
| ?- is_distinct(!x, !y).
no
| ?- !x not_free_in !y, is_distinct(!x, !y).
x = !x
y = !y
provided:
!y not_free_in [!x]
!x not_free_in [!y]

check_binder(VarList, DistinctList)
Check VarList is a valid bound variable list for a quantified term. The check also ensures every object variable in the list is different from every other. The variables in VarList are made distinct from the object variables in DistinctList, which is a closed list. If VarList is an open list, the call will be delayed when the variable representing the open end of the list is reached. This is not expected to be used directly. It is called automatically by the system when quantified terms are constructed.
3.7 Term Manipulation

This set of meta-logical predicates perform various operations over the data objects. These operations include composition/decomposition of terms, conversion of a data object from one form to another, and simplification.

Predicates:

Term =..[Functor|Arguments]

Term is a compound composed of Functor and Arguments.
mode +nonvar =.. ?cloded_list(term)
mode -nonvar =.. @cloded_list(term)
Example:

| ?- f(a, B, _C, !d, !_e) =.. List, write(List), nl, fail. [f, a, B, _119, !d, !_x0] no |

| ?- Funct =.. [f, a, B, _C, !d, !_e], write(Funct), nl, fail. f(a, B, _114, !d, !_x0) no |

| ?- f(g(a), h(b)) =.. List, write(List), nl, fail. [f, g(a), h(b)] no |

functor(Term, Functor, Arity)

Term is compound with Functor and Arity.
mode functor(-compound, +term, +integer)
mode functor(-atomic, +term, +integer)
mode functor(+compound, ?term, ?integer)
mode functor(@atomic, ?term, ?integer)
Example:

| ?- functor(f(X), Funct, Arg).
X = X
Funct = f
Arg = 1 |

| ?- functor(Term, f(X), 1).
Term = f(X)(A)
X = X |

arg(N, Term, Argument)

The N-th argument of Term is Argument.
mode arg(@integer, +compound, +term)
If \textsc{Term} has no \(N\)-th argument then an out-of-range exception will occur.

Example:

\begin{verbatim}
| ?- arg(2, f(a, B, c), b).
  B = b
| ?- arg(3, f(a, B, c), C).
  B = B
  C = c
| ?- arg(4, f(a, b, c), Arg).
  Unrecoverable error: argument 1 of arg(4, f(a, b, c), _128)
  must be in range (see manual)
  no
\end{verbatim}

\texttt{same\_args(Term1, Term2, N1, N2)}

Succeed when \textsc{Term1} and \textsc{Term2} are both atomic or compound and the arguments from position \(N1\) to \(N2\) in \textsc{Term1} and \textsc{Term2} are the same.

\begin{verbatim}
mode same\_args(+compound, +compound, @integer, @integer)
mode same\_args(@atomic, @atomic, @integer, @integer)
mode same\_args(+compound, @atomic, @integer, @integer)
mode same\_args(@atomic, +compound, @integer, @integer)
\end{verbatim}

Example:

\begin{verbatim}
| ?- same\_args(f(a,b,c,d), g(A,B,C), 1, 2).
  A = a
  B = b
  C = C
\end{verbatim}

\texttt{same\_args(Term1, Term2, N1, N2, N3)}

Succeed when \textsc{Term1} and \textsc{Term2} are both atomic or compound and the arguments from position \(N1\) to \(N2\) of \textsc{Term1} are the same as the arguments from position \(N3-N1+N2\) of \textsc{Term2}.

\begin{verbatim}
mode same\_args(+compound, +compound, @integer, @integer)
mode same\_args(@atomic, @atomic, @integer, @integer)
mode same\_args(+compound, @atomic, @integer, @integer)
mode same\_args(@atomic, +compound, @integer, @integer)
\end{verbatim}

\texttt{setarg(N, Term, Argument)}

The \(N\)-th argument of \textsc{Term} is replaced by \texttt{Argument}.

\begin{verbatim}
mode setarg(@integer, +compound, +term)
\end{verbatim}

Warning: this predicate does a backtrackable destructive update of the structure.

If \textsc{Term} has no \(N\)-th argument then an out-of-range exception will occur.
atom_chars(Atom, AtomList)

AtomList is the list of single character atoms corresponding to the successive characters of Atom.
mode atom_chars(@atom, ?closed_list(atom))
mode atom_chars(-atom, @closed_list(atom))

Example:

| ?- atom_chars(abc, Chars).  
  Chars = [a, b, c]  
| ?- atom_chars(Atom,[a,b,c]).  
  Atom = abc

atom_codes(Atom, CharCodeList)

CharCodeList is the list of character codes corresponding to the successive characters of Atom.
mode atom_codes(@atom, ?closed_list(integer))
mode atom_codes(-atom, @closed_list(integer))

Example:

| ?- atom_codes(atom, Chars).  
  Chars = [97, 116, 111, 109]  
| ?- atom_codes(Atom, [97, 116, 111, 109]).  
  Atom = atom  
| ?- atom_codes(Atom,"atom").
  Atom = atom

name(Atom, CharCodeList)

Atom is made of the character codes in CharCodeList.
mode name(@atomic, ?closed_list(integer))
mode name(-atomic, @closed_list(integer))

Example:

| ?- name(atom, Chars).  
  Chars = [97, 116, 111, 109]  
| ?- name(Atom, [97, 116, 111, 109]).
  Atom = atom  
| ?- name(Atom,"atom").
  Atom = atom

char_code(Atom, CharCode)

CharCode is the character code for the single character atom Atom.
mode char_code(@atom, ?integer)
mode char_code(-atom, @integer)

Example:
| ?- char_code(a, Code).
Code = 97
| ?- char_code(Char, 97).
Char = a

number_chars(Integer, AtomList)

AtomList is the list of characters corresponding to the successive characters of Integer.

mode number_chars(@integer, ?closed_list(atom))
mode number_chars(-integer, @closed_list(atom))

Example:

| ?- number_chars(42, Chars).
Chars = [4, 2]

number_codes(Integer, CharCodeList)

CharCodeList is the list of character codes corresponding to the successive characters of Integer.

mode number_codes(@integer, ?closed_list(integer))
mode number_codes(-integer, @closed_list(integer))

Example:

| ?- number_codes(42, Codes).
Codes = [52, 50]
| ?- number_codes(Num, [52, 50]).
Num = 42

atom_concat(Atom1, Atom2, Atom3)

Atom3 is the atom formed by concatenating the characters of Atom1 and the characters of Atom2.

mode atom_concat(@atom, @atom, -atom)
mode atom_concat(?atom, ?atom, @atom)

Example:

| ?- atom_concat(ab, cd, Atom).
Atom = abcd
| ?- atom_concat(Atom1, Atom2, ab).
Atom1 = % the empty atom
Atom2 = ab;
Atom1 = a
Atom2 = b;
Atom1 = ab
Atom2 = ;
no
atom_concat2(Atom1, Atom2, Atom3)

Atom3 is the atom formed by concatenating the characters of Atom1 and the characters of Atom2. This is a faster version of atom_concat because of its more restricted mode.

mode atom_concat2(@atom, @atom, ?atom)

atom_length(Atom, N)

Length is the number of characters in Atom.

mode atom_length(@atom, ?integer)

Example:

| ?- atom_length(abc, Len).
Len = 3

atom_search(Atom1, Location1, Atom2, Location2)

Location2 is the position in Atom1 for the first occurrence of Atom2 with the search starting at Location1 in Atom1.

mode atom_search(@atom, @integer, @atom, ?atom)

Example:

| ?- atom_search(abab, 1, b, N).
N = 2

string_to_list(String, List)

List is the ASCII list corresponding to String.

mode string_to_list(@string, ?list(integer))
mode string_to_list(-string, @list(integer))

Example:

| ?- string_to_list("ab", L).
L = [97, 98]
| ?- string_to_list(S, [97,98]).
S = "ab"

string_to_atom(String, Atom)

Atom is the atom whose name is String.

mode string_to_atom(@string, ?atom)
mode string_to_atom(-string, @atom)

Example:
| ?- string_to_atom("ab", A).
A = ab
| ?- string_to_atom(S, ab).
S = "ab"

string_concat(String1, String2, String3)
String3 is the string formed by concatenating the characters of String1 and the characters of String2.
mode string_concat(@string, @string, -string)
mode string_concat(?string, ?string, @string)
Example:

| ?- string_concat("ab", "cd", String).
String = "abcd"
| ?- string_concat(String1, String2, "ab").
String1 = []
String2 = "ab";
String1 = "a"
String2 = "b";
String1 = "ab"
String2 = [];
no

string_length(String, N)
Length is the number of characters in String.
mode string_length(@string, ?integer)
Example:

| ?- string_length("abc", Len).
Len = 3

sub_string(String, Start, Length, After, SubString)
SubString is the substring of String starting at position Start and of length Length. After is the number of characters remaining in String after the end of SubString.
mode sub_string(@string, ?integer, ?integer, ?integer, ?string)
Example:

| ?- sub_string("ab", S, L, A, SS).
S = 0
L = 0
A = 2
re_match(REString, String, Match)

Match is a list of index ranges representing a match of the regular expression REString in the string String. On backtracking all matches will be found.

See http://www.pcre.org/current/doc/html/pcre2syntax.html for details of the syntax of the regular expression library PCRE2. NOTE: the pcre library needs to be installed before building QuProlog for this predicate to be accessible.

Example:

?- re_match("(\d*)(\s*)", "12 3 45", R).
R = [0 : 5, 0 : 2, 2 : 5];
R = [5 : 7, 5 : 6, 6 : 7];
R = [7 : 9, 7 : 9, 9 : 9];
R = [9 : 9, 9 : 9, 9 : 9];

Note that the backslash of the regular expression needs to be escaped. For the first answer, 0 is the index of the beginning of the match and 5 is the index of the end of the overall match. The next two ranges give the matches for the bracketed sub-RE: 0:2 matches the digits and 2:5 matches the spaces.
The range values in combination with the use of `sub_string` can be used to extract the required substring.

**quantify(Quantified, Quantifier, VarList, Body)**

Quantified is a quantified term composed of Quantifier, VarList, and Body.

mode quantify(?quant, ?term, ?list(term), ?term)

Example:

```
| ?- quantify(!!integral(0, !x) !x f(!x), Quant, Var, Body).
x = !x
Quant = integral(0, !x)
Var = [!x]
Body = f(!x)
| ?- quantify(QT, lambda, Var, Body).
QT = !!lambda Var Body
Var = Var
Body = Body
providied:
   check_binder(Var, [])
```

**quantifier(Quantified, Quantifier)**

The quantifier of Quantified is Quantifier.

mode quantifier(+quant, ?term)

Example:

```
| ?- quantifier(!!integral(0, !x) !x f(!x), Quant).
x = !x
Quant = integral(0, !x)
```

**bound_var(Quantified, VarList)**

VarList is the bound variable list of the quantified term Quantified.

mode bound_var(+quant, ?list(term))

Example.

```
| ?- bound_var(!!q [!x:t, !y] f(A), Var).
x = !x
y = !y
A = A
Var = [!x : t, !y]
providied:
   !y not_free_in [!x]
   !x not_free_in [!y]
```
body(Quantified, Term)

Term is the body of the quantified term Quantified.

mode body(+quant, ?term)

Example.

| ?- body(!!q !x f(A), Body).
x = !x
A = A
Body = f(A)

collect_vars(Term, VarList)

VarList is a list of all the variables in Term.

mode collect_vars(@term, -closed_list(anyvar))

Example.

| ?- collect_vars(f(X, _Y, !x, !_y), VarList), write(VarList), nl, fail.
[!_x0, !x, _1C3, X]
no

concat_atom(AtomList, Atom)

concat_atom(AtomList, Atom1, Atom2)

Atom2 is the atom formed by concatenating the characters of all the atom-
ics in AtomList with Atom1 interleaving the atoms.

mode concat_atom(@closed_list(atomic), @atomic, ?atom)

Example.

| ?- concat_atom([a, b, c], Atom).
Atom = abc

| ?- concat_atom([a,b,c],'/', Atom).
Atom = a/b/c

copy_term(Term1, Term2)

Term2 is a copy of Term1 with all the variables replaced by fresh variables.

mode copy_term(@term, ?term)

Example:

| ?- copy_term(f(X,g(X,Y),Y), Term2).
X = X
Y = Y
Term2 = f(B, g(B, A), A)
get_distinct(ObVar, DistinctList)

DistinctList is the list of object variables known to be different from ObVar.
mode get_distinct(@obvar, ?closed_list(obvar))

Example.

| ?- !x not_free_in f(!y, !z), get_distinct(!x, List).
  x = !x
  y = !y
  z = !z
  List = [!z, !y]
provided:
  !z not_free_in [!x]
  !y not_free_in [!x]
  !x not_free_in [!z, !y]

parallel_sub(TermList, ObVarList, SubList)

SubList is a list representing a single parallel substitution whose range elements are in TermList and whose domain elements are in ObVarList.
mode parallel_sub(@closed_list(term), @closed_list(obvar),
  -closed_list(compound))
mode parallel_sub(?closed_list(term), ?closed_list(obvar),
  +closed_list(compound))

Example:

| ?- parallel_sub([a, f(X)], [!x, !y], SubList).
  X = X
  x = !x
  y = !y
  SubList = [a / !x, f(X) / !y]
| ?- parallel_sub(TermList, ObVarList, [a / !x, f(X) / !y]).
  TermList = [a, f(X)]
  ObVarList = [!x, !y]
  x = !x
  X = X
  y = !y

simplify_term(Term1, Term2)
simplify_term(Term1, Term2, Atom)

Term2 is the result of applying simplification to any substitution in Term1. Atom is set to true if any simplification is performed. Otherwise Atom is set to fail. Note that the Qu-Prolog interpreter calls simplify_term before printing answers.
mode simplify_term(@term, ?term)
mode simplify_term(@term, ?term, -atom)

Example:

| ?- simplify_term([a/!x]f(!x), Term2), write(Term2), nl, fail. f(a) no |
| ?- simplify_term([a/!x]f(Var), Term2, Atom), write(Term2), nl, write(Atom), nl, fail. f([a/!x]Var) true no |
| ?- simplify_term([a/!x][b/!y]f(X, !x, !y), Term2), write(Term2), nl, fail. f([a/!x, b/!y]X, [a/!x, b/!y]!x, b) no |
| ?- simplify_term([a/!x, B/!y]f(X, !x, !y), Term2), write(Term2), nl, fail. f([a/!x, B/!y]X, [a/!x, B/!y]!x, B) no |
| ?- simplify_term([a/!x]!!q !x B, Term2), write(Term2), nl, fail. !!q !x B no |
| ?- simplify_term([a/!y]!!q !x B, Term2), write(Term2), nl, fail. [a/!y](!!q !x B) no |

sub_atom(Atom1, Location, N, Atom2)

Atom2 is an atom which has N characters identical to the N characters of Atom1 starting at Location in Atom1.
mode sub_atom(@atom, ?integer, ?integer, ?atom)

Example:

| ?- sub_atom(atom1, 2, 2, Atom2). Atom2 = to |
| ?- sub_atom(atom1, Loc, 2, Atom2). Loc = 1 Atom2 = at; Loc = 2 Atom2 = to; Loc = 3 Atom2 = om; Loc = 4 |
Atom2 = m1;
no
| ?- sub_atom(atat, Loc, Num, at).
Loc = 1
Num = 2;
Loc = 3
Num = 2;
no

sub_term(Substituted, Term)

Term is Substituted with the substitution removed.

mode sub_term(+term, ?term)

Example:

| ?- sub_term([a/!x]f(A), Term).
x = x
A = A
Term = f(A)

substitution(Substituted, Substitutions)

Substitutions is the list of parallel substitutions appearing at the top-level of Substituted.

mode substitution(+term, -closed_list(closed_list(compound)))

Example:

| ?- substitution([x/y, z/x]f(X, g(Y), [y/z]Z), Subs).
y = !y
x = !x
X = X
Y = Y
z = !z
Z = Z
Subs = [[x / !y, z / !x]]

substitute(Substituted, Substitutions, Term)

Substituted has Substitutions applied to Term. See substitution/2 (page 71).

mode substitute(+term, ?closed_list(closed_list(compound)), ?term)
mode substitute(-term, @closed_list(closed_list(compound)), @term)

Example:

| ?- substitute(T, [[a/x1, b/x2], [c/x3]], f(X)), write(T), fail.
[a/x1, b/x2][c/x3]f(X)
no
uncurry(HigherGoal, Goal)

Flatten a HigherGoal to a normal Goal.
mode uncurry(+term, ?term)

Example:

\[ X = f(a, b) \]

3.8 List Processing

This set of predicates provides some frequently used list operations. The set can be divided into two parts according to the type of list on which the predicates operate. The predicates which manipulate open lists (i.e. lists terminated with a variable) have a `open_` prefix in the name. Otherwise, closed (proper) lists are assumed. When `==/2` (page 50) is used for comparison instead of unification, the predicate name is suffixed with an `_eq`.

Predicates:

`open_list(Term)`

Succeed if Term is an open list.
mode `open_list(@term)`

Example:

\[
| ?- open_list([a, b, c]).
| \text{no}
| ?- open_list([a, b, c | Tail]).
| \text{Tail = Tail}
\]

`closed_list(Term)`

Succeed if Term is a closed (proper) list.
mode `closed_list(@term)`

Example:

\[
| ?- closed_list([a, b, c]).
| \text{yes}
| ?- closed_list([a, b, c | Tail]).
| \text{no}
\]

`closed_to_open(Closed, Open)`

Convert a Closed list to an Open list by appending an unbound variable.
mode `closed_to_open(+closed_list(term), ?list(term))`

Example:
| ?- Closed = [a, b, c], closed_to_open([a, b, c], Open), write(Open), nl, fail.
[a, b, c | Tail]
no

open_to_closed(List)
Convert List from an open list to a closed list by binding the tail to [].
mode open_to_closed(?closed_list(term))
Example:
| ?- Open = [a, b, c | Tail], open_to_closed(Open), write(Open), nl, fail.
[a, b, c]
no

delete(Term, List1, List2)
List2 is List1 after deleting an instance of Term. The comparison is performed by unification.
mode delete(+term, +closed_list(term), ?closed_list(term))
Example:
| ?- delete(a, [a, b, d, a, c], List2).
List2 = [b, d, a, c];
List2 = [a, b, d, c];
no

delete_all(Term, List1, List2)
List2 is List1 after deleting all the instances of Term. The comparison is performed by unification.
mode delete_all(+term, +closed_list(term), ?closed_list(term))
Example:
| ?- delete_all(a, [a, b, d, a, c], List2).
List2 = [b, d, c];
no

intersect_list(List1, List2, List3)
List3 contains all the elements which appear in both List1 and List2. The comparison is performed by ==/2.
mode intersect_list(@closed_list(term), @closed_list(term), -closed_list(term))
Example:
intersection_list([a, b, d, a, c], [c, b, e, a, f], List3).
List3 = [a, b, a, c]

intersection_list([a, B, d, a, C], [C, b, e, a, f], List3).
B = B
C = C
List3 = [a, a, C]

union_list(List1, List2, List3)
List3 contains all the elements which appear in either List1 or List2.
The comparison is performed by ==/2.
mode union_list(+closed_list(term), +closed_list(term), ?closed_list(term))

Example:

?- union_list([a, b, d, a, c], [c, b, e, a, f], List3).
List3 = [d, c, b, e, a, f]

?- union_list([a, B, d, a, C], [C, b, e, a, f], List3).
B = B
C = C
List3 = [B, d, C, b, e, a, f]

diff_list(List1, List2, List3)
List3 contains all those elements in List1 but not in List2. The comparison is performed by ==/2.
mode diff_list(+closed_list(Term), +closed_list(Term), ?closed_list(Term))

Example:

?- diff_list([a, b, d, a, c], [c, b, e, a, f], List3).
List3 = [d]

?- diff_list([a, B, d, a, C], [C, b, e, a, f], List3).
B = B
C = C
List3 = [B, d]

distribute(Term, List1, List2)
List2 is a list of pairs (Term, X) for each X in List1.
mode distribute(+term, +closed_list(term), ?closed_list(term))
mode distribute(?term, ?closed_list(term), +closed_list(term))

Example:
distribute(X, [a, b, c], List2).
X = X
List2 = [(X, a), (X, b), (X, c)]

?- distribute(Term, List1, [(A, 1), (A, 2), (A, 3)]).
Term = A
List1 = [1, 2, 3]
A = A

?- distribute(Term, List1, [(A, 1), (B, 2), (C, 3)]).
Term = C
List1 = [1, 2, 3]
A = C
B = C
C = C

distribute_left(Term, List1, List2)
List2 is a list of pairs (Term, X) for each X in List1.
mode distribute_left(+term, +closed_list(term), ?closed_list(term))
mode distribute_left(?term, ?closed_list(term), +closed_list(term))

Example:

?- distribute_left(f(G), [a, b, c], List2).
G = G
List2 = [(f(G), a), (f(G), b), (f(G), c)]

distribute_right(Term, List1, List2)
List2 is a list of pairs (X, Term) for each X in List1.
mode distribute_right(+term, +closed_list(term), ?closed_list(term))
mode distribute_right(?term, ?closed_list(term), +closed_list(term))

Example:

?- distribute_right(term, [a, b, c], List2).
List2 = [(a, term), (b, term), (c, term)]

append(List1, List2, List3)
List3 is List1 appended to List2.
mode append(+closed_list(term), ?list(term), ?list(term))
mode append(?list(term), ?list(term), +closed_list(term))

Example:
| ?- append([a, b, c], [b, c, d], List3).
List3 = [a, b, c, b, c, d]
| ?- append([a, b, c], List2, List3).
List2 = List2
List3 = [a, b, c|List2]
| ?- append(List1, List2, [a, b, c, d]).
List1 = []
List2 = [a, b, c, d];
List1 = [a]
List2 = [b, c, d];
List1 = [a, b]
List2 = [c, d];
List1 = [a, b, c]
List2 = [d];
List1 = [a, b, c, d]
List2 = [];
no

length(List, N)
The length of List is N.
mode length(@closed_list(term), ?integer)
mode length(-closed_list(term), @integer)
Example:
| ?- length([a, b, c], Length).
Length = 3
| ?- length(List, 3).
List = [A, B, C]
| ?- length([a, b, c |Rest], 5).
Rest = [A, B]
| ?- length(List, Length).
List = []
Length = 0;
List = [A]
Length = 1;
List = [A, B]
Length = 2;
List = [A, B, C]
Length = 3

member(Term, List)
Term is an element of List. The comparison is performed by unification.
mode member(?term, ?list(term))
Example:
| ?- member(b, [a, b, c]).
yes
| ?- member(A, [a, b, c]).
A = a;
A = b;
A = c;
no
| ?- member(Term, List).
List = [Term|A];
List = [A, Term|B];
List = [A, B, Term|C];
List = [A, B, C, Term|D]

member_eq(Term, List)
    Term is an element of List. The comparison is performed by ==/2.
mode member_eq(@term, @closed_list(term))
Example:
| ?- member_eq(Term, [a, b, c]).
no
| ?- member_eq(b, [a, b, c]).
yes

open_append(Open, List)
    Append an open list, Open, and a closed list, Closed, to produce a new
    open list by instantiating the tail of Open.
mode open_append(+open_list(term), @closed_list(term))
Example:
| ?- List = [1, 2 | Tail], open_append(List, [3, 4]).
List = [1, 2, 3, 4|A]
Tail = [3, 4|A]

open_length(List, N)
    The length of the open List is N, ignoring the tail.
mode open_length(@open_list(term), -integer)
mode open_length(?open_list(term), @integer)
Example:
| ?- open_length([a, b, c | Tail], Length).
Tail = Tail
Length = 3
| ?- open_length(List, 5).
List = [A, B, C, D, E|F]
open_member(Term, Open)

Term is an element of Open. The comparison is performed by unification with existing elements only.

mode open_member(+term, +open_list(term))

Example:

| ?- open_member(b, [a, b, c | Tail]).
Tail = Tail
| ?- open_member(Term, [a, b, c | Tail]).
Term = a
Tail = Tail;
Term = b
Tail = Tail;
Term = c
Tail = Tail;
no
| ?- open_member(Term, List).
no

open_member_eq(Term, Open)

Term is an element of Open. The comparison is performed by ==/2.

mode open_member_eq(@term, @open_list(term))

Example:

| ?- open_member_eq(Term, [a, b, c | Tail]).
no
| ?- open_member_eq(b, [a, b, c | Tail]).
Rest = Rest

open_tail(Open, Variable)

Variable is the tail of the open list, Open.

mode open_tail(+open_list(term), -var)

Example:

| ?- open_tail([a, b, c | Tail], Var).
Tail = Var
Var = Var

remove_duplicates(List1, List2)
List2 is List1 without duplicates.
mode remove_duplicates(@closed_list(term), ?closed_list(term))
Example:

| ?- remove_duplicates([a, b, d, a, c], List2).
List2 = [b, d, a, c]
| ?- remove_duplicates([a, B, c, a, B, C], List2).
B = B
C = C
List2 = [c, a, B, C]

reverse(List1, List2)
List2 is the reverse of List1. The comparison is performed by ==/2 (page 50).
mode reverse(+closed_list(term), ?closed_list(term))
Example:

| ?- reverse([a, b, d, a, c], List2).
List2 = [c, a, d, b, a]

search_insert(Term, Open)
Search through Open for Term. If not found, insert Term. The comparison is performed by ==/2 (page 50).
mode search_insert(@term, +open_list(term))
Example:

| ?- List = [a, b, d, a, c | Tail], search_insert(d, List).
List = [a, b, d, a, c|Tail]
Tail = Tail
| ?- List = [a, b, d, a, c | Tail], search_insert(f, List).
List = [a, b, d, a, c, f|A]
Tail = [f|A]
| ?- List = [a, b, D, a, c | Tail], search_insert(D, List).
List = [a, b, D, a, c|Tail]
D = D
Tail = Tail
| ?- List = [a, b, D, a, c | Tail], search_insert(E, List).
List = [a, b, D, a, c, E|A]
D = D
Tail = [E|A]
E = E

sort(List1, List2)
List2 is the result of sorting List1 into the standard order with duplicates removed.

mode sort(@closed_list(term), -closed_list(term))

Example:

| ?- sort([a, b, d, a, c], List2).
List2 = [a, b, c, d]

sort(List1, List2, Order)

List2 is the result of sorting List1 into the order defined by the predicate Order with duplicates removed. So

sort(List1, List2) is the same as sort(List1, List2, '0<')

mode sort(@closed_list(term), -closed_list(term), @goal)

msort(List1, List2, Order), msort(List1, List2, Order)

The same as the above sorts but with duplicates not removed.

3.9 All Solutions

All the solutions for Goal are collected into List according to the Template format. The solutions are obtained via backtracking. For the examples given for each predicate below, it is assumed that the following predicate definition has been added to the system.

p(a, 1). p(a, 2). p(c, 1). p(b, 6). p(c, 4).

Predicates:

Variable^Goal

Execute Goal. It is treated the same as call(Goal). It is expected to be used only inside bagof/3 (page 80), findall/3 (page 81), and setof/3 (page 81), where it represents existential quantification.

mode +var ^ +goal

bagof(Template, Goal, List)

List is the collection of instances of Template, which satisfy the Goal.

mode bagof(@term, @goal, ?closed_list(term))

Example:

| ?- bagof(X, p(X,Y), R).
X = X
Y = 1
R = [a, c];
X = X

80
Y = 2
R = [a];
X = X
Y = 4
R = [c];
X = X
Y = 6
R = [b];
no
| ?- bagof(X, Y^p(X,Y), R).
X = X
Y = Y
R = [a, a, c, b, c]

setof(Template, Goal, List)

List is the collection of instances of Template, which satisfy the Goal. Duplicates are removed and the result is sorted in the standard order.

mode setof(\@term, \@goal, \@closed_list(term))

Example:

| ?- setof(X, p(X,Y), R).
X = X
Y = 1
R = [a, c];
X = X
Y = 2
R = [a];
X = X
Y = 4
R = [c];
X = X
Y = 6
R = [b];
no
| ?- setof(X, Y^p(X,Y), R).
X = X
Y = Y
R = [a, b, c]

findall(Template, Goal, List)

List is the collection of instances of Template, which satisfy the Goal. Unlike bagof/3 (page 80), there is an implied existential quantification of all variables not in Template.

mode findall(\@term, \@goal, \@closed_list(term))

Example:
forall(Goal, Test)

This predicate succeeds (without binding variables) iff whenever Goal succeeds, Test also succeeds.
mode forall(@goal, @goal)

Example:

| ?- forall(member(X, [1,2,3]), integer(X)).
  X = X
| ?- forall(member(X, [5,4,1,3,6]), X > 2).
  no

3.10 Arithmetic

These predicates perform arithmetical operations on the arguments, which are arithmetic expressions. Each expression can be a mixture of numbers, variables, and arithmetic functions. The expression must be free of unbound variables when it is evaluated.

Predicates:

Expression1 =:= Expression2

Expression1 is numerically equal to Expression2.
mode @ground =:= @ground

Term1 \neq Term2

Term1 and Term2 are not numerically equal.
mode @ground \neq @ground

Expression1 < Expression2

Expression1 is numerically less than Expression2.
mode @ground < @ground

Expression1 =< Expression2

Expression1 is numerically less than or equal to Expression2.
mode @ground =< @ground
Expression1 > Expression2
Expression1 is numerically greater than Expression2.
mode @ground > @ground

Expression1 >= Expression2
Expression1 is numerically greater than or equal to Expression2.
mode @ground >= @ground

Example:

?- 12 =:= 3 * 4.
?- 12 =\= 14.
?- -3 < 3.
?- 3+4 =< 12.
?- A = 12, A > 8.
?- A = 12, B = -2, A >= B.

between(Integer1, Integer2, N)
Integer1 <= N <= Integer2. This predicate can generate N, and may be used to provide a "for loop" like iteration driven by failure.
mode between(@integer, @integer, ?integer)

Example:

?- between(1, 3, A).
A = 1;
A = 2;
A = 3;
no

is(Value, Expression)
The result of evaulating Expression unifies with Value. The following operators are available.

• +
  Addition
• -
  Subtraction/Negation
• *
  Multiplication
• //
  Integer division
• /
  Division
• **
  Power
• >>
  Bitwise right shift. Effectively divides by two and rounds towards negative infinity.
• <<
  Bitwise left shift. Effectively multiplies by two.
• /
  Bitwise AND
• \/
  Bitwise OR
• \n
  Bitwise complement
• abs
  Absolute value
• sqrt
  Square root
• exp
  Exponentiation
• sin
  Sin
• cos
  Cos
• tan
  Tan
• asin
  Asin
• acos
  Acos
• atan
  Atan
• round
  Round to nearest integer
• floor
  Floor
• ceiling
  Ceiling truncate
  Truncate

The atoms \(\pi\) and \(e\) are also available for use in arithmetic expressions.

\(\text{is}\) is an infix operator.

\text{mode} \ is(?number, @gcomp)
\text{mode} \ is(?number, @number)

Example:

\|-- ?- A is 3 + 5.
   A = 8
\|-- ?- A is 7 / 4.
   A = 1
\|-- ?- A is 9 // 2.
   A = 4
\|-- ?- A is 7 mod 4.
   A = 3
\|-- ?- A is 2'1010 /\ 2'1001.
   A = 8
\|-- ?- A is 10 \// 9.
   A = 11
\|-- ?- A is \pi*3.5**2.
   A = 38.4845

The following predicates provide uniformly distributed pseudo-random numbers.

Predicates:

\text{srandom(Seed)}

Initialize the random numbers with a seed. If \text{Seed} is supplied then that seed is used for initialization. Otherwise, a seed is generated from the current time and that seed is used for initialization and \text{Seed} is instantiated to that value.

\text{mode} \ srandom(?integer)

\text{random(R)}

\text{R} is unified with a random double between 0 and 1.

\text{mode} \ random(?double)

\text{random(Lower, Upper, I)}

\text{I} is unified with a random integer in the range \([\text{Lower}, \text{Upper}]\).

\text{mode} \ random(@integer, @integer, ?integer)

\text{irandom(I)}

\text{I} is unified with a random integer.

\text{mode} \ irandom(?integer)
3.11 Term Expansion

When a file is consulted or compiled, every clause, fact and query within it is subject to term expansion. User term expansions are installed by defining clauses for term_expansion/2,3; the add_ and del_ predicates are provided to assist with this. Goal is a higher-order goal that will be called with two or three extra arguments: the input and output terms (Term1, Term2), and perhaps a list VariableNames of (Variable=Name) pairs for the input clause (as in read_term/[2,3] (page 38)). Definite Clause Grammar (DCG) expansions are carried out automatically after user expansions.

Multi-term expansion is defined using term expansion. When multi-term expansion is active, the current set of multi-expansions will be applied repeatedly to each clause until none succeeds. It is the user’s responsibility to avoid cycles; a depth limit helps identify these.

Subterm expansion is defined using term expansion. When subterm expansion is active, the current set of subterm expansions will be applied to every subterm of every clause until none succeeds. It is the user’s responsibility to avoid cycles; a depth limit helps identify these.

The output of term expansion and multi-term expansion (but not of subterm expansion) may be a single term or a list of terms.

Predicates:

- list_expansions
  List all current term expansions, multi-term expansions and subterm expansions.

- add_expansion(Goal)
  Add Goal to the definition of term_expansion/2. Goal will be called with two extra arguments: the input term and the output term (or list of terms).
  mode add_expansion(@goal)

- add_expansion_vars(Goal)
  Add Goal to the definition of term_expansion/3. Goal will be called with three extra arguments: the input term and the output term (or list of terms), and a list of Variable=Name pairs for the term being expanded.
  mode add_expansion_vars(@goal)

- add_multi_expansion(Goal)
  Add Goal to the current set of multi-term expansions. Also add multi_expand_term/2 (page 89) to the definition of term_expansion/2 if necessary. Goal will be called with two extra arguments: the input term and the output term (or list of terms).
  mode add_multi_expansion(@goal)
add_multi_expansion_vars(Goal)

Add Goal to the current set of multi-term expansions with variables and add multi_expand_term/3 (page 89) to the definition of term_expansion/3 if necessary. Goal will be called with three extra arguments: the input term and the output term (or list of terms), and a list of Variable=Name pairs for the term being expanded.

mode add_multi_expansion_vars(@goal)

add_subterm_expansion(Goal)

Add Goal to the current set of subterm expansions. Also add expand_subterms/2 (page 89) to the definition of term_expansion/2 if necessary. Goal will be called with two extra arguments: the input subterm and the output subterm.

mode add_subterm_expansion(@goal)

Examples (See end of section):

```prolog
| ?- assert(macro_eg(append3(A,B,C,D), (append(A,X,D), append(B,C,X)))).
| ?- add_subterm_expansion(macro_eg).
yes
| ?- list_expansions.
term_expansion(B, A) :-
    macro_eg(B, A).
yes
```

add_subterm_expansion_vars(Goal)

Add Goal to the current set of subterm expansions with variables, and add expand_subterms/3 (page 89) to the definition of term_expansion/3 if necessary. Goal will be called with three extra arguments: the input subterm and the output subterm, and a list of Variable=Name pairs for the term being expanded.

mode add_subterm_expansion_vars(@goal)

del_expansion(Goal)

Delete Goal from the definition of term_expansion/2.

mode del_expansion(@goal)

del_expansion_vars(Goal)

Delete Goal from the definition of term_expansion/3.

mode del_expansion_vars(@goal)

del_multi_expansion(Goal)
Delete Goal from the current set of multi-term expansions, and delete multi Expand Term/2 (page 89) from the definition of term Expansion/2 if it is no longer necessary.
mode del_multi_expansion(@goal)

del_multi_expansion_vars(Goal)
Delete Goal from the current set of multi-term expansions with variables, and delete multi expand term/3 (page 89) from the definition of term expansion/3 if it is no longer necessary.
mode del_multi_expansion_vars(@goal)

del_subterm_expansion(Goal)
Delete Goal from the current set of subterm expansions, and delete expand_subterms/2 (page 89) from the definition of term expansion/2 if it is no longer necessary.
mode del_subterm_expansion(@goal)

del_subterm_expansion_vars(Goal)
Delete Goal from the current set of subterm expansions with variables, and delete expand_subterms/3 (page 89) from the definition of term expansion/3 if it is no longer necessary.
mode del_subterm_expansion_vars(@goal)

'C'(List1, Term, List2)
List1 is connected by Term to List2.
This predicate typically appears only in preprocessed DCG rules.
mode 'C'(?list(term), ?term, ?list(term))
Example:
| ?- 'C'([a,b,c],Y,Z).
  Y = a
  Z = [b, c]
dcg(Rule, Clause)
Clause is the expansion of Definite Clause Grammars Rule.
mode dcg(@nonvar, ?nonvar)
Examples (See end of section):
| ?- dcg((pairing(P1 + P2) --> filler, pair(P1), pairing(P2)), X),
  portray_clause(X),fail.
  pairing(P1 + P2, D, A) :- 
    filler(D, C),
    pair(P1, C, B),
    pairing(P2, B, A).
expand_term(Term1, Term2)
expand_term(Term1, Term2, VariableNames)

Term2 is the result of applying term expansions to Term1. VariableNames is a list of variables and their names.
mode expand_term(+term, ?term)
mode expand_term(+term, ?term, ?closed_list(compound))

multi expand_term(Term1, Term2)
multi expand_term(Term1, Term2, VariableNames)

Term2 is the result of applying multi-term expansions to Term1. VariableNames is a list of variables and their names.
mode multi expand_term(+term, ?term)
mode multi expand_term(+term, ?term, ?closed_list(compound))

expand_subterms(Term1, Term2)
expand_subterms(Term1, Term2, Vars)

Term2 is the result of applying subterm expansions to Term1. VariableNames is a list of variables and their names.
mode expand_subterms(+term, ?term)
mode expand_subterms(+term, ?term, ?closed_list(compound))

multi expand_depth_limit(Limit)

The depth limit for multi-term expansion is Limit. Multi-term expansion will stop and a warning (range_exception) will be generated if this limit is exceeded in any one term. This built-in can be used to query the current limit or set a new one. The default depth is 100.
mode multi expand_depth_limit(?integer)

subterm expand_depth_limit(Limit)

The depth limit for subterm expansion is Limit. Subterm expansion will stop and a warning (range_exception) will be generated if this limit is exceeded in any one subterm. This built-in can be used to query the current limit or set a new one. The default depth is 100.
mode subterm expand_depth_limit(?integer)

phrase(Rule, List1)
phrase(Rule, List1, List2)

List1 is parsed according to the DCG Rule. List2 is the remaining symbols from List1 after the parsing.
mode phrase(+nonvar, +closed_list(term))
mode phrase(+nonvar, +closed_list(term), ?closed_list(term))

Examples (See end of section):
The following example uses subterm expansion to achieve macro expansion. One way to append three lists together is to write and use a predicate that appends the lists. Another way is to think of \texttt{append3/4} as a macro for a pair of appends. Term expansion can then be used to expand occurrences of \texttt{append3/4} within programs.

This can be achieved by first declaring the term expansion rule in a file (say \texttt{expand.ql}) as follows.

\[
\text{macro_eg}(\text{append3}(A,B,C,D), (\text{append}(A,X,D), \text{append}(B,C,X))).
\]

\[
?-\text{add_subterm_expansion(macro_eg)}.
\]

This file can then be supplied to the Qu-Prolog compiler to term expand supplied programs. For example, assume the file \texttt{expand_eg.ql} contains the following definition.

\[
p(Term, Result) :-
    \text{extract_lists}(Term, L1, L2, L3),
    \text{append3}(L1, L2, L3, L),
    \text{process_lists}(L, Result).
\]

The compiler can then be invoked to carry out term expansion as in the following example.

\[
qc ~-G ~-R \text{expand} \text{expand_eg.ql}
\]

The ~-R switch tells the compiler to use \texttt{expand.ql} (or \texttt{expand.qo} if it is compiled) for the term expansion rules and the ~-G switch tells the compiler to stop after term expansion. The result is that a file \texttt{expand_eg.qg} will be generated that is an encoded and term expanded version of \texttt{expand_eg.ql}.

\[
p(Term, Result) :-
    \text{extract_lists}(Term, L1, L2, L3),
    \text{append}(L1, A, L),
    \text{append}(L2, L3, A),
    \text{process_lists}(L, Result).
\]
The example given below uses a Definite Clause Grammar to determine the bracket pairing structure of a string of characters.

The following rules define the grammar.

pairing(P1 + P2) --> filler, pair(P1), pairing(P2).
pairing(P) --> filler, pair(P), filler.
pair(pair) --> [0'|], filler, [0'|].
pair(pair(P)) --> [0'|], pairing(P), [0'|].
filler --> [X], {X \= 0', X \= 0'}, filler.
filler --> [].

When this grammar is compiled or consulted the DCG expansion will transform the grammar rules into the following Qu-Prolog rules. Note that Qu-Prolog code may appear within grammar rules as long as it is enclosed by parentheses (for example \(\{X \neq 0', X \neq 0'\}\)) and in this case the DCG expansion will leave this code untouched.

pairing(D + C, B, A) :-
    filler(B, E),
    pair(D, E, F),
    pairing(C, F, A),
    true.

pairing(C, B, A) :-
    filler(B, D),
    pair(C, D, E),
    filler(E, A),
    true.

filler(B, A) :-
    'C'(B, D, C),
    D \= 40,
    D \= 41,
    filler(C, A),
    true.

filler(A, A) :-
    true,
    true.

pair(pair, B, A) :-
    'C'(B, 40, C),
    filler(C, D),
    'C'(D, 41, A),
    true.

pair(pair(C), B, A) :-
    'C'(B, 40, D),
    pairing(C, D, E),
    'C'(E, 41, A),

91
true.

Once this grammar is consulted or loaded then \texttt{phrase/2} (page 89) may be used to parse strings as in the following example.

\begin{verbatim}
| ?- phrase(pairing(T), "a(b(c()))(e)f()g").
T = pair(pair(pair) + pair) + pair
\end{verbatim}

3.12 Database

The system includes three non-backtrackable databases: the dynamic database, the record database, and the global state database.

3.12.1 Dynamic Database

Changes to the dynamic database can be achieved with the following predicates. The dynamic database is composed entirely of interpreted clauses. Clauses can be added to (asserted) or removed from (retracted) the dynamic database. The predicates cannot be applied to compiled clauses.

Clause indexing information is stored in a dynamic hash table that grows as more clauses are added. Consequently, if a dynamic predicate has many clauses then it is more efficient to declare the expected size of the predicate so that extending and rehashing is avoided. This is done with the \texttt{dynamic} predicate.

Predicates:

\begin{verbatim}
dynamic(PredicateName/Arity)
dynamic(PredicateName/Arity, Index)
dynamic(PredicateName/Arity, Index, Size)
\end{verbatim}

\texttt{PredicateName} with \texttt{Arity} is declared to be an interpreted predicate, initially with no clauses.

The second argument determines the index argument - first argument indexing is the default.

The third argument declares the size of the indexing hash table for the predicate (default 4).

mode \texttt{dynamic(@compound)}
mode \texttt{dynamic(@compound,@integer)}
mode \texttt{dynamic(@compound,@integer,@integer)}

Example:

\begin{verbatim}
| ?- p(A, B).
no definition for p/2
no
\end{verbatim}
?- dynamic(p/2).
yes
?- p(A, B).
no

multifile(PredicateName/Arity)

Essentially the same as `dynamic(PredicateName/Arity)` and is to provide an approximation to multifile as used in other Prologs.

assert(Clause)

Add the Clause to the end of the current interpreted program.

mode assert(@term)

Example:

?- assert((p(X) :- q(X))).
X = X

asserta(Clause)
assertz(Clause)

Add the Clause to the beginning (end) of the current interpreted program.

Example:

?- asserta((p(X,Y) :- q(X,Z), r(Z,Y))).
X = X
Y = Y
Z = Z
?- assertz((p(X,Y) :- p(Y,X)), Ref).
X = X
Y = Y
Ref = 324747

get_name(Clause, PredicateName/Arity)

The head of Clause has PredicateName and Arity.

mode get_name(@goal, ?gcomp)

Example:

?- get_name((p(X,Y) :- q(X,Y)), Pred).
X = X
Y = Y
Pred = p / 2
get_predicate_timestamp(Predicates, StampedPredicates)

Predicates is a list of the form P/N and StampedPredicates is a list of the form P/N-Stamp. The stamp is a number representing the number of times this predicate has been modified and is useful for thread_wait_on_goal to prevent an unnecessary initial call of the goal. As with thread_wait_on_goal, the predicates can be prefixed with a + or a - and in which case the stamp will be for a timestamp for asserting or retracting.

mode get_predicate_timestamp(@list, -list)

update_predicate_timestamp(OldStampedPredicates, StampedPredicates)

OldStampedPredicates and StampedPredicates are lists of the form P/N-Stamps. This is like get_predicate_timestamp except that stamped predicates are used rather than predicates.

mode update_predicate_timestamp(@list, -list)

changed_predicates(StampedPredicates, SPredicates)

StampedPredicates is a list of the form P/N-Stamp and Predicates is a list of the form P/N. Predicates is the list of all predicates from StampedPredicates that have more recent timestamps than those given. Typically this is used in conjunction with get_predicate_timestamp/2 above to determine what predicates have changed since the last call to get_predicate_timestamp/2.

mode changed_predicates(@list, -list)

clause(Head, Body)

There is an existing clause with the given Head and Body.

mode clause(+goal, ?goal)

Examples (Continued from last):

| ?- clause((p(A,B)), Body).
| A = A
| B = B
| Body = q(A, C) , r(C, B);
| A = A
| B = B
| Body = p(B, A);
| no

listing

listing(PredicateList)

List all interpreted predicates or only those given in PredicateList.

Examples (Continued from last):

94
| ?- listing.
p(A) :-
    q(A).
p(B, A) :-
    q(B, C),
    r(C, A).
p(B, A) :-
p(A, B).
yes
| ?- listing(p/1).
p(A) :-
    q(A).
yes

retract(Clause)
Remove the first clause that matches Clause.
mode retract(+nonvar)
Examples (Continued from last):
| ?- retract((p(X) :- q(X))).
X = X

retractall(Head)
Remove all the clauses whose heads match Head.
mode retract(@goal)
Example: (Continued from last):
| ?- retractall((p(A,B))).
A = A
| ?- clause((p(A,B)), Body).
no

abolish(PredicateName/Arity)
Make PredicateName/Arity undefined. This does not work on compiled code.
mode abolish(@gcomp)
Example:
| ?- abolish(p/2).
yes
| ?- p(A, B).
no definition for p/2
no
add_linking_clause(HigherHead, HigherBody, Arity)

A collection of Arity new variables is added to the end of HigherHead and to the end of HigherBody to form Head and Body respectively. The clause Head :- Body is then added to the database.

mode add_linking_clause(@nonvar, @nonvar, @integer)

Example:

Linking clauses may be used for connecting data stored in the static code area with data stored in the dynamic database.

Assume that there is a table of facts compiled_facts/2 in the static code area and a table of facts dynamic_facts/2 in the dynamic database. These facts can be linked together by executing the following goals.

add_linking_clause(all_facts, compiled_facts, 2)
add_linking_clause(all_facts, dynamic_facts, 2)

This asserts the following rules to the dynamic database.

all_facts(X,Y) :- compiled_facts(X,Y).
all_facts(X,Y) :- dynamic_facts(X,Y).

The combined rules can then be queried via the all_facts/2 predicate.

get_linking_clause(HigherHead, HigherBody, Arity)

A collection of Arity new variables is added to the end of HigherHead and to the end of HigherBody to form Head and Body respectively. The clause Head :- Body is then retrieved from the database.

mode get_linking_clause(+goal, ?goal, @integer)

Examples (Continued from last):

| ?- get_linking_clause(all_facts, Body, 2).
  Body = compiled_facts;
  Body = dynamic_facts;
  no

del_linking_clause(HigherHead, HigherBody, Arity)

A collection of Arity new variables is added to the end of HigherHead and to the end of HigherBody to form Head and Body respectively. The clause Head :- Body is then deleted from the database.

mode del_linking_clause(@goal, @goal, @integer)

Examples (Continued from last):

| ?- del_linking_clause(all_facts, dynamic_facts, 2).
  yes
| ?- clause(all_facts(A,B), Body).
A = A
B = B
Body = compiled_facts(A, B);
no

index(PredicateName, Arity, N)

Index PredicateName/Arity with the N-th argument. The default is the first argument. This is used as a directive to the compiler to produce clause indexing for static code.
mode index(@atom, @integer, @integer)
Example:

| ?- index(all_facts, 2, 2).
yes

3.12.2 Record Database

The record database associates terms with atoms. Terms can be added to (recorda/3, recordz/3) or removed from (erase/1) the record database.

Predicates:
recorda(Atom, Term, Reference)
recordz(Atom, Term, Reference)

Record Term as the first (last) term associated with Atom in the record database. Reference is the reference to this entry.
mode recorda(@atom, @term, -integer)
mode recordz(@atom, @term, -integer)
Example:

| ?- recorda(colour, sky(blue), Ref).
Ref = 70
| ?- recorda(colour, sun(yellow), Ref).
Ref = 72
| ?- recordz(colour, grass(green), Ref).
Ref = 74
| ?- recordz(colour, ocean(blue), Ref).
Ref = 76

recorded(Atom, Term, Reference)

Lookup Term associated with Atom in the record database at Reference.
mode recorded(@atom, ?term, ?integer)
Examples (Continued from last):

97
instance(Reference, Term)

Unifies Term with the term at Reference in the record database. This call does not instantiate variables in the term at Reference.

mode instance(@integer, +term)

Examples (Continued from last):

| ?- instance(70, Term).
| Term = sky(blue)
| ?- instance(72, Object(Colour)).
| Object = sun
| Colour = yellow

erase(Reference)

Erase the record database entry at Reference.

mode erase(@integer)

Examples (Continued from last):

| ?- erase(74).
| yes
| ?- recorded(colour, Term, Ref).
| Term = sun(yellow)
| Ref = 72;
| Term = sky(blue)
| Ref = 70;
| Term = ocean(blue)
| Ref = 76;
| no

3.12.3 Global State Database

The global state database associates atoms/integers with atoms. It is designed to give efficient (non-backtrackable) access/update of atom/integer values and increment/decrement of integer values.
Predicates:

```prolog
global_state_set(Atom, AtomOrInt)
    The value of the global state associated with Atom is set to AtomOrInt.
mode global_state_set(@atom, @atom) mode global_state_set(@atom, @integer)
Example:
    |- global_state_set(year, 1999).
    yes

global_state_lookup(Atom, AtomOrInt)
    The current value of the global state associated with Atom is unified with AtomOrInt. Fails if the global state associated with Atom has not been given a value.
mode global_state_lookup(@atom, ?atom) mode global_state_lookup(@atom, ?integer)
Examples (Continued from last):
    |- global_state_lookup(year, Term).
    Term = 1999

global_state_increment(Atom, Integer)
    The integer stored in the global state associated with Atom is incremented and the new value is returned in Integer. Fails if the global state associated with Atom does not contain an integer.
mode global_state_increment(@atom, -integer)
Examples (Continued from last):
    |- global_state_increment(year, Value).
    Value = 2000

global_state_decrement(Atom, Integer)
    The integer stored in the global state associated with Atom is decremented and the new value is returned in Integer. Fails if the global state associated with Atom does not contain an integer.
mode global_state_decrement(@atom, -integer)
Examples (Continued from last):
    |- global_state_decrement(year, 1999).
    yes
```
3.12.4 Hash Table

The hash table is a table of terms (global to all threads), with two-level indexing: the first index being an atom and the second being an atomic. The hash table provides efficient (non-backtrackable) access/update of terms associated with index pairs.

Predicates:

hash_table_insert(Atom, Atomic, Term)

The term of the hash table associated with the index pair (Atom, Atomic) is set to Term.

mode hash_table_insert(@atom, @atomic, @term)

Example:

| ?- hash_table_insert(student, 1234567, info('Fred', 'Bloggs', 'BSc')). yes |

hash_table_lookup(Atom, Atomic, Term)

The current value indexed by (Atom, Atomic) in the hash table is unified with Term. Fails if the term associated with the index pair has not been given a value.

mode hash_table_lookup(@atom, @atomic, ?term)

Examples (Continued from last):

| ?- hash_table_lookup(student, 1234567, Term). Term = info(Fred, Bloggs, BSc) |

hash_table_remove(Atom, Atomic)

The current value (if any) indexed by (Atom, Atomic) in the hash table is removed.

mode hash_table_remove(@atom, @atomic)

Examples (Continued from last):

| ?- hash_table_remove(student, 1234567). yes |
| ?- hash_table_lookup(student, 1234567, Term). no |

hash_table_search(Fst, Snd, Term)

Return, on backtracking, each entry in the hash table that has an index pair that unifies with (Fst, Snd) and corresponding value that unifies with Term. There is no guarantee about the order in which answers are returned.

mode hash_table_search(?atom, ?atomic, ?term)
3.13 Loading Programs

Programs can be loaded into Qu-Prolog in a number of ways. If the source of the program is available, \texttt{consult/1} (page 101) can be used. If the program has been compiled, it can be loaded with \texttt{load/1} (page 103).

Predicates:

\begin{verbatim}
[Files]

A synonym for consult(Files).

consult(Files)

Read the clauses from 	exttt{Files} into the dynamic database. Predicate definitions in 	exttt{Files} replace all current definitions of the same name and arity. The .ql extension is added to the file name only after consult fails to locate 	exttt{File}. consult also accepts .qle, .qg and .qge files as input. The .qle files are encoded files, the .qg files are term-expanded files and the .qge files are encoded files after term-expansion has been carried out.

reconsult(Files)

Read the clauses from 	exttt{Files} into the current interpreted program. Predicate definitions in 	exttt{Files} replace all current definitions of the same name and arity. The .ql extension is added to the file name only after reconsult fails to locate 	exttt{File}.

fcompile(Files)

fcompile(Files, OptionList)

Compile the code in 	exttt{Files}, using the compiler options given in \texttt{OptionList}. A .ql extension is added, if needed, to each file name in 	exttt{Files} to give the source file while the corresponding object file has a .qo extension. The behaviour is similar to that of qc. The options fall into three categories: those that affect the behaviour of the compiler; those that affect the storage allocated to the compiler; and those that affect the storage allocated to the executable.

1. Compiler Behaviour options:
   \begin{verbatim}
   define(AtomList)
   The preprocessor behaves as if a line of the form
   #define A
   had been added to the input files for each atom A in AtomList.
   (Default: [])
   preprocess_only(Boolean)
   Stop processing after the preprocessor phase.
   (Default: false.)
   expand_only(Boolean)
   Stop processing after the term expansion phase.
   (Default: false.)
   \end{verbatim}
\end{verbatim}

101
compile_only(Boolean)
Stop processing after the compilation phase.
(Default: false.)

assemble_only(Boolean)
Stop processing after the assembly phase.
(Default: false.)

object_file(File)
If the assemble_only flag is true, then the output of the assembly phase, a .qo file, is placed in this File. Otherwise, a pair of files are created to hold the final executable. These files are the result of linking. The first file's name is given by File, the other file's name is given by appending .qx to File.
(Default: No default.)

term-expand_file(File)
Term expansion rules are given in File.
(Default: '/dev/null'.)

verbose(Boolean)
Produce diagnostic output during processing.
(Default: false.)

2. Compiler Storage Options:

compiler_binding_trail(Integer)
The size in kilobytes of the binding trail used by the compiler.
(Default: 32.)

compiler_other_trail(Integer)
The size in kilobytes of the other trail used by the compiler.
(Default: 32.)

compiler_choice_point_stack(Integer)
The size in kilobytes of the choice point stack used by the compiler.
(Default: 64.)

compiler_environment_stack(Integer)
The size in kilobytes of the environment stack used by the compiler.
(Default: 64.)

compiler_heap(Integer)
The size in kilobytes of the heap storage used by the compiler.
(Default: 100.)

compiler_scratchpad(Integer)
The size in kilobytes of the scratchpad storage used by the compiler.
(Default: 2.)

compiler_name_table(Integer)
The number of entries in the compiler's name table.
(Default: 10000.)

compiler_ip_table(Integer)
The number of entries in the compiler's implicit parameter table.
(Default: 10000.)

3. Executable Storage Options:

executable_atom_table(Integer)
The number of entries in the executable's atom table.
(Default: 10000.)

**executable_code_area(Integer)**
The size in kilobytes of the executable’s code area.
(Default: 400.)

**executable_predicate_table(Integer)**
The number of entries in the executable's predicate table.
(Default: 10000.)

4. Options Affecting Both Compiler and Executable:

**string_table(Size_in_k)**
The size in kilobytes of the string table.
(Default: 64.)

**load(File)**
Load the object file File. Paths from QPLIBPATH and the suffix ‘.qo’ are added to the file name.

**define_dynamic_lib(Library, PredicateList)**
The predicates in PredicateList are defined in Library.
The declaration
define_dynamic_lib(mylib, [p/1, q/2])

will cause the file mylib.qo to be loaded when a call to an undefined p/1 or q/2 is made.

**mode** define_dynamic_lib(@atom, @list(compound))

### 3.14 Debugging

The debugger is based on the Procedure Box model of execution, which views program control flow in terms of movement about the program text. The debugger prints out the instantiation states of the goals being debugged at different points of interest (also known as ports). The ports are as follows:

- **call** The initial invocation of a predicate for a given goal.
- **exit** A successful termination from the predicate.
- **redo** Backtracking into the predicate.
- **fail** Failure of the predicate with respect to the initial goal.
- **throw** A throw/1 (page 21) occurs out of the predicate.

Through these five ports, information about the initial call and its outcome can be obtained without knowledge of the internal processing of the predicate being debugged.

Interaction is allowed at a port if

- The debugger is in **trace** mode and leashing is set for that port; or
The debugger is in debug mode, there is a spy point on the current predicate, and there are no spy conditions associated with the predicate; or

The debugger is in debug mode, there is a spy point on the current predicate, and some spy condition associated with the predicate succeeds.

A spy condition \texttt{spy\_cond(Goal1, Port, Goal2)} (page 106) succeeds only if \texttt{Goal1} and \texttt{Port} unify with the debugger goal and port and \texttt{Goal2} succeeds. Any bindings for variables in the debugger goal are discarded after unification with \texttt{Goal1} and execution of \texttt{Goal2}.

It is impractical to trace through a big program step by step. Spy points provide a method to skip over part of the program and interact with the debugger at predicates which are of interest. A spy point is placed at the predicate where the control flow will be viewed.

At each port, a message in the format below is displayed.

\textbf{Spy Id Depth Port: Goal ?}

- \textbf{Spy} A '+' indicates that there is a spy point at this predicate.
- \textbf{Id} An unique identifier for this invocation.
- \textbf{Depth} The number of ancestors for the current search path.
- \textbf{Port} The name of the port.
- \textbf{Goal} The goal.

If an interaction is allowed at a port, a '?' is printed and the debugger waits for a command. The available commands are given below.

\begin{itemize}
  \item \texttt{c} Creep. Take a single step to the next port.
  \item <RETURNN> The same as \texttt{c}.
  \item \texttt{l} Leap. Continue the execution until a spy point is reached or the program terminates.
  \item \texttt{s} Skip. Jump to the exit or the fail port of this predicate. This is valid at the call or the redo ports only.
  \item \texttt{f} [id] Fail. Fail this predicate or fail the execution to the goal given in id.
  \item \texttt{r} [id] Retry. Retry this predicate or the goal given in id.
  \item < n Print depth. Set the print depth to \texttt{n} (10, by default). The print depth is used to control the amount of detail printed for each goal.
\end{itemize}
d
Display. Display the current message again without using any operator property.

w
Write. Display the current message again using the available operator property.

p [n]
Print. Display the current message again using n or the default as the temporary print depth.

g [n]
Goals. Print the last n or all the ancestors of the current goal.

+ [arg]
Spy. Add a spy point to the current predicate. With an argument, read and add a spy condition.

- 
Nospy. Remove the spy point and any conditions from the current predicate.

= 
Print the current debugging status.

n 
Nodebug. Switch off the debugger.

@ 
Read and execute a goal.

b 
Break. Start another invocation of the interpreter.

a 
Abort. Abort the current execution.

h 
Help. Display a help message.

? 
The same as h.

The user can tailor the actions of the debugger by defining clauses for the hook predicates debugger_hook/4 (page 108) (called at every port) and debugger_cmd_hook/5 (page 109) (called after reading a debugger command). The Goal and Port arguments of these hooks are unified with the debugger goal and port respectively. Note that unification can bind variables in the debugger goal, as well as in Goal, so hooks can affect the execution of the program being debugged. DebugState captures the current debugger state in a structure debug_state(Spy, Id, Depth, Leash, PrintOptions). Spy, Id and Depth give further information about the port; Leash gives the current leash state; and PrintOptions is a list of options (see write_term/[2,3] (page 41)) that should be used for any output. If the hook succeeds, Action should be an atom controlling the next action of the debugger. Possible values for Action are listed below.
interact
Continue with the normal interaction for the current port.

creep
Set the debugger mode to trace and continue execution to the next port (as if by the c command).

continue
Continue execution in the current debugger mode.

fail
Fail the current predicate (as if by the f command).

If the hook fails, the debugger interacts normally.

To aid debugging of multiple threads, debugging is turned on/off for each thread independently.

Further, if the process (typically qp) is named then a single debugger GUI will appear for each thread for which debugging is enabled.

All examples are based on debugging a simple ancestor program.

Predicates:

debugging
Display status information about the debugger.
Example:

| ?- debugging.
The debugger is switched to off.
Leashing at [call exit redo fail exception ] ports.
yes

spy PredicateList
Add spy points at PredicateList, removing any associated conditions.
mode spy @closed_list(gcomp)
mode spy @gcomp
Example:

| ?- spy [male/1].
yes
| ?- debugging.
The debugger is switched to off
Spy points:
male/1
Leashing at [call exit redo fail exception ] ports.

spy_cond(Goal1, Port, Goal2)

106
Add a condition to the spypoint for Goal1’s predicate; if there is no
spypoint, create one.

\[
\text{mode spy_cond(@goal, @atom, @goal)}
\]

\[
\text{mode spy_cond(@goal, @var, @goal)}
\]

\text{nospy PredicateList}

Remove spy points and conditions at PredicateList.

\[
\text{mode nospy @closed_list(gcomp)}
\]

\[
\text{mode nospy @gcomp}
\]

\text{nospyall}

Remove all the spy points and conditions.

\text{debug}

Switch on the debugger. Produce trace at spy points only. If \text{xdebug} is
used instead then the debug GUI will start - see Section 3.25.

Example:

\[
| ?- debug. 
yes
| ?- debugging. 
The debugger is switched to debug. 
Spy points: 
\text{male/1}
Leashing at [call exit redo fail exception ] ports.
yes
\]

\[
| ?- father(john, X). 
+ 3 2 \text{call: male(john) ? l}
+ 3 2 \text{exit: male(john) ? l}
X = george;
+ 3 2 \text{redo: male(john) ? l}
+ 1 1 \text{fail: male(john) ? l}
+ 3 2 \text{call: male(john) ? l}
+ 3 2 \text{exit: male(john) ? l}
X = anne;
+ 3 2 \text{redo: male(john) ? l}
+ 1 1 \text{fail: male(john) ? l}
no
\]

\text{trace}

Switch on the debugger. Produce trace at every interpreted clause. If
\text{xtrace} is used instead then the debug GUI will start - see Section 3.25.

Example:
The debugger is switched to trace.
Spy points:
  male/1
Leashing at [call exit redo fail exception] ports.

?- father(john, X).
  1 1  call: father(john, X) ?
  2 2  call: parent(john, X) ?
  2 2  exit: parent(john, george) ?
+ 3 2  call: male(john) ?
+ 3 2  exit: male(john) ?
  1 1  exit: father(john, george) ?
X = george

leash(Mode)

Set leashing at the list of ports given in Mode. The available ports are:
call, exit, redo, fail, exception.
Mode can also be one of the following:
  none
  loose call
  half call, redo
  tight call, redo, fail, exception
  full  call, exit, redo, fail, exception
mode leash(@atom)

nodebug

Switch off the debugger.

notrace

Switch off the debugger. It is the same as nodebug/0 (page 108).

with_debugging_off(Goal)

Execute the Goal with the debugger (debug/0 (page 107) and trace/0 (page 107)) turned off.
mode with_debugging_off(+goal)

debugger_hook(Goal, Port, DebugState, Action)

A user-defined predicate that is called (if it exists) on every debugger port.
Example:
?- assert((debugger_hook(male(X), call, D, continue) :-
      write(X), nl, write(D), nl)).
X = X
D = D

?- father(john, X).
john
debug_state(spy, 4, 2, _125, [])
+ 4 2 exit: male(john) ? l
X = george

debugger_cmd_hook(DebugCmd, DebugArg, Goal, Port, Action)
A user-defined predicate that is called (if it exists) after reading a debugger command. DEBUG_CMD and DEBUG_ARG are the command and argument (as atoms) to be processed.

3.15 Foreign Language Interface

This interface enables procedures/functions written in another programming language to be called from Qu-Prolog. Currently, it supports C and C++. The interface can be divided into two levels as described below.

3.15.1 High Level Foreign Language Interface.

With this high level interface, arguments and function values are automatically converted between the representations in Qu-Prolog and the types used in the foreign language. The conversion is carried out by foreign/3 (page 109).

Predicates:

foreign_file(ObjectFile, ForeignFns)
The list of foreign functions, ForeignFns, which are callable from Qu-Prolog, are stored in ObjectFile. Any function that cannot be accessed should not appear in ForeignFns. The ObjectFile must have the .o extension. The file supplying this information cannot be compiled.

foreign(ForeignFn, ForeignSpec)
foreign(ForeignFn, Language, ForeignSpec)
The foreign function, ForeignFn, is written in Language, which default is C. ForeignSpec specifies the name of the predicate that will call ForeignFn. The arguments of ForeignSpec indicate how the arguments are transferred between the predicate and ForeignFn. Each of these arguments is declared by a combination of mode and a type given below.

Allowable modes
An argument to the function.
- A reference (pointer) argument to the function.
[-] The function returns this argument.

Allowable types
- integer  long
- float    double
- atom char *
- string char *

load_foreign_files(ObjectFiles)
load_foreign_files(ObjectFiles, Libraries)

Link the ObjectFiles written in another language with the support Libraries. This "high level foreign language interface" generates interface functions between Qu-Prolog and the foreign language. The .o extension in ObjectFiles is optional.

mode load_foreign_files(@atom)
mode load_foreign_files(@closed_list(atom))
mode load_foreign_files(@atom, closed_list(atom))
mode load_foreign_files(@closed_list(atom), closed_list(atom))

generate_foreign_interface(ObjectFiles, Interface)
generate_foreign_interface(ObjectFiles, Libraries, Interface)

This predicate is used to generate low-level interface files. Interface is the root name of the interface. This predicates generates a .cc and .ql file with root Interface. The .cc file is compiled. ObjectFiles and Libraries are as above. It requires definitions for foreign_file/2 and foreign/3.

mode generate_foreign_interface(@atom, @atom)
mode generate_foreign_interface(@closed_list(atom), @atom)
mode generate_foreign_interface(@atom, closed_list(atom), @atom)
mode generate_foreign_interface(@closed_list(atom), closed_list(atom), @atom)

Consider the following example code in 'test.cc'.

```
// foreign(twice, c, twice(+ integer, [- integer])).
// extern "C" long
twice(long a)
{
   return(2 * a);
}
//
```
The file is compiled with the appropriate compiler to obtain the object file.

g++ -c -fPIC test.cc

The interface is defined in test.ql with the following clauses.

foreign_file('test.o', [twice, triple, mkfoo]).
foreign(twice, c, twice(+ integer, [- integer])).
foreign(triple, 'c++', triple(+ float, - float)).
foreign(mkfoo, 'c++', mkfoo(- atom)).

When test.ql and the object file test.o is loaded, interface functions, such as those below, are generated for each function. These interface functions perform the necessary type checking and conversion.

#include "QuProlog.h"
extern "C" long twice(long);
extern void triple(double, double*);
extern void mkfoo(char **);
extern "C" bool
twice_interface(ForeignInterface* fi)
{
    bool result = true;
    long integer0;
    long integer1;
    Object* object0;
    Object* object1;
    Object* outarg1;
    object0 = fi->getXReg(0);
    if (!object0->isInteger())
{  
    return(false);
}

integer0 = object0->getNumber();
object1 = fi->getXReg(1);
integer1 = twice(integer0);
outarg1 = fi->makeInteger(integer1);
result = result && fi->unify(object1, outarg1);
return(result);
}

extern "C" bool
triple_interface(ForeignInterface* fi)
{
    bool result = true;
    double float0;
    double float1;
    Object* object0;
    Object* object1;
    Object* outarg1;
    object0 = fi->getXReg(0);
    if (!object0->isNumber())
    {
        return(false);
    }
    if (object0->isInteger())
    {
        float0 = object0->getNumber();
    }
    else
    {
        float0 = object0->getDouble();
    }
    object1 = fi->getXReg(1);
    triple(float0, &float1);
    outarg1 = fi->makeDouble(float1);
    result = result && fi->unify(object1, outarg1);
    return(result);
}

extern "C" bool
mkfoo_interface(ForeignInterface* fi)
{
    bool result = true;
    char * atom0;
    Object* object0;
    Object* outarg0;
    object0 = fi->getXReg(0);
Here is a sample Qu-Prolog session using these foreign functions.

```prolog
| ?- consult('test.ql'),
load_foreign_files('test.o'),
abolish(foreign_file/2),
abolish(foreign/3).
yes
| ?- twice(2, N).
N = 4;
no
| ?- triple(3.1, N).
N = 9.3;
no
| ?- mkfoo(X).
X = foo;
no
| ?- generate_foreign_interface('test.o', test_interface).
yes
```

The files test_interface.cc, test_interface.o and test_interface.ql are generated by the above call. test_interface.ql contains the required query to load_foreign/2 to initialize the low-level interface.

### 3.15.2 Low Level Foreign Language Interface

Interface functions are not generated for this interface. Operations such as dereferencing, type checking and conversions, and unification, are the responsibility of the user.

**Predicates:**

- `load_foreign(ObjectFiles, PredicateList)`
- `load_foreign(ObjectFiles, PredicateList, Libraries)`

   Link the ObjectFiles written in another language with the support Libraries. The ObjectFiles contains the predicates specified in PredicateList. Each element in PredicateList must be in either Name/Arity or Name/Arity=Function format. Name/Arity specifies the predicate that will call Function. The default name for Function is Name. This "low level foreign language interface" does not generate any additional interface functions. The
ObjectFiles must have the .o extension.

Using the interface functions in the above example, the following line will produce the same effect as load_foreign_files('test.o') (page 110).

```prolog
| ?- load_foreign(['test_interface.o', 'test.o'],
                 [twice/2=twice_interface,
                  triple/2=triple_interface,
                  mkfoo/1 = mkfoo_interface]).
```

### 3.16 Macros

The effect of macro expansions can be achieved in two ways. One is by using term expansion (page 86). The other is by using the inline declaration. The inline declarations are used by the compiler to inline code. Inlining is applied recursively by the compiler. The same directive to the runtime system also gives the interpreted code access to inline definitions.

The main difference between using term expansions and inlining is that term expansion is run as a preprocessor of the compiler, whereas inlining is done during compilation. It is therefore possible to, for example, inline code after higher-order goal unfolding and other compiler transformations.

**Predicates:**

- **inline(Goal1, Goal2)**
- **inline(Goal1, Goal2, Code)**

Declares Goal2 to be the inline expansion of Goal1. Code is code typically used to construct Goal2 from Goal1. Goal1 can be a special term of the form Term @ ArgList where Term is a term applied to the list of terms in ArgList.

The semantics for inlining is the same as for a call to Goal1 with definition Goal1 :- Goal2. For compiled code this predicate call is avoided (and Goal2 replaces Goal1) where the semantics is not affected.

**WARNING:** For efficiency reasons, an inlined goal that is also defined as a predicate has a different behaviour for compiled and interpreted code. It is therefore best to avoid using the same name for an inlined goal and a predicate.

- **mode inline(@nonvar, @goal)**
- **mode inline(@nonvar, @term, @goal)**
The following examples illustrate the use of inline to add two lists of numbers pairwise.

One way to do this is to define add/3 as a predicate and then define add_lists/3 as follows.

```
add_lists(L1, L2, Result) :-
    map(add, [L1, L2, Result]).
```

A slight inefficiency of this approach is that a call from add/3 to is/2 (page 83) is made on each recursive call for map (page 118).

On the other hand, if add/3 is defined using inlining as below then this overhead is avoided.

```
?- inline(add(X,Y,Z), Z is X+Y).
```

Note that, in this case, the compiler unfolds add_lists/3 to the following (up to the choice of the introduced predicate name).

```
add_lists(L1, L2, Result) :-
    '$add_lists_1'(L1, L2, Result).
'$add_lists_1'([], [], []).
'$add_lists_1'([H1|T1], [H2|T2], [H3|T3]) :-
    H3 is H1 + H2,
    '$add_lists_1'(T1, T2, T3).
```

This example can be further extended in a generic way by using inlining to define ‘anonymous’ (or ‘lambda’) predicates as below. Once this is done add_lists/3 may be defined as follows.

```
add_lists(L1, L2, Result) :-
    map(!!lambda ![x, !y, !z] (!z is !x + !y), [L1, L2, Result]).
```

The compiler will also unfold this to the code given earlier.

Anonymous predicates can be defined as follows.

```
:- inline(nfi_rev(X,Y), Y not_free_in X).
:- inline(!!lambda X B @ ArgList, C, 
    ( collect_vars(B, BVars),
      diff_list(BVars, X, BFreeVars),
      map(nfi_rev(BFreeVars), [X]),
      parallel_sub(ArgList, X, Sub),
      substitute(Call, [Sub], B),
      simplify_term(Call, C)
    )).
```

115
The inlining of lambda terms defines beta reduction. The first three goals in the third argument of the inline declaration make the bound variables not free in each free variable occurring in the body of the lambda term. The last three goals carry out the beta reduction by constructing the required substitution, applying it, and simplifying the result.

### 3.17 Higher-Order Predicates

The first argument of each of these higher-order predicates is a goal with zero or more arguments missing. These arguments are filled in during processing from components of the remaining arguments of the higher-order predicate. The order of arguments is the same as the order of the remaining arguments in the higher-order predicate.

Note that any variable in `HigherGoal` instantiated during a call to a higher-order predicate remains instantiated throughout the computation.

Predicates:

- **front_with** 
  > `front_with(HigherGoal, List1, List2)`
  
  List2 is the longest initial segment of `List1`, which satisfies `HigherGoal`.
  
  **mode** `front_with(+goal, +closed_list(term), ?closed_list(term))`
  
  **Example:**
  
  ```prolog
  | ?- front_with(atom, [a,b,1,c,d], R).
  R = [a, b]
  ```

- **after_with** 
  > `after_with(HigherGoal, List1, List2)`
  
  List2 is `List1` without the longest initial segment, whose elements satisfy `HigherGoal`.
  
  **mode** `after_with(+goal, +closed_list(term), ?closed_list(term))`
  
  **Example:**
  
  ```prolog
  | ?- after_with(atom, [a,b,1,c,d], R).
  R = [1, c, d]
  ```

- **build_structure** 
  > `build_structure(Functor, Arity, HigherGoal, Term)`
  
  Term is the structure with functor `Functor` and arity `Arity` and whose arguments are constructed using the higher order goal `HigherGoal`. `HigherGoal` is missing two arguments. The first is the position of the argument to be constructed and the second is the constructed argument. This method is preferred for constructing compound terms rather than using `functor` (page 60) and `arg` (page 60) because it avoids unnecessary occurs checks, and because it fills in the arguments with values directly (rather than by
creating a structure whose arguments are new variables and then in-
stantiating them).

mode build_structure(+term, @integer, +goal, ?compound)

Example:

| ├?- inline(rev_arg(F,N,A), arg(N,F,A)).  
| F = F  
| N = N  
| A = A  
| ├?- build_structure(f, 3, rev_arg(g(a,b,c,d)),R).  
| R = f(a, b, c)  
| ├?- inline(gen_args(Prefix, Number, Result),  
| (number_codes(Number, NumberCodes),  
| atom_codes(Atom, NumberCodes),  
| atom_concat(Prefix, Atom, Result) )).  
| Prefix = Prefix  
| Number = Number  
| Result = Result  
| NumberCodes = NumberCodes  
| Atom = Atom  
| ├?- gen_args(arg, 2, R).  
| R = arg2  
| ├?- build_structure(g, 4, gen_args(arg), R).  
| R = g(arg1, arg2, arg3, arg4)

filter(HigherGoal, List1, List2)

List2 contains all the elements from List1 which satisfy HigherGoal.

mode filter(+goal, +closed_list(term), ?closed_list(term))

Example:

| ├?- filter(atom, [a,X,1,b],R).  
| X = X  
| R = [a, b]  
| ├?- filter('<(3), [-2, 8, 1, 2, 4], R).  
| R = [8, 4]

fold(HigherGoal, Identity, List, Result)

Summarise the List with HigherGoal to give Result. Identity is the value used if List is an empty list.

The same as fold_right(HigherGoal, Identity, List, Result).

mode fold(+goal, +term, +closed_list(term), ?term)

Example:
fold_left(HigherGoal, Identity, List, Result)

Summarise the List with HigherGoal to give Result. Identity is the value used if List is an empty list. The evaluation is performed on the head of the list before the tail of the list.

Example:

| ?- fold_left(p, 0, [1,2,3], R).
| R = 1 + (2 + (3 + 0))

fold_right(HigherGoal, Identity, List, Result)

Summarise the List with HigherGoal to give Result. Identity is the value used if List is an empty list. The evaluation is performed on the tail of the list before the head of the list.

Example:

| ?- fold_right(p, 0, [1,2,3], R).
| R = 1 + (2 + (3 + 0))

map(HigherGoal, List)

List is a list of lists of arguments to HigherGoal. HigherGoal is applied in turn to the argument list constructed from the n'th elements of each list of arguments.

Example:
collect_simple_terms(HigherGoal, Term, Term1, Term2)

Summarise the simple terms (atomic or any variable) of Term into Term2 by carrying out a top-down, left-right pass over Term. Term1 is used as an initial value for the summarisation. Any simple terms that cause HigherGoal to fail are ignored.

Example:

| ?- inline(sum(A,B,C), (integer(A), C is A+B)).
A = A
B = B
C = C
| ?- collect_simple_terms(sum, f(g(2), a, 5), 0, X).
X = 7
| ?- inline(collect_var(A,B,C), (var(A), C = [A|B])).
A = A
B = B
C = C
| ?- collect_simple_terms(collect_var, f(g(X),a,Y), [], R).
X = X
Y = Y
R = [Y, X]
| ?- inline(build(New, Current, List), (atom(New),
    concat_atom([Current, '-', New], List))).
New = New
Current = Current
List = List
| ?- collect_simple_terms(build, foo(g(2), a, h(bar), c),
    start, R).
R = start-foo-g-a-h-bar-c

transform_simple_terms(HigherGoal, Term1, Term2)

This generates a sequence of calls HigherGoal(T1, T2). T1 is instantiated to each simple term in Term1, and the result of the transformation should be stored in T2. Term2 is the overall result of the transformation.
of Term1.

mode transform_simple_terms(+goal, +term, ?term)

Example.

| ?- inline(inc(X,Y), (integer(X) -> Y is X+1; Y=X)).  
X = X  
Y = Y  
| ?- transform_simple_terms(inc, f(X,1,g(2)), R).  
X = X  
R = f(X, 2, g(3))  

transform_subterms(HigherGoal, Term1, Term2)

This generates a sequence of calls HigherGoal(T1, T2). T1 is instantiated to each non-variable subterm of Term1 in turn, iterating from the bottom up (so arguments are transformed before structures and quantifications). Term2 is the overall result of the transformation of Term1.

mode transform_subterms(+goal, +term, ?term)

Example.

| ?- inline(change(X,Y), (X = a ->  
  Y = f(a); X = f(T) -> Y = f(T, T); Y = X)).  
X = X  
Y = Y  
T = T  
| ?- transform_subterms(change, g(f(f(a))), R).  
R = g(f(f(f(a)), f(a))), f(f(a), f(a))))

### 3.18 Implicit Parameters

Consider applications that manage internal state. Such applications include, for example, Definite Clause Grammars processors and theorem provers. In ordinary Prolog implementations, the state is typically handled by including arguments to predicates that represent input and output state information. For some applications where the state is simple, such as Definite Clause Grammars programming, these extra arguments are added by preprocessing the input program. In theorem proving applications the state is usually complex and is typically handled either by including many input/output state pairs to predicates, or by imbedding the state in a data structure for inclusion in predicates.

For processing complex states the above approaches are often clumsy and inefficient. Also, for applications such as interactive theorem provers, it is difficult to hide the internal state from users and from user tactics. Implicit parameters avoid these problems completely by providing an efficient and logically sound solution to the management of internal state.
The declarative semantics of implicit parameters can be explained in terms of a transformed program where a collection of input/output pairs of explicit parameters are added to predicates. The efficient implementation of implicit parameters is achieved via a form of backtrackable assignment.

Predicates:

```prolog
ip_set(Name, Term)
ip_set(Name1, Name2, Term)
```

The value `
Term`
 is stored in the implicit parameter `Name`. The value `
Term`
 is stored in the implicit parameter array `Name1` at offset `Name2`. `Name2` is atomic.

```prolog
mode ip_set(@atom, @term)
mode ip_set(@atom, @atomic, @term)
```

Example:

```
| ?- ip_set(language, english).
  yes
| ?- ip_set(weekday, 0, 'Sunday').
  yes
| ?- ip_set(weekday, 1, Day), Day = 'Monday'.
  Day = Monday
```

```prolog
ip_lookup(Name, Term)
ip_lookup(Name1, Name2, Term)
```

The implicit parameter `Name` contains `Term`. If `Name` is initially unset, the value `Term` is stored in it. Note that variables occurring in the implicit parameter may be instantiated by the unification with `Term`. The implicit parameter array `Name1` contains `Term` at offset `Name2`. `Name2` is atomic.

```prolog
mode ip_lookup(@atom, ?term)
mode ip_lookup(@atom, @atomic, ?term)
```

Example:

```
| ?- ip_set(language, english), ip_lookup(language, Lang).
  Lang = english
| ?- ip_set(weekday, 0, Day), ip_lookup(weekday, 0, 'Sunday').
  Day = Sunday
| ?- ip_set(weekday, 1, 'Monday'), ip_lookup(weekday, 1, 'Tuesday').
  no
```

```prolog
ip_array_clear(Name)
```

Clear the implicit parameter array associated with `Name`

```prolog
mode ip_array_clear(@atom)
```

Example:

```
```

121
\[ \text{ip_set}(\text{weekday}, 0, \text{'Sunday'}), \\
\text{ip_set}(\text{weekday}, 1, \text{'Monday'}), \\
\text{ip_set}(\text{weekday}, 2, \text{'Tuesday'}), \\
\text{ip_lookup}(\text{weekday}, 1, \text{Day1}), \\
\text{ip_array_clear}(\text{weekday}), \\
\text{ip_lookup}(\text{weekday}, 1, \text{Day2}). \]

\[ \text{Day1} = \text{Monday} \\
\text{Day2} = \text{Day2} \]

\text{ip_array_init}(\text{Name}, \text{Size})

Initialize the implicit parameter array associated with \text{Name} giving it a size that is the smallest power of two greater or equal to \text{Size} (the default size is 128). This is typically used when it is expected that the implicit parameter array will be used to store a large number of data items. If the default size is used then access/update performance will degrade as the number of entries gets large. If the array is initialized to a size that is greater of equal to the expected maximum number of data items then the time to access/update entries will remain constant.

\text{mode} \text{ip_array_init}(@\text{atom}, @\text{integer})

\text{ip_array_get_entries}(\text{Name}, \text{Entries})

This predicate returns a list of the ip array indices that have a corresponding value. The entries can then be used to iterate through the ip array and extract all the stored values.

The following predicates define ‘methods’ for a stack whose state is stored in an implicit parameter with name \text{stack}.

\text{init_stack} :- \text{ip_set}(\text{stack}, []). \\
\text{push_stack}(X) :- \\
\text{ip_lookup}(\text{stack}, \text{S}), \\
\text{ip_set}(\text{stack}, [X|\text{S}]). \\
\text{pop_stack}(X) :- \\
\text{ip_lookup}(\text{stack}, [X|\text{S}]), \\
\text{ip_set}(\text{stack}, \text{S}). \\
\text{empty_stack} :- \\
\text{ip_lookup}(\text{stack}, []). 

The stack should then behave as follows:

\[ \text{?- init_stack,} \\
\text{push_stack}(4), \text{push_stack}(3), \text{push_stack}(5), \\
\text{pop_stack}(X), \text{pop_stack}(Y), \text{pop_stack}(Z), \\
\text{write_term_list}([\text{w}(X), \text{tab}(3), \text{w}(Y), \text{tab}(3), \text{w}(Z), \text{nl}]), \text{fail}. \]

5 3 4

\text{no}
3.19 Unification Control

Many applications need to control unifications. For example, some theorem provers apply rewrite rules to carry out expression simplification. Such rules are often applied repeatedly until no further rewrites are applicable. Consider a rewrite system that includes the rule \( X \text{ and true} \rightarrow X \). If this rewrite rule is naively applied to the term \( A \) then infinite recursion will result. This problem can be controlled by implementing a form of one-sided unification. Applications written in Qu-Prolog can implement one-sided unification and other more sophisticated constrained unifications by ‘freezing’ variables. When a variable is frozen, it cannot be instantiated. Rewrite rules can be controlled by freezing the variables in the term to be simplified. This implements one-sided unification.

There are other reasons for controlling unifications in the implementation of theorem provers. One important example is when users wish to prove schematic theorems. It is usually the case that the user does not want variables appearing in the statement of the theorem to be instantiated during the proof. On the other hand, users may be happy if variables introduced during the proof (by the application of inference rules) are instantiated. Terms appearing in a proof often contain a mixture of the two kinds of variables described above. It is therefore critical that the instantiation of variables be controlled during rule applications. This is particularly important when executing tactics. Frozen variables can be used in this situation to achieve the desired result.

The "temperatures" of variables are reset on backtracking.

Predicates:

\[ \text{Term1 = Term2} \]

Unify Term1 and Term2.

\[ \text{mode +term = +term} \]

Example:

\[
\text{?- A is 10, A = B.}
A = 10
B = 10
\]

\[ \text{?- A is 10, B is 20, A = B.}
\]

\[ \text{no} \]

\[ \text{unify_with_occurs_check(Term1, Term2)} \]

This is not really needed because Qu-Prolog’s unification always carries out occurs checking. It is included for compatibility with the ISO standard.

\[ \text{structural_unify(Term1, Term2)} \]
Unify Term1 and Term2 as structures. Quantified terms and terms with substitutions are treated like compound terms.

**Example:**

```prolog
A = C
x = !y
B = D
C = C
y = !y
D = D
```

**Term1 \(\neq\) Term2**

Term1 and Term2 are unifiable.

**Example:**

```prolog
| ?- A is 10, A \(\neq\) B.
A = 10
B = B
| ?- A is 10, B is 20, A \(\neq\) B.
no
```

**freeze_term(Term)**

**freeze_term(Term, VarList)**

Freeze all the variables in Term, returning the list of newly frozen variables in VarList.

**Example:**
\begin{verbatim}
| ?- Term1 = f(A, B), Term2 = f(C, D),
  Term1 = Term2,
  write(Term1), nl, write(Term2), nl, fail.
  f(C, D)
  f(C, D)
  no
| ?- Term1 = f(A, B), Term2 = f(C, D),
  freeze_term(Term1, VarList), Term1 = Term2,
  write(Term1), nl, write(Term2), nl,
  write(VarList), nl, fail.
  f(A, B),
  f(A, B),
  [A, B]
  no

thaw_term(Term)
thaw_term(Term, VarList)

  Thaw all the variables in Term and return the newly thawed variables in VarList.
  mode thaw_term(+term)
  mode thaw_term(+term, -closed_list(anyvar))

freeze_var(Variable)

  Freeze Variable.
  mode freeze_var(+anyvar)

Example:

| ?- Term1 = f(A, B), Term2 = f(C, D), freeze_var(A), Term1 = Term2,
  write(Term1), nl, write(Term2), nl, fail.
  f(A, D)
  f(A, D)
  no

thaw_var(Variable)

  Thaw a Variable.
  mode thaw_var(+anyvar)

frozen_var(Term)

  Succeed if Term is a frozen variable.
  mode frozen_var(@term)

thawed_var(Term)

  Succeed if Term is a thawed variable.
  mode thawed_var(@term)
\end{verbatim}
One-sided unification may be implemented as follows.

```prolog
one_sided_unify(T1, T2) :-
    freeze_term(T1, FrozenVars),
    T1 = T2,
    thaw_term(FrozenVars).
```

### 3.20 Delayed Problem Handling

The delay mechanism suspends the execution of a goal until the variable given in `delay/2` (page 126) is instantiated. When the variable is bound, all the goals associated with this variable are woken up and made ready for execution. The woken goals are executed automatically at the next goal or cut. As well as the automatic retry mechanism, the delayed problems, both woken and suspended goals, can be retried manually by one of the `retry_` calls. Qu-Prolog may generate its own unification, `not_free_in` (page 58), and `check_binder` (page 59) delayed problems.

Predicates:

- `delay(Variable, Goal)`
  - Goal is delayed until `Variable` is instantiated. If `Variable` is not a variable or object variable at the time of call then `Goal` will be called immediately.
  - mode delay(+term, +goal)
  - Example:
    ```prolog
    | ?- delay(X, write(X)), write(b), X=a, write(c), nl, fail.
    bac
    no
    | ?- X=a, delay(X, write(X)), write(b), write(c), nl, fail.
    abc
    no
    ```

- `delay_until(Term, Goal)`
  - Goal is delayed until the delay condition `Term` is satisfied.
  - The possible values of `Term` are given below.
    - `nonvar(X)`
      - Satisfied when `X` becomes a non-variable.
    - `ground(X)`
      - Satisfied when `X` becomes ground.
    - `bound(X)`
      - Satisfied when the variable (or object variable) `X` is bound to a term (including another variable or object variable).
• identical_or_apart(X, Y)
  Satisfied when identical_or_apart/2 succeeds.
• or(A,B)
  Satisfied when either of the delay conditions A or B is satisfied.
• and(A,B)
  Satisfied when both of the delay conditions A or B are satisfied.

delay_until(+compound, +goal)
Example:

| ?- delay_until(bound(X), write(X)),
  write(a), X = Y, write(b), Y = f(Z), write(c),
  Z=a, write(d), nl, fail.
aYbcd
no |

| ?- delay_until(nonvar(X), write(X)),
  write(a), X = Y, write(b), Y = f(Z),
  write(c), Z=a, write(d), nl, fail.
abf(Z)cd
no |

| ?- delay_until(ground(X), write(X)),
  write(a), X = Y, write(b), Y = f(Z), write(c),
  Z=a, write(d), nl, fail.
abcf(a)d
no |

get_delays(DelayList)

DelayList is the list of delayed problems.

get_delays(?closed_list(goal))
Example:

| ?- delay_until(ground(X), write(X)), get_delays(L).
  X = X
  L = [delay_until(ground(X), write(X))]
  provided:
  delay_until(ground(X), write(X)) |
| ?- [A/x]B = 3, get_delays(L).
  A = A
  x = !x
  B = B
  L= [[A/x]B = 3]
  provided:
  [A/x]B = 3

127
Note from this example that the interpreter displays all delays related to the query in the provided section.

```
get_var_delays(Var, DelayList)

   DelayList is the list of delayed problems associated with Var.
   mode get_var_delays(@var, ?closed_list(goal))
```

```
get_unify_delays(DelayList)

   DelayList is the list of delayed unification problems.
   mode get_unify_delays(?closed_list(goal))
```

```
get_unify_delays_avoid(DelayList, Avoid)

   DelayList is the list of delayed unification problems other than those in Avoid. This predicate is useful for situations where some computation might generate new delayed unification problems. This predicate can be used to extract any new problems.
   mode get_unify_delays_avoid(?closed_list(goal), +closed_list(goal))

Example:

```
| ?- get_unify_delays(Old), some_computation, get_unify_delays_avoid(New, Old).
```

```
retry_delays

   Retry all the unsolved delayed problems.

Example:

```
| ?- delay(X, write(X)), write(b), retry_delays, write(c), nl, fail.
bXc
no
| ?- [A/x]B = 3, retry_delays, get_delays(L).
A = A
x = !x
B = B
L = [[A/x]B = 3]
provided:
[A/x]B = 3
| ?- [A/x]B = 3, A = 2, retry_delays, get_delays(L).
A = 2
x = !x
B = 3
L = []
```

```
retry_var_delays(Var)
```

128
Retry all the delayed problems associated with Var.

\texttt{retry\_woken\_delays}  
\texttt{retry\_woken\_delays(Goal)}

Retry all the delayed problems which are woken up by unification and then execute Goal. These predicates are generally not needed in applications as woken delays will be automatically retried at the next call.

\texttt{collect\_constraints(VarList, List1, List2, List3)}

List1, List2, and List3 contain lists of constraints associated with the variables in VarList. List1 gives the distinctness information as \texttt{not\_free\_in/2} (page 58) goals, while other delayed \texttt{not\_free\_in/2} goals are in List2. List3 contains all the remaining delayed problems.

\begin{verbatim}
mode collect_constraints(@closed_list(anyvar), -closed_list(goal),  
                        -closed_list(goal), -closed_list(goal))

Example:

| ?- [A/!x]B = 3, !x not_free_in f(A, !y),  
   delay_until(ground(C), A = 2),  
   collect_constraints([C, !x, !y], List1, List2, List3).
A = A  
x = !x  
B = B  
y = !y  
C = C  
List1 = [!x not_free_in [!y], !y not_free_in [!x]]  
List2 = [!x not_free_in A]  
List3 = [[A/!x]B = 3, delay_until(ground(C), A = 2)]
provided:  
[A/!x]B = 3  
delay_until(ground(C), A = 2)  
!x not_free_in A  
!y not_free_in [!x]  
!x not_free_in [!y]
\end{verbatim}

### 3.21 Program State

Information about the program state can be obtained via the following predicates.

Predicates:

\texttt{current\_atom(Atom)}
Atom is a currently defined atom.
mode current_atom(?atom)

current_predicate(PredicateName/Arity)

PredicateName with Arity is a currently defined predicate.
mode current_predicate(?compound)

Example:

| ?- current_predicate(foobar/1).  no |
| ?- current_predicate(current_predicate/1). yes |

get_args(List)

List is the list of arguments supplied at the invocation of the application.
mode get_args(?closed_list(ground))

main(List)

A user defined predicate. It is the default entry point for applications.
List is the list of command line arguments given at the invocation of the application. These are arguments other than the arguments used to set the area sizes and name the process. The command line arguments are the arguments following '–' or arguments without a preceding '-'.

predicate_property(Head, Property)

The predicate specified by Head has Property (built_in, multile, dynamic, static, foreign).
mode predicate_property(?goal, ?atom)

Example:

| ?- predicate_property(true, Value). Value = built_in |
| ?- assert((father(A, B) :- male(A), parent(A, B))). A = A B = B |
| ?- predicate_property(father(A, B), Value). A = A B = B Value = dynamic |

statistics
statistics(Key, Value)
statistics(ThreadID, Key, Value)
The first predicate displays a summary of the statistical information on the current output. The remaining two predicates return in Value the statistics for Key. If ThreadID is given then that statistic for that thread is returned. The three argument version is of most use for gathering thread specific statistics of another thread. Fails if ThreadID is not a current thread.

The possible values for Key are given below.

- **global_stack**
  Value: \([\text{Used}, \text{Free}, \text{MaxUsage}]\).
  Unit: words.
  Meaning: Used and free space and the maximum usage in the global stack (heap).

- **scratchpad**
  Value: \([\text{Used}, \text{Free}, \text{MaxUsage}]\).
  Unit: words.
  Meaning: Used and free space and the maximum usage in the scratchpad stack.

- **local_stack**
  Value: \([\text{Used}, \text{Free}, \text{MaxUsage}]\).
  Unit: words.
  Meaning: Used and free space and the maximum usage in the local (environment) stack.

- **choice**
  Value: \([\text{Used}, \text{Free}, \text{MaxUsage}]\).
  Unit: words.
  Meaning: Used and free space and the maximum usage in the choice point stack.

- **binding_trail**
  Value: \([\text{Used}, \text{Free}, \text{MaxUsage}]\).
  Unit: words.
  Meaning: Used and free space and the maximum usage in the binding trail.

- **other_trail**
  Value: \([\text{Used}, \text{Free}, \text{MaxUsage}]\).
  Unit: words.
  Meaning: Used and free space and the maximum usage in the other trail.

- **code**
  Value: \([\text{Used}, \text{Free}]\).
  Unit: words.
  Meaning: Used and free space in the code area.

- **string**
  Value: \([\text{Used}, \text{Free}]\).
Unit: words.
Meaning: Used and free space in the string area.

- **name**
  Value: [Used, Free].
  Unit: words.
  Meaning: Used and free entries in the name table.

- **ip_table**
  Value: [Used, Free].
  Unit: words.
  Meaning: Used and free entries in the implicit parameter table.

- **atom**
  Value: [Used, Free].
  Unit: words.
  Meaning: Used and free entries in the atom table.

- **predicate**
  Value: [Used, Free]
  Unit: words.
  Meaning: Used and free entries in the predicate table.

- **runtime**
  Value: [Start, Last].
  Unit: milliseconds.
  Meaning: Runtime since the start, and last runtime statistics/2 call.

Example:

```prolog
| ?- statistics.  
```

<table>
<thead>
<tr>
<th>Data Area</th>
<th>Total</th>
<th>Used</th>
<th>Free</th>
<th>Max Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>global stack</td>
<td>102400</td>
<td>161</td>
<td>102239</td>
<td>717</td>
</tr>
<tr>
<td>scratchpad</td>
<td>2560</td>
<td>0</td>
<td>2560</td>
<td>0</td>
</tr>
<tr>
<td>local stack</td>
<td>65536</td>
<td>101</td>
<td>65435</td>
<td>107</td>
</tr>
<tr>
<td>choice stack</td>
<td>65536</td>
<td>107</td>
<td>65429</td>
<td>171</td>
</tr>
<tr>
<td>binding trail</td>
<td>32768</td>
<td>0</td>
<td>32768</td>
<td>3</td>
</tr>
<tr>
<td>other trail</td>
<td>32768</td>
<td>0</td>
<td>32768</td>
<td>0</td>
</tr>
<tr>
<td>code area</td>
<td>409600</td>
<td>321343</td>
<td>88257</td>
<td>321343</td>
</tr>
<tr>
<td>string area</td>
<td>65536</td>
<td>51337</td>
<td>14199</td>
<td>51337</td>
</tr>
<tr>
<td>number table</td>
<td>4096</td>
<td>840</td>
<td>3256</td>
<td></td>
</tr>
<tr>
<td>name table</td>
<td>32768</td>
<td>0</td>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>IP table</td>
<td>32768</td>
<td>3</td>
<td>32765</td>
<td></td>
</tr>
<tr>
<td>atom table</td>
<td>32768</td>
<td>3301</td>
<td>29467</td>
<td></td>
</tr>
<tr>
<td>predicate table</td>
<td>32768</td>
<td>2286</td>
<td>30482</td>
<td></td>
</tr>
<tr>
<td>runtime</td>
<td>120 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

yes

```prolog
| ?- statistics(code, Value).  
Value = [321343, 88257]  
```

132
3.22 Exception Handling

There are two kinds of exception handlers: local and global. When an exception is raised (whether by the system or by user code), the current handler state is checked. If a local handler for that exception is in scope, the innermost such handler is called; if the handler returns, execution continues after the `with_local_exception_handler/3` (page 137) call. Otherwise, if a global handler for the exception has been defined, the most recent matching handler is called. If the handler returns, execution continues after the goal that raised the exception (unless the severity of the exception forbids that). If neither a local or global handler for the exception is defined, the default action is taken: this default action is to throw an exception term.

Because of problems with signal handling in POSIX threads the only signal that is caught by Qu-Prolog is `SIGINT`. All other signals have their default behaviour. The `SIGINT` signal is passed to `exception/1` (page 133) for handling.

When a `SIGINT` is detected the system forks a Qu-Prolog thread to handle the signal.

Because local exception handlers are thread specific then a local exception handler cannot trap `SIGINT` signals. The Qu-Prolog interpreter catches thrown exceptions (printing a suitable error message) and also catches `ctrlC_reset` thrown by a reset in the default `SIGINT` handler.

Predicates:

`exception(Data)`

Raise an exception. Exceptions are represented by terms of the form `ExceptionKind(Severity, Goal, Message, ...)`. `Severity` is an atom indicating how severe the exception is (which determines the default and minimum actions for that exception): see `exception_severity/3` (page 136). `Goal` is the goal being executed when the exception happened. `Message` determines the message to display by default when that error happens: see `get_exception_message/2` (page 136). Each `ExceptionKind` will have other arguments giving information specific to that kind. The following built-in exceptions have been defined.

- `context_error(Severity, Goal, Message)`
  Goal appeared in the wrong context.
- `declaration_error(Severity, Goal, Message)`
  A missing declaration.
- `dynamic_code_error(Severity, Goal, Message)`
  An attempt to assert a clause for a static predicate.
- `exception_error(Severity, Goal, Message)`
  A problem was detected with an exception handler. `Goal` is the exception call that lead to the problem.
• instantiation_error(Severity, Goal, Message, ArgNo, Modes)
  ArgNo is the argument number (of Goal) where the error was detected. Modes is a list of mode declarations giving the allowable modes for Goal.

• permission_error(Severity, Goal, Message)
  A permission error occurred when opening a stream.

• range_error(Severity, Goal, Message, ArgNo, Range)
  ArgNo is the argument number (of Goal) where the error was detected. Range is a term indicating the expected range.

• signal(Severity, Goal, Message, Name)
  Name is the name of the signal. Under Unix, this is an atom like 'SIGINT', 'SIGXCPU' etc.

• stream_error(Severity, Goal, Message, Stream)
  Stream is the stream being manipulated by the erroneous goal.

• syntax_error(Severity, Goal, Message, Stream, StreamPos0, StreamPosN, LineNo0, LineNoN, Tokens, TokensPos)
  Stream is the stream with the syntax error.
  StreamPos0 and StreamPosN give the stream position at the start and end of the term with the syntax error.
  LineNo0 and LineNoN give the line number at the start and end of the term with the syntax error.
  Tokens is the list of tokens that could not be parsed.
  TokensPos is the index into Tokens where the error was detected.

• type_error(Severity, Goal, Message, ArgNo, Modes)
  ArgNo is the argument number (of Goal) where the error was detected. Modes is a list of mode declarations giving the allowable modes for Goal.

• undefined_predicate(Severity, Goal, Message)
  The called predicate is undefined.

The possible values for Severity are given below. The default action will be done (after printing a message) if no handler is found. The minimum action will be done if the user handler returns.

• information

• warning

• recoverable
  Default: fail. Minimum: true (Note 1).

• unrecoverable
  Default: fail. Minimum: fail (Note 2).

• fatal
  Default: halt. Minimum: halt (Note 3).
Notes:

1. A handler for a recoverable error should succeed only if it has achieved the effect of Goal in some way.
2. A handler for an unrecoverable error may fail or throw.
3. A handler for a fatal error may throw, but should normally halt the program.

Message is typically a list of the form used for write_term_list.

Example:

?- exception(declaration_error(recoverable, Goal, 'some silly type')).
Recoverable error: missing declaration for some silly type

?- exception(type_error(fatal, dwarf(happy), default, 1, [dwarf(+grumpy)])).
Fatal error: type error in dwarf(happy) at argument 1
    allowable modes are [dwarf(+ grumpy)]

exception_exception(Goal)

Raise an exception_error exception for Goal. The same as exception_exception(unrecoverable, Goal, default)) (page 133).

Example:

?- exception_exception(Goal).
Unrecoverable error: exception error while handling Goal

instantiation_exception(Goal, N, PredicateMode)

Raise an instantiation_error exception for Goal with PredicateMode at argument N. The type of this argument should be AbstractType.

Example:
% instantiation_exception(goal(Arg1, Arg2), 2, 
[goal(-integer, +goal), goal(+integer, -goal)])
Recoverable error: instantiation error in goal(Arg1, Arg2) at argument 2
allowable modes are [goal(-integer, +goal), goal(+integer, -goal)]
no

type_exception(Goal, N, PredicateMode)
type_exception(Goal, N, PredicateMode, AbstractType)

Raise a type_error exception for Goal with PredicateMode at argument N. The type of this argument should be AbstractType.
mode type_exception(+goal, @integer, @closed_list(compound))
mode type_exception(+goal, @integer, @closed_list(compound), @atom)
Example:

| ?- type_exception(dwarf(2), 1, [dwarf(+dwarf_type)], dwarf_type).
Unrecoverable error: type error in dwarf(happy) at argument 1
expected type dwarf_type
no

exception_severity(Data, Action1, Action2)

The default and minimum actions for exception Data are Action1 and Action2, respectively.
mode exception_severity(+compound, ?atom, ?atom)
Example:

| ?- exception_severity(exception_error(recoverable, G, M), Min, Max).
G = G
M = M
Min = fail
Max = true
| ?- exception_severity(exception_error(fatal, G, M), Min, Max).
G = G
M = M
Min = halt
Max = halt

get_exception_message(Data, Msg)

The default message for exception Data is Msg.
If user_exception_message(Data, Msg) succeeds, the result is used as the message. If the Msg argument of Data is a list of terms, it is used as the message. Otherwise, a message is constructed according to the
particular exception type. In any case the resulting \texttt{Msg} is a suitable second argument for \texttt{write_term_list/[1,2]} (page 44).

\textbf{mode get_exception_message(+compound, ?closed_list(term))}

Example:

\begin{verbatim}
| ?- get_exception_message(type_error(Sev, Goal, Msg, Arg, Mode),
   Message), write_term_list(Message).
Information: type error in Goal at argument Arg allowable modes are Mode
Sev = information
Goal = Goal
Msg = default
Arg = Arg
Mode = Mode
Message = [Information: , type error in , w(Goal),
  at argument , w(Arg), nl, allowable modes are , w(Mode), nl]
\end{verbatim}

\textbf{with_local_exception_handler(Goal1, Data, Goal2)}

Set up \texttt{Goal2} as a local handler for exception \texttt{Data}, then call \texttt{Goal1}.

\textbf{mode with_local_exception_handler(+goal, +compound, +goal)}

Consider an application where certain arithmetic evaluations are to recover from such errors by returning 0. This can be achieved using the following definition.

Example (assuming the following predicate definition):

\begin{verbatim}
eval(Expr, Result) :-
   with_local_exception_handler(Result is Expr,
   instantiation_error(_, is(_,_,_,_,_),
   Result = 0).
| ?- X is Y+1.
Recoverable error: instantiation error in _FA is Y + 1 at argument 2 allowable modes are [? number is @ gcomp, ? number is @ number]
no
| ?- eval(Y+1,X).
Y = Y
X = 0
\end{verbatim}

\textbf{add_global_exception_handler(Data, Goal)}

Add \texttt{Goal} as a global handler for exception \texttt{Data}. If an exception matching \texttt{Data} happens and is not caught by any local handler or any later global handler, \texttt{Goal} will be called (with no arguments). See Section 3.11.

\textbf{mode add_global_exception_handler(@compound, @goal)}

Example:
?- add_global_exception_handler(
    instantiation_error(_, is(R,_,_,_,_), R = 0).
| ?- X is Y+1.
Y = Y
X = 0

remove_global_exception_handler(Data, Goal)
Remove the most recent global exception handler matching Data and Goal.
mode remove_global_exception_handler(?compound, ?goal)
Example (Continued from last):
| ?- remove_global_exception_handler(instantiation_error(_, is(_,_,_,_,_), _).
| ?- Y is X +1.
Recoverable error: instantiation error in _103 is X + 1 at argument 2
  allowable modes are [? integer is @ gcomp, ? integer is @ integer]
no

default_exception_handler(Data)
Perform the default exception handler for Data (see exception/1 (page 133) for the possible values for Data). The default handler typically throws an exception term.
mode default_exception_handler(+compound)
default_exception_error(Data)
Write the default error message for the given exception data to standard error.
mode default_exception_error(+compound)
default_signal_handler(Signal)
Execute the default handler for Signal. In the current implementation, if the signal is SIGINT then the default handler produces a menu of choices for how to proceed.
mode default_signal_handler(@atom)
The following example shows how to declare a signal handler that responds to a SIGINT signal by displaying a message and continuing.

?- add_global_exception_handler(signal(recoverable, _, _, 'SIGINT'),
    write_msg).
write_msg :-
    write('You can’t kill me'),nl.
Because the signal handler executes in a special thread it does not have access to the state of any of the other threads. The following example shows how to declare a signal handler that is specific to the main thread (thread 0). It is assumed that my_handler is a defined predicate to be called in the main thread when SIGINT is detected.

?- add_global_exception_handler(signal(recoverable, _, _, 'SIGINT'), thread_push_goal(0, my_handler)).

It is possible to write the SIGINT handler so that local exception handlers can be used to indirectly trap SIGINT signals. For example, the predicate my_handler could be defined as follows.

my_handler :-
   exception(my_exception(_,_,_)).

If the following call is made then the local exception handler will respond to SIGINT.

with_local_exception_handler(MyGoal, my_exception(_,_,_), ExceptGoal)

### 3.23 Multiple Threads

This section describes predicates for creating and managing threads within a single Qu-Prolog process. All threads have a symbolic name (an atom).

Predicates:

**thread_set_symbol(Name)**

Set the symbolic name for this thread to Name. Fails if the name is already taken or is self or pedro.

*mode* thread_set_symbol(@atom)*

Example:

| ?- thread_set_symbol(foo). |

**thread_symbol(Name)**

Returns the symbolic Name of the current thread.

*mode* thread_symbol(-atom)*

Example (Continued from last):

| ?- thread_symbol(Symbol). |

Symbol = foo
creates a new thread called ThreadName and sets its goal to be Goal. ThreadName cannot be the name of any existing threads and cannot use the names self or pedro. Use the current defaults for the sizes of the data areas unless otherwise specified in Sizes. The call will fail if the thread cannot be created, if the name is unavailable, if the goal cannot be set or if the supplied sizes are erroneous. Note that the new thread is placed just before the creator thread in the run queue and so does not obtain a timeslice until just before the creator thread gets its next turn.

If ThreadName is a variable at the time of call then the system will generate a name. If you wish to generate your own thread names based on a root name then use the RootName (an atom). The system will generate a name which is the root name followed by an integer. Note that such names will be reused (i.e. the name given to a now-terminated thread can be given to a new thread). You need to be careful because the thread handle will be reused and so messages to the terminated thread will be picked up by the new thread with the same handle.

Possible items for inclusion in Sizes are:

- choice_size(Integer)
  Integer is the size of the choice stack in K words.
  Default: 64

- env_size(Integer)
  Integer is the size of the environment stack in K words.
  Default: 64

- heap_size(Integer)
  Integer is the size of the heap in K words.
  Default: 400

- binding_trail_size(Integer)
  Integer is the size of the binding trail in K words.
  Default: 32

- other_trail_size(Integer)
  Integer is the size of the other trail in K words.
  Default: 32

- scratchpad_size(Integer)
  Integer is the size of the scratchpad heap in K words.
  Default: 100

- name_table_size(Integer)
  Integer is the size of the name table.
  Default: 10000
• `ip_table_size(Integer)`
  Integer is the size of the IP table.
  Default: 10000

`thread_fork(?atom, @gcomp)`
`thread_fork(?atom, @gcomp, @closed_list(compound)`

Example:

```plaintext
| ?- thread_fork(myThread,
   (repeat, Msg <<= Addr:_,
    write_term_list(['Recieved: ', w(Msg), ' from: ',
    w(Addr), nl]), fail)
   ).
Msg = Msg
Addr = Addr
| ?- 'hello world' ->> myThread, fail.
no
| ?- Recieved: hello world from: thread0
(A is 10 + 5) ->> myThread, fail.
no
| ?- Recieved: A is 10 + 5 from: thread0
```

The message communication predicates `<<` and `->>` are described in Section 3.24.

Note that the execution of `myThread`'s goal is independent from the success or failure of any other thread, and its result is written to standard output after the completion of the query in the initial thread (`thread0`). Thus the message appears next to the prompt, while the next query is written on the line below.

`thread_is_thread(Thread)`

Succeeds iff `Thread` is a valid thread ID and the thread that it refers to exists.

mode `thread_is_thread(+thread_id)`

Examples (Continued from last):

```plaintext
| ?- thread_is_thread(foo). \% the main thread
yes
| ?- thread_is_thread(myThread).
yes
| ?- thread_is_thread(aaa).
no

thread_is_initial_thread
```

141
Succeeds iff the current thread is an initial thread. The main thread is an initial thread.

mode thread_is_initial_thread

thread_exit
thread_exit(Thread)

End execution of the current thread or the specified Thread. Fails if Thread is not a current thread.

mode thread_exit
mode thread_exit(+thread_id)

thread_push_goal(Thread, Goal)

Push Goal to the front of the current conjunction of goals on the specified Thread. This predicate is used in the default handler for SIGINT. The given thread must be different from the thread in which this call is made. Fails if Thread is not a current thread.

mode thread_push_goal(+thread_id, +term)

thread_atomic_goal(Goal)

Call Goal with multi-threading turned off. Multi-threading is turned back on when Goal exits (either by success or failure). This predicate has the effect of making Goal atomic with respect to multi-threading.

mode thread_atomic_goal(+term)

thread_wait
thread_wait(Conditions)

Wait on the default set of conditions or on the given set of Conditions. Execution resumes when any condition is satisfied. If a condition isn’t specified, its default is used.

The available conditions are:

- db(Boolean)
  Wait on any database change (including changes to the global state and hash table).
  default: true

- timeout(T)
  The call will wait for T seconds before continuing.
  default: block

Note: A timeout of block means that the thread will never timeout. Currently, if the conditions are set to false, false and block then the thread can never be woken.

mode thread_wait
mode thread_wait(+closed_list(compound))
thread_wait_on_goal(Goal)
thread_wait_on_goal(Goal, Options)

Options is a list containing:

- at most one occurrence of a term of the form \texttt{wait_preds(PredList)}
  where \texttt{PredList} is a list of terms of the form \texttt{P/N} or of the form \texttt{P/N-Stamp} where \texttt{Stamp} is a database stamp - see \texttt{get_predicate_timestamp} on page 94
- at most one occurrence of a term of the form \texttt{wait_for(Secs)}
- at most one occurrence of a term of the form \texttt{retry_every(Secs)}
- at most one of the above two terms

\texttt{Goal} is first tried (unless \texttt{wait_preds(PredList)} includes stamps), and if it succeeds, then \texttt{thread_wait_on_goal} succeeds. Otherwise, \texttt{thread_wait_on_goal} blocks and waits for the database to change before retrying \texttt{Goal}. If \texttt{wait_preds(PredList)} is given as an option then it waits until one of the listed predicates change. If \texttt{wait_for(Secs)} is given and \texttt{Goal} is not successfully retried within the given number of seconds then \texttt{thread_wait_on_goal} fails. If \texttt{retry_every(Secs)} is given then \texttt{Goal} is retried at least every \texttt{Secs} seconds, even if there is no change to the dynamic database.

It is sometimes useful for \texttt{Goal} to be able to access the predicates that have changed when timestamps are used in \texttt{wait_preds}. To support this, the variant \texttt{wait_preds(PredList, CPreds)} can be used (where \texttt{CPreds} is an unbound variable at the time of call). If \texttt{PredList} contains timestamps then each time \texttt{Goal} is tried \texttt{CPreds} will be rebound to those predicates in \texttt{PredList} that have been changed since the last time \texttt{Goal} was tried. As a further enhancement, the predicates in \texttt{PredList} can be prefixed by + or -. A + (respectively -) means that the goal will be retried if the predicate is changed by an assert (respectively retract) of the predicate.

Note that if the supplied goal produces a side-effect and then fails, then the side-effect will be produced each time the database is modified. Also note that, if \texttt{retry_every(Secs)} is given, one way the goal could succeed without a change to the database is if the Prolog state changed in another way, for example by the use of random or because of a message or socket event.

mode thread_wait_on_goal(+term)
mode thread_wait_on_goal(+term, @list(terms))

thread_sleep(Time_Out)
This is the same as
\texttt{thread_wait([timeout(Time_Out), db(false)]).}
mode thread_sleep(+number)

thread_sleep_until(Time)
This is similar to `thread_sleep` except that the argument is the time at which the thread should wake as the number of seconds since the Unix epoch. See `realtime/[1]` (page 156).

```
mode thread_sleep_until(+integer)
```

```
thread_defaults(DefaultList)
```

Get the values for the default sizes of data areas used when a new thread is created.

The entries returned in `DefaultList` are the same as for `thread_fork/[3,4]` (page 140).

```
mode thread_defaults(-closed_list(compound)
```

```
thread_set_defaults(DefaultList)
```

Set the values for the default sizes of data areas used when a new thread is created.

The items allowed in `DefaultList` are the same as for `thread_fork/[3,4]` (page 140).

```
mode thread_set_defaults(+closed_list(compound)
```

```
thread_errno(Errno)
```

Returns the current errno for this thread.

```
mode thread_errno(-integer)
```

```
schedule_threads_now(ThreadNames)
```

This runs a single timeslice for all threads whose names are listed in `ThreadNames`. This predicate produces an error if a listed thread does not exist, the calling thread is listed, or is called by a thread within the execution of `schedule_threads_now`.

```
mode schedule_threads_now(@list(atoms))
```

```
thread_yield
```

The thread gives up its timeslice.

```
mode thread_yield
```

### 3.24 Interprocess Communication

Qu-Prolog supports high-level interprocess communication using the Pedro server. Pedro supports both subscription/notification and peer-to-peer messages. A Qu-Prolog process can connect to the Pedro server and then send messages to and receive messages from other Pedro clients.

The reader is referred to the Pedro reference manual for further details.

For peer-to-peer messages addresses (handles) are used to determine the recipient of a message, and the received message will be accompanied by the address of the sender and a reply-to address if required.
A handle is of the form \texttt{ThreadName:ProcessName@MachineName} where the components are respectively the name of the thread, the registered name of the process and the name of the machine on which the process is running.

When specifying addresses the handle can be used, but if the address is for a process on the same machine then the machine name can be elided. Further, if the address is for a thread on the same process then the process name can also be elided. The special address \texttt{self} is the shorthand for the address of this thread.

For non-Qu-Prolog processes for which the thread name is not relevant, handles can be of the form \texttt{ProcessName@MachineName}. If a process uses such a handle when sending a message to a Qu-Prolog process then the initial thread (thread0) will receive the message. If the thread name is a variable then all running threads on the process will receive the message.

It is also possible, when sending a message, to use a handle with either the process name or the machine name being a variable. If the process name is a variable all registered processes on that machine will receive the message. If the machine name is a variable all registered processes with that name (on any machine) will receive the message. If both are variables then the message will be sent to all registered processes on all machines that have registered processes.

All messages (both notifications and peer-to-peer), together with address information, received by a thread are placed in a message queue for that thread. A notification received by a thread that is not a peer-to-peer message will get the handle \texttt{pedro}

Predicates:

\begin{verbatim}
pedro_connect
pedro_connect(Machine)
pedro_connect(Machine, Port)

Connect to an Pedro server. Machine is the machine on which the Pedro server is running (default localhost) and Port is the port on which the Pedro server is listening (default 4550).

mode pedro_connect
mode pedro_connect(@atom)
mode pedro_connect(@atom, @integer)

Example:

|?- pedro_connect.
|?- pedro_connect(fred).
|?- pedro_connect(fred, 4999).
\end{verbatim}

\begin{verbatim}
pedro_disconnect

Disconnect from the Pedro server.

mode pedro_disconnect
\end{verbatim}

145
pedro_is_connected

Succeeds if and only if the process is connected to Pedro.
mode pedro_is_connected

pedro_register(Name)

Register Name with the Pedro server. The process needs to be connected first. This name is used as part of the address of this process for peer-to-peer messages. If the process is started with the -Aname switch then the process connects with the Pedro server and registers the supplied name.
mode pedro_register(@atom)

pedro_deregister

Deregisters the current name with the Pedro server.
mode pedro_deregister

pedro_is_registered

Succeeds if and only if the process is registered with Pedro.
mode pedro_is_registered

pedro_subscribe(Head, Body, ID)

Subscribe to notifications which unify with Head and satisfy the goal Body. The Pedro reference manual lists the goals that can be used in Body. If the subscription succeeds ID will be instantiated to the ID for this subscription. This ID is used when unsubscribing.
mode pedro_subscribe(@compound, @goal, ?integer)

pedro_unsubscribe(ID)

Remove the subscription with the supplied ID.
mode pedro_unsubscribe(@integer)

pedro_notify(T)

Sends the compound term T as a notification.
mode pedro_notify(@compound)

thread_handle(Handle)

Handle is the handle (peer-to-peer address) of this thread.
mode thread_handle(?handle)

same_handle(Handle1, Handle2)

This is true if Handle1 and Handle2 represent the same handle. Handle1 must be ground at the time of call and be an handle.
mode same_handle(@handle, ?handle)

ipc_send(Message, ToAddress)

ipc_send(Message, ToAddress, OptList)
Send Message to ToAddress. The options supplied in OptList will determine the actions taken during sending.

The only available option is:

- `remember_names(Value)`
  Remember the names of the variables in Message.
  Default: true.

```
mode ipc_send(+term, @handle)
mode ipc_send(+term, @handle, @handle)
mode ipc_send(+term, @handle, @handle, +closed_list(compound))
```

```
ipcpeek(Message)
ipcpeek(Message, FromAddress)
ipcpeek(Message, FromAddress, OptList)
```

Examine the incoming message queue of this thread for a message that will unify with Message and an address that will unify with FromAddress. The options supplied in OptList will determine the actions taken on peeking at the messages in the queue.

Address unification is done via `same_handle/2` and so shortened forms of handles can be given in the address patterns.

The available options are:

- `timeout(Value)`
  Value is either block, poll or a number. If Value is block then the call delays until a message arrives. If Value is poll then the call fails immediately if no message is in the queue. If Value is a number, N, then the call will wait N seconds for a message, and if no matching message arrives in that time the call fails.
  Default: block

- `remember_names(Value)`
  Remember the names of variables in the incoming Message
  Default: true

```
mode ipcpeek(?term)
mode ipcpeek(?term, ?handle)
mode ipcpeek(?term, ?handle, +closed_list(compound))
```

```
ipcrecv(Message)
ipcrecv(Message, FromAddress)
ipcrecv(Message, FromAddress, OptList)
```

Attempt to unify Message with a message from the thread’s message queue if possible. FromAddress will be unified with the address of the sender of the message. The available options are identical to those available with `ipcpeek/4` (page 147).
mode `ipc_recv(?term)
mode `ipc_recv(?term, ?handle)
mode `ipc_recv(?term, ?handle, +closed_list(compound))

Examples (of the above four predicates):

| ?- thread_handle(MyHandle),
  ipc_send('Hello World', MyHandle, []).
MyHandle = thread0 : foo @ bloggs
| ?- ipc_peek(Msg, FromHandle).
Msg = Hello World
FromHandle = thread0 : foo @ bloggs
| ?- `ipc_recv(Msg, Th:PID).
Msg = Hello World
Th = thread0
PID = foo

Msg ->> Address
Send Msg to Address.
mode @term ->> @handle

Msg <<- Address
Read a message from the thread’s message queue and set Address to the
senders address.
mode ?term <<- ?handle

Msg <<= Address
Search the thread’s message queue for a message that will unify with Msg
and Address.
mode ?term <<= ?handle

Examples (Of the above three predicates):

| ?- 'Other World' ->> self.
yes
| ?- Msg <<- Th:_.
Msg = Hello World
Th = thread0
| ?- 'Other World' <<= self.
yes

broadcast(Term)
Send Term as a message to all current threads within this process.
mode broadcast(+term)
message_choice(ChoiceGoals)
message_choice(ChoiceGoals;timeout(T) -> TimeoutGoal)

ChoiceGoals takes the form:
Term1 -> Goal1; Term2 -> Goal2; ... ; TermN -> GoalN

Term1 to TermN are either message patterns or of the form Msg::Goal
where Msg is a message pattern and Goal is a goal (for compatibility with
logtalk %% can be used instead of ::). Message patterns take the form
Msg <<- Addr. Where timeout(T) -> TimeoutGoal appears it should
be last as any subsequent choices are ignored.

Messages in the incoming message queue are tested against the test terms
Term1 to TermN. The first message to match against the test terms and
the first such matching test term are chosen.

The message is removed from the message queue, alternatives are cut and
the goal Goali corresponding to the chosen term Termi is called.

If no matching messages are found then the call suspends until another
message arrives or until the (optional) timeout is reached, in which case
the TimeoutGoal is called.

mode message_choice(+term)

Example:

| ?- msg(apple, orange) ->> self.  yes |
| ?- msg(pear, orange) ->> self.    yes |
| ?- msg(grape, grape) ->> self.    yes |
| ?- msg(pear, grape) ->> self.    yes |
| ?- repeat, message_choice((
  (msg(apple, _) <<- _) -> (write('Apple Pie'), nl, fail);
  (msg(_, orange) <<- _) -> (write('Orange Juice'), nl, fail);
  (msg(F1, F2) <<- _)::(F1=F2) -> (write('Matching Fruit'), nl, fail);
  timeout(5) -> (write('No (more) matching messages'), nl)
))

Apple Pie
Orange Juice
Matching Fruit
No (more) matching messages
F1 = F1
F2 = F2

pedro_address(IPAddress)

IPAddress is the IP address of the Pedro server with which the process
is connected.

mode pedro_address(?atom)
pedro_port(Port)

Port is the port used by the Pedro server with which the process is connected.
mode pedro_port(?integer)

set_default_message_thread(ThreadName)

Set the name of the thread that receives peer-to-peer messages that do not include a thread name as part of the address.
mode set_default_message_thread(@atom)

default_message_thread(ThreadName)

Return true iff ThreadName is the name of the default message thread.
mode default_message_thread(?atom)

3.25 Graphical User Interface

Qu-Prolog comes with a GUI built on the QT libraries. The program xqp is a GUI interface to qp built on these libraries. The program xqpdebug is a GUI interface to the debugger also using QT. If a qp process is registered with Pedro then starting the debugger using xtrace or xdebug will cause the xqpdebug interface to start. This debugger is specific to the thread that started the debugger. It is possible to start debuggers in severals threads.

3.26 Garbage Collection

Garbage collection of the heap (global stack) is triggered when the heap becomes over 90% full or by making a call to gc. If the garbage collector does not cause the heap to drop below 90% full then the process terminates with a heap overflow error.

Note that the garbage collector may "thrash" if it only recovers enough space for the heap to drop just below 90% full.

Predicates:

gc

Trigger the garbage collector.

3.27 TCP/IP

It is expected that most communications between processes will be carried out using Pedro communication. Where communication with applications such as FTP or HTTP servers are required, communications using sockets may be needed. The predicates described in this section provide support for basic socket communication.
A simple example using these predicates is given in the examples directory.

Predicates:

**tcp_server(Socket, Port)**

Open a socket using the specified Port and IPAddress. This call is intended for setting up a socket that will be used in the server end of some processing relationship. If Port is zero then some unused port will be chosen.

mode tcp_server(-integer, +integer)

mode tcp_server(-integer, +integer, +integer)

**tcp_client(Port, IPAddress, Socket)**

Open a socket using the specified Port and IPAddress. This call is intended for setting up a socket that will be used in the client end of some processing relationship.

mode tcp_client(+integer, +integer, -integer)

**tcp_open(Socket)**

Open a socket.

mode tcp_open(-integer)

**tcp_close(Socket)**

Close a socket. This closes an opened input (output) stream for this socket.

mode tcp_close(+integer)

**tcp_bind(Socket, Port)**

Bind Socket to the specified Port and IPAddress. If Port is zero then some unused port will be chosen.

Corresponds to the C function: bind(3N).

mode tcp_bind(+integer, +integer)

mode tcp_bind(+integer, +integer, +integer)

**tcp_listen(Socket)**

Listen for a connection on the socket.

Corresponds to the C function: listen(3N).

mode tcp_listen(+integer)

**tcp_connect(Socket, Port, IPAddress)**

Establish a connection between Socket and the specified Port and IPAddress. Corresponds to the C function: connect(3N).

mode tcp_connect(+integer, +integer, +integer)
tcp_accept(Socket, NewSocket)
tcp_accept(Socket, NewSocket, Port, IPAddress)

Accept a connection to Socket.
Corresponds to the C function: accept(3N).
mode tcp_accept(@integer, -integer)
mode tcp_accept(@integer, -integer, @integer, @integer)

open_socket_stream(Socket, IOMode, Stream)

Open a stream for reading or writing on Socket.
mode open_socket_stream(@integer, @atom, -integer)

tcp_getsockname(Socket, Port, IPAddress)

Return the local port and IP address associated with the socket.
Corresponds to the C function: getsockname(3N)
mode tcp_getsockname(+integer, -integer, -integer)

tcp_getpeername(Socket, Port, IPAddress)

Return the remote port and IP address associated with the socket.
Corresponds to the C function: getpeername(3N)
mode tcp_getpeername(+integer, -integer, -integer)

tcp_host_to_ip_address(Name, IPAddress)

Look up the IP address of the given host.
Corresponds to the C function: nslookup(1N)
mode tcp_host_to_ip_address(+atom, -integer)

tcp_host_from_ip_address(Name, IPAddress)

Look up the host name of the given IP address.
Corresponds to the C function: nslookup(1N)
mode tcp_host_from_ip_address(-atom, +integer)

3.28 CHR System

This section briefly describes the Qu-Prolog implementation of the K.U.Leuven
Constraint Handling Rules (CHR) system (see http://www.cs.kuleuven.be/dtai/projects/CHR/).
This implementation is reasonably consistent with that of SWI Prolog.

The syntax of CHR rules is given below.

chr_rules --> chr_rule, chr_rules.
chr_rules --> [].
chr_rule --> name, rule_part, pragma, ['.'].
name --> atom, ['@'].
name --> [].

152
The semantics of the rules are as follows.
Whenever a new constraint is added or some variable in an existing constraint is bound (causing the constraint of be retried) then the constraint becomes 'active'. The rules are then tried in order. If the active constraint together with other ('passive') constraints match the head of a rule and the guard is true then the body of the rule is executed. Note that variables occurring in the constraints are not allowed to be bound when matching against the rule head or in the guard.

There are three rules types:

- **Simplification**
  The matched head constraints are removed from the constraint store.

- **Propagation**
  The matched head constraints are not removed and the rule cannot be replied to the same collection of constraints (in the same order) again.

- **Simpagation**
  The constraints before the \ are retrained while the constraints after the \ are removed.

The following examples are adapted from the reference above.

```%
CHR rules for less_than_or_equal
```
The directive `chr_init` must occur before any CHR directives or rules. This loads the CHR operator declarations and enables the term expansion rules to convert the rules to Qu-Prolog clauses.

The `chr_constraint` declares the allowed constraints. This is followed by the required rules. Normal Qu-Prolog code can be freely intermixed with CHR rules.

The next example uses the passive pragma.

```prolog
:- chr_init.
:- chr_constraint fibonacci/2.
\%\% fibonacci(N,M) is true iff M is the Nth Fibonacci number.
fibonacci(N,M1) # Id \ fibonacci(N,M2) <=> var(M2) \ M1 = M2 pragma passive(Id).
fibonacci(0,M) ==> M = 1.
fibonacci(1,M) ==> M = 1.
fibonacci(N,M) ==> N > 1 | 
    N1 is N-1,
    fibonacci(N1,M1),
    N2 is N-2,
    fibonacci(N2,M2),
    M is M1 + M2.
```

In this example, the first constraint in the first rule can only be passive.

### 3.29 Miscellaneous

Predicates:

`os(Term)`

Access the operating system.

The possible values for `Term` are

- `access(File, Mode)`
  The same as `access(File, Mode, 0)`.
- `argv(Args)`
  The same as `get_args/1` (page 130).
- `exit(Integer)`
  The same as `halt/1` (page 19).
• **mktemp(Atom, File)**
  Call the Unix system call `mktemp(3)` which returns a unique file name, `File`. The name is derived from the `Atom` which must have six trailing Xs.

• **system(Cmd)**
  Execute the `Cmd` using `/bin/sh`.

• **system(Cmd, Status)**
  Execute the `Cmd` and return exit code in `Status` using `/bin/sh`.

**mode os(+compound)**
Example:

```prolog
| ?- os(mktemp(fileXXXXXX, File)).
File = filenCwJaB
| ?- os(system(date)).
Tue Jan 11 14:40:04 EST 2000
yes
```

**current_prolog_flag(Flag, Value)**

Extract the Value of the prolog Flag. Currently, the only flag is `version` – the version number of Qu-Prolog.

**mode current_prolog_flag(@atom, ?term)**

**interpreter**

Start the standard Qu-Prolog interpreter. This is useful for stand-alone applications that need an interpreter as part of the processing.

**mode interpreter**

**chdir(Dir)**

Change directory to that given by the atom `Dir`.

**mode chdir(@atom)**

Example:

```prolog
| ?- chdir('..').
changes to the parent directory.
```

**getcwd(Dir)**

Get the current working directory.

**mode getcwd(-atom)**

**working_directory(Dir)**

**working_directory(OldDir, NewDir)**
Get/set the current working directory with working_directory(Dir).
If Dir is a variable then it is set to the working directory. Otherwise it
should be an atom representing a directory, in which case the working
directory is set to Dir.
working_directory(OldDir, NewDir) sets the working directory. NewDir
is the new working directory and OldDir is previous working directory.
These are interfaces to chdir and getcwd.
mode working_directory(?atom)
mode working_directory(-atom, @atom)

absolute_file_name(File, Path)
Expand File (file or directory) to its full path (Path). Environment vari-
ables, , . and .. are expanded.
mode absolute_file_name(+atom, -atom)

file_directory_name(File, Path)
Expand File (file or directory) to its full path and return the directory
(Path). Environment variables, , . and .. are expanded.
mode file_directory_name(+atom, -atom)

file_base_name(File, FileBase)
Expand File (file or directory) to its full path and return the file base
name (FileBase). Environment variables, , . and .. are expanded.
mode file_base_name(+atom, -atom)

getenv(EnvVar, Value)
Get the environment variable EnvVar and unify with Value.
mode getenv(+atom, -atom)

setenv(EnvVar, Value)
Set the environment variable EnvVar to Value.
mode setenv(+atom, +atom)

prompt(Variable, Atom)
The current prompt is returned in Variable before setting the prompt
to Atom. The prompt is reset to the previous value on backtracking.
mode prompt(?atom, @atom)
Example:
    | ?- prompt(P, atom), write(P), nl, prompt(P2, P).
    | ?- P = | ?-
P = atom

realtime(Time)
The argument is unified with the current time given as the number of seconds since the Unix epoch. It has the same semantics as the Unix C function `time(2)`.

```prolog
mode realtime(?integer)
```

gmtime(Time, GMT)

gmtime(Time, GMT) is true if and only if GMT is a compound term of the form `time(Year, Month, Day, Hour, Min, Sec)` that corresponds to the GMT time for Time as the number of seconds since the Unix epoch. When Time is given, it has the same semantics as the Unix C function `gmtime`.

```prolog
mode gmtme(@integer, ?compound)
mode gmtme(?integer, +compound)
```

Example:

```
?- gmtme(ET, time(100, 5, 6, 10, 5, 20)).
ET = 960285920
?- gmtme(960285920,GT).
GT = time(100, 5, 6, 10, 5, 20)
```

localtime(Time, LTime)

```prolog
localtime(Time, LTime) is true if and only if LTime is a compound term of the form `time(Year, Month, Day, Hour, Min, Sec, IsDS)` that corresponds to the local time for Time as the number of seconds since the Unix epoch. When Time is given, it has the same semantics as the Unix C function `localtime`.
```

```prolog
mode localtime(@integer, ?compound)
mode localtime(?integer, +compound)
```

gettimeofday(Time)

The argument is unified with the current time given as the number of seconds since the Unix epoch. It is similar to `realtime` except it contains the fractional part of the time.

```prolog
mode gettimeofday(?double)
```

create_timer(Goal, Time, OneTime, ID)

This creates a timer that in Time seconds will call Goal. If OneTime is true then the timer will be deleted when the goal is called. Otherwise, the timer will reset so that the goal is repeadly called every Time seconds.
ID is a unique ID given to the timer. When the thread that created the timer exists then all its timers will be removed. The goal will be called once and then failed over and so the goal should have a side-effect such as database modification. If an exception is thrown in the goal then the entire process terminates.

```prolog
mode create_timer(+goal, @number, @term, -integer)
```

delete_timer(ID)

Delete the timer with the given ID. This thread must be the the thread that created the timer.

```prolog
mode delete_timer(@integer)
```

## 4 Standard Operators

```prolog
:- op(1200, xfx, [ :- , --> ]).
:- op(1200, fx, [ :- , ?- ]).
:- op(1100, xfy, [ ; ]).
:- op(1050, xfy, [ -> ]).
:- op(1024, xfx, [ '::' ]).
:- op(1000, xfy, [ ',' ]).
:- op(900, fy, [ \*, spy, nospy ]).
:- op(700, xfx, [ =, \=, ?=, ==, \==, @<, @=<, @>, @=>, 
                   =.., is, =:=, =\=, <, <=, >, >= ]).
:- op(600, xfx, [ not_free_in, is_free_in, is_not_free_in ]).
:- op(500, xfy, [ +, - , \/, \// ]).
:- op(500, fy, [ message_choice ]).
:- op(452, xfx, [ ->>, +>>, <<- , <<= ]).
:- op(400, yfx, [ * , / , // , rem , mod , << , >> ]).
:- op(200, xfx, [ ** ]).
:- op(200, xfy, [ {} ]).
:- op(200, fy, [ ? , @ , + , - , \ ]).
:- op(100, xfx, [ @ ]).
:- op(50, xfx, [ : ]).
```

## 5 Notation

Below is a summary of the notations used to describe the built-in predicates.

**AbstractType**

An abstract Qu-Prolog type.

**Action**

An atom specifying an action.
Argument
A term which is an argument of a compound term.

Arity
An integer giving the arity of a predicate or a compound term.

Associativity
An atom giving the associativity of an operator.

Atom
An atom.

Atomic
An atom or a number.

AtomList
A list of atoms.

Body
The body of a clause or a term.

Boolean
One of the atoms true or false.

Character
An atom denoting a single character.

CharCode
An integer giving the character code of a character.

CharCodeList
A list of character codes.

Clause
A clause.

Closed
A closed (proper) list.

Data
A compound term describing the exception.

DebugState
A compound term describing the state of the debugger.

**DebugArg**

An atom standing for an argument of a debugger command.

**DebugCmd**

A character (atom) standing for a debugger command typed by the user.

**DelayList**

A list of delayed problems.

**DistinctList**

A list object variables.

**Expression**

An expression.

**File**

An atom giving the name of a file.

**Files**

A list of files separated by commas.

**Float**

A double precision float.

**ForeignFn**

An atom denoting the name of the foreign function.

**ForeignFns**

A list of atoms denoting the names of the foreign functions.

**ForeignSpec**

A predicate describing the type and mode of its arguments.

**Functor**

The functor of a compound term.

**Goal**

An atom or a compound term.

**Head**
The head of a clause.

**HigherBody**

An atom or a compound term. Extra arguments will be added to the term to form a body.

**HigherGoal**

An atom or a compound term. Extra arguments will be added to the term to form a goal.

**HigherHead**

An atom or a compound term. Extra arguments will be added to the term to form a head.

**Identity**

An identity of an operation (goal).

**Integer**

An integer.

**Language**

The name of the foreign programming language.

**Library**

An atom naming a library.

**Libraries**

A list of atoms giving the names of the libraries.

**Limit**

An integer.

**List**

A list.

**Location**

An integer.

**Mode**

An atom or a list of atoms.

**Message**

A list of terms.
N
An integer.

Name
An atom.

ObjectFile
An atom giving the name of the object file.

ObjectFiles
A list of atoms or an atom giving the names of the object files.

ObVar
An object variable.

OnOff
An atom on or an atom off.

Open
An open (improper) list.

Operator
An atom describing a binary relationship.

OptionList
A list of options, which depend on individual predicate.

Permission
An integer encoding the access permission of a file.

Port
An atom naming a port of the debugger.

Precedence
An integer giving the precedence of an operator.

PredicateList
A list of "predicate/arity" pairs.

PredicateMode
A list of mode declarations.

PredicateName
An atom which gives the name of a predicate.

**Property**

An atom or a compound term describing a property.

**Quantified**

A quantified term.

**Quantifier**

The quantifier of a quantified term.

**Reference**

An integer, which is like a pointer.

**Result**

The result of an operation (goal).

**Rule**

A Definite Clause Grammars rule.

**Signal**

An atom.

**Stream**

A compound term holding the information about a stream.

**StringMode**

An atom or a compound term describing the mode of operation and the string to be parsed.

**Substituted**

A term with a substitution.

**Substitutions**

A list of lists representing a sequence of substitutions.

**Table**

An atom naming a table.

**Template**

A compound term.

**Term**

163
Any Qu-Prolog term.

Type
   An atom describing the type of an argument.

Variable
   A meta or object variable.

VariableNames
   A list of Variable=Name pairs.

VarList
   A list of variables.
6 Index

! ................................................................. 19
'C' ............................................................... 88
, ................................................................. 17
-> ............................................................... 18
->; ............................................................... 18
->>> ............................................................. 148
+>>>> .......................................................... ??
; ................................................................. 17
< ............................................................... 82
<< ............................................................. 148
<= ............................................................. 148
= ............................................................... 123
=:= ............................................................. 60
=:= ............................................................. 82
=<= ............................................................. 82
== ............................................................. 50
=
= ............................................................... 82
> ............................................................... 83
>= ............................................................. 83
?=. .............................................................. 124
@< ............................................................. 51
@= ............................................................. 51
@=< ............................................................. 51
@> ............................................................. 51
@>= ............................................................. 51
[] ............................................................. 101
\+ ............................................................. 18
\= ............................................................. 124
\== .......................................................... 50
- ............................................................... 80
abolish ......................................................... 95
absolute_file_name ................................. 156
access ......................................................... 23
add_expansion ............................................. 86
add_expansion_vars ................................. 86
add_global_exception_handler ............................. 137
add_linking_clause ......................................... 96
add_multi_expansion ....................................... 86
add_multi_expansion_vars ............................... 87
add_subterm_expansion ................................. 87
add_subterm_expansion_vars .............................. 87
after_with .................................................. 116
any_variable ................................................ 53
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>75</td>
</tr>
<tr>
<td>arg</td>
<td>60</td>
</tr>
<tr>
<td>assert</td>
<td>93</td>
</tr>
<tr>
<td>asserta</td>
<td>93</td>
</tr>
<tr>
<td>assertz</td>
<td>93</td>
</tr>
<tr>
<td>at_end_of_stream</td>
<td>26</td>
</tr>
<tr>
<td>atom</td>
<td>52</td>
</tr>
<tr>
<td>atom_chars</td>
<td>62</td>
</tr>
<tr>
<td>atom_codes</td>
<td>62</td>
</tr>
<tr>
<td>atom_concat</td>
<td>63</td>
</tr>
<tr>
<td>atom_concat2</td>
<td>64</td>
</tr>
<tr>
<td>atom_length</td>
<td>64</td>
</tr>
<tr>
<td>atom_search</td>
<td>64</td>
</tr>
<tr>
<td>atomic</td>
<td>52</td>
</tr>
<tr>
<td>bagof</td>
<td>80</td>
</tr>
<tr>
<td>between</td>
<td>83</td>
</tr>
<tr>
<td>body</td>
<td>68</td>
</tr>
<tr>
<td>bound_var</td>
<td>67</td>
</tr>
<tr>
<td>break</td>
<td>19</td>
</tr>
<tr>
<td>broadcast</td>
<td>148</td>
</tr>
<tr>
<td>build_structure</td>
<td>116</td>
</tr>
<tr>
<td>call</td>
<td>20</td>
</tr>
<tr>
<td>callable</td>
<td>20</td>
</tr>
<tr>
<td>call_cleanup</td>
<td>22</td>
</tr>
<tr>
<td>call_predicate</td>
<td>20</td>
</tr>
<tr>
<td>catch</td>
<td>21</td>
</tr>
<tr>
<td>changed_predicates</td>
<td>94</td>
</tr>
<tr>
<td>char_code</td>
<td>62</td>
</tr>
<tr>
<td>chdir</td>
<td>155</td>
</tr>
<tr>
<td>check_binder</td>
<td>59</td>
</tr>
<tr>
<td>clause</td>
<td>94</td>
</tr>
<tr>
<td>close</td>
<td>25</td>
</tr>
<tr>
<td>closed_list</td>
<td>72</td>
</tr>
<tr>
<td>closed_to_open</td>
<td>72</td>
</tr>
<tr>
<td>collect_constraints</td>
<td>129</td>
</tr>
<tr>
<td>collect_simple_terms</td>
<td>119</td>
</tr>
<tr>
<td>collect_vars</td>
<td>68</td>
</tr>
<tr>
<td>compare</td>
<td>51</td>
</tr>
<tr>
<td>compound</td>
<td>55</td>
</tr>
<tr>
<td>concat_atom</td>
<td>68</td>
</tr>
<tr>
<td>consult</td>
<td>101</td>
</tr>
<tr>
<td>copy_term</td>
<td>68</td>
</tr>
<tr>
<td>create_timer</td>
<td>157</td>
</tr>
<tr>
<td>current_atom</td>
<td>129</td>
</tr>
<tr>
<td>current_input</td>
<td>26</td>
</tr>
<tr>
<td>current_obvar_prefix</td>
<td>34</td>
</tr>
</tbody>
</table>
filter ................................. 117
findall ................................ 81
float .................................. 53
flush_output .......................... 28
fold .................................. 117
cold_left .............................. 118
cold_right ............................ 118
forall .................................. 82
foreign ................................. 109
foreign_file ......................... 109
freeze_term .......................... 124
freeze_var ......................... 125
front_with ......................... 116
frozen_var .......................... 125
frozen_var ......................... 125
functor ................................ 60
gc .................................. 150
generate_foreign_interface ........ 110
get .................................. 46
get0 .................................. 46
get_args .............................. 130
get_char .............................. 47
get_code .............................. 47
get_predicate_timestamp .......... 94
get_delays ......................... 127
get_distinct ........................ 69
get_exception_message ......... 136
get_line .............................. 47
get_linking_clause ............... 96
get_name ............................ 93
get_open_streams ................ 31
get_unify_delays ............... 128
get_unify_delays_avoid ........ 128
get_unnamed_vars ............... 37
get_var_delays .................. 128
get_var_name .................... 36
getcwd ................................ 155
getenv ................................ 156
gettimeofday ...................... 157
global_state_decrement .......... 99
global_state_increment .......... 99
global_state_lookup ............. 99
gmtime ................................ 157
global_state_set ................ 99
ground ................................ 54
halt .................................. 19
<table>
<thead>
<tr>
<th>Variable</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>nl</td>
<td>48</td>
</tr>
<tr>
<td>nodebug</td>
<td>108</td>
</tr>
<tr>
<td>nonvar</td>
<td>54</td>
</tr>
<tr>
<td>nospy</td>
<td>107</td>
</tr>
<tr>
<td>nospyall</td>
<td>107</td>
</tr>
<tr>
<td>not_free_in</td>
<td>58</td>
</tr>
<tr>
<td>notrace</td>
<td>108</td>
</tr>
<tr>
<td>number</td>
<td>53</td>
</tr>
<tr>
<td>number_chars</td>
<td>63</td>
</tr>
<tr>
<td>number_codes</td>
<td>63</td>
</tr>
<tr>
<td>obvar</td>
<td>55</td>
</tr>
<tr>
<td>obvar_name_to_prefix</td>
<td>35</td>
</tr>
<tr>
<td>obvar_prefix</td>
<td>34</td>
</tr>
<tr>
<td>obvar_prefix_table</td>
<td>34</td>
</tr>
<tr>
<td>once</td>
<td>21</td>
</tr>
<tr>
<td>op</td>
<td>33</td>
</tr>
<tr>
<td>op_table</td>
<td>32</td>
</tr>
<tr>
<td>op_table_inherit</td>
<td>32</td>
</tr>
<tr>
<td>open</td>
<td>23</td>
</tr>
<tr>
<td>open_append</td>
<td>77</td>
</tr>
<tr>
<td>open_length</td>
<td>77</td>
</tr>
<tr>
<td>open_list</td>
<td>72</td>
</tr>
<tr>
<td>open_member</td>
<td>78</td>
</tr>
<tr>
<td>open_member_eq</td>
<td>78</td>
</tr>
<tr>
<td>open_msgstream</td>
<td>25</td>
</tr>
<tr>
<td>open_socket_stream</td>
<td>152</td>
</tr>
<tr>
<td>open_string</td>
<td>24</td>
</tr>
<tr>
<td>open_tail</td>
<td>78</td>
</tr>
<tr>
<td>open_to_closed</td>
<td>73</td>
</tr>
<tr>
<td>os</td>
<td>154</td>
</tr>
<tr>
<td>pedro_address</td>
<td>149</td>
</tr>
<tr>
<td>pedro_connect</td>
<td>145</td>
</tr>
<tr>
<td>pedro_deregister</td>
<td>146</td>
</tr>
<tr>
<td>pedro_disconnect</td>
<td>145</td>
</tr>
<tr>
<td>pedro_is_connected</td>
<td>146</td>
</tr>
<tr>
<td>pedro_is_registered</td>
<td>146</td>
</tr>
<tr>
<td>pedro_notify</td>
<td>146</td>
</tr>
<tr>
<td>pedro_port</td>
<td>150</td>
</tr>
<tr>
<td>pedro_register</td>
<td>146</td>
</tr>
<tr>
<td>pedro_subscribe</td>
<td>146</td>
</tr>
<tr>
<td>pedro_unsubscribe</td>
<td>146</td>
</tr>
<tr>
<td>parallel_sub</td>
<td>69</td>
</tr>
<tr>
<td>phrase</td>
<td>89</td>
</tr>
<tr>
<td>portray</td>
<td>46</td>
</tr>
<tr>
<td>portray_clause</td>
<td>45</td>
</tr>
<tr>
<td>predicate_property</td>
<td>130</td>
</tr>
</tbody>
</table>

170
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>45</td>
</tr>
<tr>
<td>prompt</td>
<td>156</td>
</tr>
<tr>
<td>put</td>
<td>49</td>
</tr>
<tr>
<td>put_char</td>
<td>49</td>
</tr>
<tr>
<td>put_code</td>
<td>49</td>
</tr>
<tr>
<td>put_line</td>
<td>47</td>
</tr>
<tr>
<td>quant</td>
<td>56</td>
</tr>
<tr>
<td>quantifier</td>
<td>67</td>
</tr>
<tr>
<td>quantify</td>
<td>67</td>
</tr>
<tr>
<td>random</td>
<td>85</td>
</tr>
<tr>
<td>read</td>
<td>38</td>
</tr>
<tr>
<td>readR</td>
<td>39</td>
</tr>
<tr>
<td>readR_1_term</td>
<td>40</td>
</tr>
<tr>
<td>read_1_term</td>
<td>39</td>
</tr>
<tr>
<td>realtime</td>
<td>156</td>
</tr>
<tr>
<td>read_term</td>
<td>38</td>
</tr>
<tr>
<td>reconsult</td>
<td>101</td>
</tr>
<tr>
<td>recorda</td>
<td>97</td>
</tr>
<tr>
<td>recorded</td>
<td>97</td>
</tr>
<tr>
<td>recordz</td>
<td>97</td>
</tr>
<tr>
<td>remove_duplicates</td>
<td>78</td>
</tr>
<tr>
<td>remove_global_exception_handler</td>
<td>138</td>
</tr>
<tr>
<td>remove_obvar_prefix</td>
<td>35</td>
</tr>
<tr>
<td>repeat</td>
<td>21</td>
</tr>
<tr>
<td>reset_std_stream</td>
<td>25</td>
</tr>
<tr>
<td>retract</td>
<td>95</td>
</tr>
<tr>
<td>retractall</td>
<td>95</td>
</tr>
<tr>
<td>retry_delays</td>
<td>128</td>
</tr>
<tr>
<td>retry_var_delays</td>
<td>128</td>
</tr>
<tr>
<td>retry_woken_delays</td>
<td>129</td>
</tr>
<tr>
<td>reverse</td>
<td>79</td>
</tr>
<tr>
<td>re_match</td>
<td>66</td>
</tr>
<tr>
<td>same_args</td>
<td>61</td>
</tr>
<tr>
<td>same_handle</td>
<td>146</td>
</tr>
<tr>
<td>schedule_threads_now</td>
<td>144</td>
</tr>
<tr>
<td>search_insert</td>
<td>79</td>
</tr>
<tr>
<td>see</td>
<td>27</td>
</tr>
<tr>
<td>seeing</td>
<td>27</td>
</tr>
<tr>
<td>seen</td>
<td>27</td>
</tr>
<tr>
<td>set_autoflush</td>
<td>28</td>
</tr>
<tr>
<td>set_default_message_thread</td>
<td>150</td>
</tr>
<tr>
<td>set_input</td>
<td>26</td>
</tr>
<tr>
<td>set_obvar_name</td>
<td>36</td>
</tr>
<tr>
<td>set_output</td>
<td>27</td>
</tr>
<tr>
<td>set_std_stream</td>
<td>25</td>
</tr>
<tr>
<td>set_stream_position</td>
<td>28</td>
</tr>
</tbody>
</table>
tcp_open ................................................................. 151
tcp_listen ............................................................. 152
tcp_host_from_ip_address ......................................... 152
tcp_getsockname .................................................... 152
tcp_getpeername .................................................... 152
tcp_connect .......................................................... 151
tcp_close ............................................................. 151
tcp_client .............................................................. 151
tcp_bind ............................................................... 152
subterm.expand.depth.limit ....... 172
substring .............................................................. 65
sub_atom ............................................................... 56
structural.unify ...................................................... 123
sub ................................................................. 56
sub_atom .............................................................. 70
sub_string .......................................................... 65
sub_term ............................................................. 71
substitute ........................................................... 71
substitution .......................................................... 71
subterm.expand.depth.limit ....... 172
stat ................................................................. 23
statistics ........................................................... 130
std_compound ........................................................ 57
std_nonvar .......................................................... 57
std_var .............................................................. 57
stream_property ..................................................... 28
stream_to_atom ..................................................... 29
stream_to_chars .................................................... 30
stream_to_string ................................................... 30
string_to_atom ...................................................... 64
string_concat ........................................................ 65
string_length ......................................................... 65
string_to_list ....................................................... 64
structural_unify ..................................................... 123
sub ................................................................. 56
sub_atom .............................................................. 70
sub_string .......................................................... 65
sub_term ............................................................. 71
substitute ........................................................... 71
substitution .......................................................... 71
subterm.expand.depth.limit ....... 172
tab ................................................................. 48
tcp_accept ........................................................... 152
tcp_bind ............................................................. 151
tcp_client ............................................................ 151
tcp_close ............................................................. 151
tcp_connect .......................................................... 151
tcp_getpeername ..................................................... 152
tcp_getsockname .................................................... 152
tcp_host_from_ip_address .................. 152
tcp_host_to_ip_address ...................... 152
tcp_listen ........................................................... 151
tcp_open ............................................................. 151
tcp_server .............................................. 151
tell ....................................................... 28
telling ................................................... 28
thaw_term ............................................. 125
thaw_var .............................................. 125
thawed_var ........................................... 125
thread_defaults ....................................... 144
thread_errno ......................................... 144
thread_exit ......................................... 142
thread_atomic_goal .................................. 142
thread_fork .......................................... 140
thread_handle ........................................ 146
thread_is_initial_thread ......................... 141
thread_is_thread .................................... 141
thread_push_goal .................................... 142
thread_set_defaults ................................ 144
thread_set_symbol ................................... 139
thread_sleep ......................................... 143
thread_sleep_until .................................. 143
thread_symbol ........................................ 139
thread_wait .......................................... 142
thread_wait_on_goal ................................ 143
thread_yield .......................................... 144
throw .................................................. 21
told .................................................... 28
trace ................................................... 107
transform_simple_terms ............................. 119
transform_subterms .................................. 120
true ....................................................... 18
type_exception ....................................... 136
unify_with_occurs_check ......................... 123
uncurry ............................................... 72
union_list .......................................... 74
unwind_protect ...................................... 22
update_predicate_timestamp ....................... 94
var ...................................................... 54
with_debugging_off .................................. 108
with_local_exception_handler ..................... 137
working_directory .................................. 155
write ................................................... 41
writeR .................................................. 42
writeRT ............................................... 42
writeRTq ............................................... 42
writeRq ............................................... 42
writeT .................................................. 42
writeTq ................................................................. 42
write_atom ............................................................ 42
write_canonical ....................................................... 43
write_integer .......................................................... 43
write_string ........................................................... 43
write_term ............................................................... 41
write_term_list ........................................................ 44
writeq ................................................................. 42
writeq_atom ........................................................... 42
writeq_string .......................................................... 43