GNU PROLOG

A Native Prolog Compiler with Constraint Solving over Finite Domains
Edition 1.0, for GNU Prolog version 1.0.0
April 18, 1999

by Daniel Diaz
Copyright (C) 1999 Daniel Diaz

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation¹, 59 Temple Place - Suite 330, Boston, MA 02111, USA.

¹http://www.fsf.org/
Acknowledgements

I would like to thank the department of computing science\textsuperscript{2} at the university of Paris 1 for allowing me the time and freedom necessary to achieve this project.

I am grateful to the members of the Loco project\textsuperscript{3} at INRIA Rocquencourt\textsuperscript{4} for their encouragement. Their involvement in this work led to useful feedback and exchange.

I would particularly like to thank Jonathan Hodgson\textsuperscript{5} for the time and effort he put into the proofreading of this manual. His suggestions, both regarding ISO technical aspects as well as the language in which it was expressed, proved invaluable.

The in-line HTML version of this document was created using H\textsuperscript{\textregistered}VA\textsuperscript{6} developed by Luc Maranget who kindly devoted so much of his time extending the capabilities of H\textsuperscript{\textregistered}VA in order to handle such a sizeable manual.

Jean-Christophe Aude kindly improved the visual aspect of both the illustrations and the GNU Prolog web pages.

Thanks to Richard A. O’Keefe for his advice regarding the implementation of some Prolog built-in predicates and for suggesting me the in-place installation feature.

Many thanks to all those people at GNU\textsuperscript{7} who helped me to finalize the GNU Prolog project.

Finally, I would like to thank everybody who tested preliminary releases and helped me to put the finishing touches to this system.

\textsuperscript{2}http://panoramix.univ-paris1.fr/CRINF
\textsuperscript{3}http://loco.inria.fr/
\textsuperscript{4}http://www.inria.fr/Unites/ROCUENCOURT-eng.html
\textsuperscript{5}http://www.sju.edu/~jhodgson
\textsuperscript{6}http://para.inria.fr/~maranget/heva
\textsuperscript{7}http://www.gnu.org
1 Introduction

GNU Prolog is a free Prolog compiler with constraint solving over finite domains developed by Daniel Diaz.

For recent information about GNU Prolog please consult the GNU Prolog page.

GNU Prolog is a Prolog compiler based on the Warren Abstract Machine (WAM) [8, 1]. It first compiles
a Prolog program to a WAM file which is then translated to a low-level machine independent language
called mini-assembly specifically designed for GNU Prolog. The resulting file is then translated to the
assembly language of the target machine (from which an object is obtained). This allows GNU Prolog
to produce a native stand alone executable from a Prolog source (similarly to what does a C compiler
from a C program). The main advantage of this compilation scheme is to produce native code and to be
fast. Another interesting feature is that executables are small. Indeed, the code of most unused built-in
predicates is not included in the executables at link-time.

A lot of work has been devoted to the ISO compatibility. Indeed, GNU Prolog is very close to the ISO
standard for Prolog [5].

GNU Prolog also offers various extensions very useful in practice (global variables, OS interface, sockets,...). In particular, GNU Prolog contains an efficient constraint solver over Finite Domains (FD). This
opens constraint logic programming to the user combining the power of constraint programming to the
decarativity of logic programming. The key feature of the GNU Prolog solver is the use of a single (low-
level) primitive to define all (high-level) FD constraints. There are many advantages of this approach:
constraints can be compiled, the user can define his own constraints (in terms of the primitive), the solver
is open and extensible (as opposed to black-box solvers like CHIP),... Moreover, the GNU Prolog solver
is rather efficient, often more than commercial solvers.

GNU Prolog is inspired from two systems developed by the same author:

- **wamcc**: a Prolog to C compiler [3]. the key point of wamcc was its ability to produce stand alone
  executables using an original compilation scheme: the translation of Prolog to C via the WAM.
  Its drawback was the time needed by gcc to compile the produced sources. GNU Prolog can also
  produce stand alone executables but using a faster compilation scheme.

- **clp(FD)**: a constraint programming language over FD [4]. Its key feature was the use of a single
  primitive to define FD constraints. GNU Prolog is based on the same idea but extends the power
  of primitive to make it possible more complex constraint definitions. In comparison to clp(FD),
  GNU Prolog offers new predefined constraints, new predefined heuristics, reified constraints,...

Here are some features of GNU Prolog:

- **Prolog system**:
  - conforms to the ISO standard for Prolog (floating point numbers, streams, dynamic code,...).
  - a lot of extensions: global variables, definite clause grammars (DCG), sockets interface, oper-
    ating system interface,...
  - more than 300 Prolog built-in predicates.
  - Prolog debugger and a low-level WAM debugger.
  - line editing facility under the interactive interpreter with completion on atoms.
  - powerful bidirectional interface between Prolog and C.

- **Compiler**:
  - native-code compiler producing stand alone executables.
  - simple command-line compiler accepting a wide variety of files: Prolog files, C files, WAM
    files,...

---

http://pasillac.inria.fr/~diaz
http://www.gnu.org/software/prolog
http://www.logic-programming.org/prolog_std.html
- direct generation of assembly code 15 times faster than wamcc + gcc.
- most of unused built-in predicates are not linked (to reduce the size of the executables).
- compiled predicates (native-code) as fast as wamcc on average.
- consulted predicates (byte-code) 5 times faster than wamcc.

- Constraint solver:
  - FD variables well integrated into the Prolog environment (full compatibility with Prolog variables and integers). No need for explicit FD declarations.
  - very efficient FD solver (comparable to commercial solvers).
  - high-level constraints can be described in terms of simple primitives.
  - a lot of predefined constraints: arithmetic constraints, boolean constraints, symbolic constraints, reified constraints,.
  - several predefined enumeration heuristics.
  - the user can define his own new constraints.
  - more than 50 FD built-in constraints/predicates.
2 Using GNU Prolog

2.1 Introduction

GNU Prolog offers two ways to execute a Prolog program:

- interpreting it using the GNU Prolog interactive interpreter.
- compiling it to a (machine-dependent) executable using the GNU Prolog native-code compiler.

Running a program under the interactive interpreter allows the user to list it and to make full use of the debugger on it (section 3, page 18). Compiling a program to native code makes it possible to obtain a stand-alone executable, with a reduced size and optimized for speed. Running a Prolog program compiled to native code is around 3-5 times faster than running it under the interpreter. However, it is not possible to make full use of the debugger on a program compiled to native code. Nor is it possible to list the program. In general, it is preferable to run a program under the interpreter for debugging and then use the native-code compiler to produce an autonomous executable. It is also possible to combine these two modes by producing an executable that contains some parts of the program (e.g., already debugged predicates whose execution-time speed is crucial) and interpreting the other parts under this executable. In that case, the executable has the same facilities as the GNU Prolog interpreter but also integrates the native-code predicates. This way to define a new enriched interpreter is detailed later (section 2.4.5, page 15).

2.2 The GNU Prolog interactive interpreter

2.2.1 Starting/exiting the interactive interpreter

GNU Prolog offers a classical Prolog interactive interpreter also called top-level. It allows the user to execute queries, to consult Prolog programs, to list them, to execute them and to debug them. The top-level can be invoked using the following command:

% gprolog

(the % symbol is the operating system shell prompt)

The top-level invocation does not need any supplemental line-command option. When present these options can be retrieved using argument_value/2 (section 6.27.2, page 127) or argument_list/1 (section 6.27.3, page 128). However, the --verbose and --help options are recognized for compatibility purpose with other GNU programs.

To quit the top-level type the end-of-file key sequence (Ctrl-D) or its term representation: end_of_file. It is also possible to use the built-in predicate halt/0 (section 6.18.1, page 99).

2.2.2 The interactive interpreter read-execute-write loop

The GNU Prolog top-level is built on a classical read-execute-write loop that also allows for re-executions (when the query is not deterministic) as follows:

- display the prompt, i.e. '| ?-'.
- read a query (i.e. a goal).
- execute the query.
- in case of success display the values of the variables of the query.
• if there are remaining alternatives (i.e. the query is not deterministic), display a ? and ask the user who can use one of the following commands: RETURN to stop the execution, ; to compute the next solution or a to compute all remaining solution.

Here is an example of execution of a query ("find the lists X and Y such that the concatenation of X and Y is [a,b]"): 

\[
| \text{?- append(X,Y,[a,b,c]).} \\
X = [] \\
Y = [a,b,c] ? \text{ (here the user presses ; to compute another solution)} \\
X = [a] \\
Y = [b,c] ? \text{ (here the user presses a to compute all remaining solutions)} \\
X = [a,b] \\
Y = [c] \text{ (here the user is not asked and the next solution is computed)} \\
X = [a,b,c] \\
Y = [] \text{ (here the user is not asked and the next solution is computed)} \\
\text{no \ (no more solution)}
\]

In some cases the top-level can detect that the current solution is the last one (no more alternatives remaining). In such a case it does not display the ? symbol (and does not ask the user). Example:

\[
| \text{?- (X=1 ; X=2).} \\
X = 1 ? \text{ (here the user presses ; to compute another solution)} \\
X = 2 \text{ (here the user is not prompted since there are no more alternatives)} \\
\text{yes}
\]

The user can stop the execution even if there are more alternatives by typing RETURN.

\[
| \text{?- (X=1 ; X=2).} \\
X = 1 ? \text{ (here the user presses RETURN to stop the execution)} \\
\text{yes}
\]

The top-level tries to display the values of the variables of the query in a readable manner. For instance, when a variable is bound to a query variable, the name of this variable appears. When a variable is a singleton an underscore symbol _ is displayed (_ is a generic name for a singleton variable, it is also called an anonymous variable). Other variables are bound to new brand variable names. When a query variable name X appears as the value of another query variable Y it is because X is itself not instantiated otherwise the value of X is displayed. In such a case, nothing is output for X itself (since it is a variable). Example:

\[
| \text{?- X=f(A,B,_,A), A=k.} \\
A = k \text{ (the value of A is displayed also in f/3 for X)} \\
X = f(k,B,\_,k) \text{ (since B is a variable which is also a part of X, B is not displayed)} \\
\text{?- functor(T,f,3), arg(1,T,X), arg(3,T,X).} \\
T = f(X,\_,X) \text{ (the 1^{st} and 3^{rd} args are equal to X, the 2^{nd} is an anonymous variable)}
\]
| ?- read_from_atom('k(X,Y,X).',T).

    T = k(A,_,A)          (the 1st and 3rd args are unified, a new variable name \( A \) is introduced)

The top-level uses variable binding predicates (section 6.5, page 43). To display the value of a variable, the top-level calls `write_term/3` with the following option list: `[quoted(true), numbervars(false), namevars(true)]` (section 6.14.6, page 83). A term of the form `'$VARNAME'(Name)` where `Name` is an atom is displayed as a variable name while a term of the form `'$VAR'(N)` where \( N \) is an integer is displayed as a normal compound term (such a term could be output as a variable name by `write_term/3`). Example:

| ?- X='$VARNAME'(Y), Y='$VAR'(1).

    X = Y                (the term `'$VARNAME'(Y)` is displayed as \( Y \))
    Y = '$VAR'(1)        (the term `'$VAR'(1)` is displayed as \( 1 \))

| ?- X=Y, Y='$VAR'(1).

    X = '$VAR'(1)
    Y = '$VAR'(1)

In the first example, \( X \) is explicitly bound to `'$VARNAME'(Y)` by the query so the top-level displays \( Y \) as the value of \( X \). \( Y \) is unified with `'$VAR'(1)` so the top-level displays it as a normal compound term. It should be clear that \( X \) is not bound to \( Y \) (whereas it is in the second query). This behavior should be kept in mind when doing variable binding operations.

Finally, the top-level computes the user-time (section 6.24.2, page 123) taken by a query and displays it when it is significant. Example:

| ?- retractall(p(_)), assertz(p(0)),
    repeat,
    retract(p(X)),
    Y is X+1,
    assertz(p(Y)),
    X=1000, !.

    X = 1000
    Y = 1001

    (180 ms) yes          (the query took 180ms of user time)

2.2.3 Consulting a Prolog program

The top-level allows the user to consult Prolog source files. Consulted predicates can be listed, executed and debugged (while predicates compiled to native-code cannot). For more information about the difference between a native-code predicate and a consulted predicate refer to the introduction of this section (section 2.1, page 4) and to the part devoted to the compiler (section 2.4.1, page 10).

To consult a program use the built-in predicate `consult/1` (section 6.23.1, page 120). The argument of this predicate is a Prolog file name or `user` to specify the terminal. This allows the user to directly input the predicates from the terminal. In that case the input shall be terminated by the end-of-file key sequence (Ctrl-D) or its term representation: `end_of_file`. A shorthand for `consult(FILE)` is `[FILE]`. Example:
2.2 The GNU Prolog interactive interpreter

<table>
<thead>
<tr>
<th>Command</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>abort</td>
<td>abort the current execution. Same as abort/0 (section 6.18.1, page 99)</td>
</tr>
<tr>
<td>e</td>
<td>exit</td>
<td>quit the current Prolog process. Same as halt/0 (section 6.18.1, page 99)</td>
</tr>
<tr>
<td>b</td>
<td>break</td>
<td>invoke a recursive top-level. Same as break/0 (section 6.18.1, page 99)</td>
</tr>
<tr>
<td>c</td>
<td>continue</td>
<td>resume the execution</td>
</tr>
<tr>
<td>t</td>
<td>trace</td>
<td>start the debugger using trace/0 (section 3.3.1, page 18)</td>
</tr>
<tr>
<td>d</td>
<td>debug</td>
<td>start the debugger using debug/0 (section 3.3.1, page 18)</td>
</tr>
<tr>
<td>h or ?</td>
<td>help</td>
<td>display a summary of available commands</td>
</tr>
</tbody>
</table>
2.2.5 The line editor

The line editor (linedit) allows the user to build/update the current input line using a variety of commands. This facility is available if the linedit part of GNU Prolog has been installed. linedit is implicitly called by any built-in predicate reading from a terminal (e.g. get_char/1, read/1,...). This is the case when the top-level reads a query.

**Bindings:** each command of linedit is activated using a key. For some commands another key is also available to invoke the command (on some terminals this other key may not work properly while the primary key always works). Here is the list of available commands:

<table>
<thead>
<tr>
<th>Key</th>
<th>Alternate key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctl-B</td>
<td>←</td>
<td>go to the previous character</td>
</tr>
<tr>
<td>Ctl-F</td>
<td>→</td>
<td>go to the next character</td>
</tr>
<tr>
<td>Esc-B</td>
<td>Ctl-←</td>
<td>go to the previous word</td>
</tr>
<tr>
<td>Esc-F</td>
<td>Ctl-→</td>
<td>go to the next word</td>
</tr>
<tr>
<td>Ctl-A</td>
<td>Home</td>
<td>go to the beginning of the line</td>
</tr>
<tr>
<td>Ctl-E</td>
<td>End</td>
<td>go to the end of the line</td>
</tr>
<tr>
<td>Ctl-H</td>
<td>Backspace</td>
<td>delete the previous character</td>
</tr>
<tr>
<td>Ctl-D</td>
<td>Delete</td>
<td>delete the current character</td>
</tr>
<tr>
<td>Ctl-U</td>
<td>Ctl-Home</td>
<td>delete from beginning of the line to the current character</td>
</tr>
<tr>
<td>Ctl-K</td>
<td>Ctl-End</td>
<td>delete from the current character to the end of the line</td>
</tr>
<tr>
<td>Esc-L</td>
<td></td>
<td>lower case the next word</td>
</tr>
<tr>
<td>Esc-U</td>
<td></td>
<td>upper case the next word</td>
</tr>
<tr>
<td>Esc-C</td>
<td></td>
<td>capitalize the next word</td>
</tr>
<tr>
<td>Ctl-T</td>
<td></td>
<td>exchange last two characters</td>
</tr>
<tr>
<td>Ctl-V</td>
<td>Insert</td>
<td>switch on/off the insert/replace mode</td>
</tr>
<tr>
<td>Ctl-I</td>
<td>Tab</td>
<td>complete word (twice displays all possible completions)</td>
</tr>
<tr>
<td>Ctl-space</td>
<td></td>
<td>mark beginning of the selection</td>
</tr>
<tr>
<td>Esc-W</td>
<td>Tab</td>
<td>copy (from the begin selection mark to the current character)</td>
</tr>
<tr>
<td>Ctl-W</td>
<td></td>
<td>cut (from the begin selection mark to the current character)</td>
</tr>
<tr>
<td>Ctl-Y</td>
<td></td>
<td>paste</td>
</tr>
<tr>
<td>Ctl-P</td>
<td>↑</td>
<td>recall previous history line</td>
</tr>
<tr>
<td>Ctl-N</td>
<td>↓</td>
<td>recall next history line</td>
</tr>
<tr>
<td>Esc-P</td>
<td></td>
<td>recall previous history line beginning with the current prefix</td>
</tr>
<tr>
<td>Esc-N</td>
<td></td>
<td>recall next history line beginning with the current prefix</td>
</tr>
<tr>
<td>Esc-&lt;</td>
<td>Page Up</td>
<td>recall first history line</td>
</tr>
<tr>
<td>Esc-&gt;</td>
<td>Page Down</td>
<td>recall last history line</td>
</tr>
<tr>
<td>Ctl-C</td>
<td></td>
<td>generate an interrupt signal (section 2.2.4, page 7)</td>
</tr>
<tr>
<td>Ctl-D</td>
<td></td>
<td>generate an end-of-file character (at the begin of the line)</td>
</tr>
<tr>
<td>RETURN</td>
<td></td>
<td>validate a line</td>
</tr>
<tr>
<td>Esc-?</td>
<td></td>
<td>display a summary of available commands</td>
</tr>
</tbody>
</table>

**History:** when a line is entered (i.e. terminated by RETURN), linedit records it in an internal list called history. It is later possible to recall history lines using appropriate commands (e.g. Ctl-P recall the last entered line) and to modify them as needed. It is also possible to recall a history line beginning with a given prefix. For instance to recall the previous line beginning with write simply type write followed by Esc-P. Another Esc-P will recall an earlier line beginning with write,...

**Completion:** another important feature of linedit is its completion facility. Indeed, linedit maintains a list of known words and uses it to complete the prefix of a word. Initially this list contains all predefined atoms and the atoms corresponding to available predicates. This list is dynamically updated when a new atom appears in the system (whether read at the top-level, created with a built-in predicate, associated to a new consulted predicate,...). When the completion key (Tab) is pressed linedit acts as follows:
2.3 Adjusting the size of Prolog stacks

- use the current word as a prefix.
- collect all words of the list that begin with this prefix.
- complete the current word with the longest common part of all matching words.
- if more than one word matches emit a beep (a second Tab will display all possibilities).

Example:

```
| ?- argu               (here the user presses Tab to complete the word) |
| ?- argument_          (lindedit completes argu with argument_ and emits a beep) |
| argument_counter      (lindedit completes with argument_counter)        |
| argument_list         (lindedit shows 3 possible completions)             |
| argument_value        (lindedit redisplays the input line)               |
| ?- argument_C         (to select argument_counter the user presses c and Tab) |
| ?- argument_Counter   (lindedit completes with argument_counter)         |
```

Finally, lindedit allows the user to check that (square/curly) brackets are well balanced. For this, when a close bracket symbol, i.e., `)` or `]`, is typed, lindedit determines the associated open bracket, i.e., `(` or `{, and temporarily repositions the cursor on it to show the match.

2.3 Adjusting the size of Prolog stacks

GNU Prolog uses several stacks to execute a Prolog program. Each stack has a static size and cannot be dynamically increased during the execution. For each stack there is a default size but the user can define a new size by setting an environment variable. When a GNU Prolog program is run it first consults these variables and if they are not defined uses the default sizes. The following table presents each stack of GNU Prolog with its default size and the name of its associated environment variable:

<table>
<thead>
<tr>
<th>Stack name</th>
<th>Default size (Kb)</th>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>2048</td>
<td>LOCALSZ</td>
<td>control stack (environments and choice-points)</td>
</tr>
<tr>
<td>global</td>
<td>4096</td>
<td>GLOBALSZ</td>
<td>heap (compound terms)</td>
</tr>
<tr>
<td>trail</td>
<td>2048</td>
<td>TRAILSZ</td>
<td>conditional bindings (bindings to undo at backtracking)</td>
</tr>
<tr>
<td>cstr</td>
<td>2048</td>
<td>CSTRSZ</td>
<td>finite domain constraint stack (FD variables and constraints)</td>
</tr>
</tbody>
</table>

If the size of a stack is too small an overflow will occur during the execution. In that case GNU Prolog emits the following error message before stopping:

```
S stack overflow (size: N Kb, environment variable used: E)
```

where $S$ is the name of the stack, $N$ is the current stack size in Kb and $E$ the name of the associated environment variable. When such a message occurs it is possible to (re)define the variable $E$ with the new size. For instance to allocate 4096 Kb to the local stack under a Unix shell use:

```
LOCALSZ=4096; export LOCALS             (under sh or bash)
setenv LOCALSZ 4096                      (under csh or tcsh)
```

This method allows the user to adjust the size of Prolog stacks. However, in some cases it is preferable not to allow the user to modify these sizes. For instance, when providing a stand alone executable whose behavior should be independent of the environment in which it is run. In that case the program should not consult environment variables and the programmer should be able to define new default stack sizes.
The GNU Prolog compiler offers this facilities via several command-line options such as --local-size or --fixed-sizes (section 2.4.3, page 12).

Finally note that GNU Prolog stacks are virtually allocated (i.e. use virtual memory). This means that a physical memory page is allocated only when needed (i.e. when an attempt to read/write it occurs). Thus it is possible to define very large stacks. At the execution, only the needed amount of space will be physically allocated.

2.4 The GNU Prolog compiler

2.4.1 Different kinds of codes

One of the main advantages of GNU Prolog is its ability to produce stand alone executables. A Prolog program can be compiled to native code to give rise to a machine-dependent executable using the GNU Prolog compiler. However native-code predicates cannot be listed nor fully debugged. So there is an alternative to native-code compilation: byte-code compilation. By default the GNU Prolog compiler produces native-code but via a command-line option it can produce a file ready for byte-code loading. This is exactly what consult/1 does as was explained above (section 2.2.3, page 6). GNU Prolog also manages interpreted code using a Prolog interpreter written in Prolog. Obviously interpreted code is slower than byte-code but does not require the invocation of the GNU Prolog compiler. This interpreter is used each time a meta-call is needed as by call/1 (section 5.2.3, page 35). This also the case of dynamically asserted clauses. The following table summarizes these three kinds of codes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Speed</th>
<th>Debug?</th>
<th>For what</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpreted-code</td>
<td>slow</td>
<td>yes</td>
<td>meta-call and dynamically asserted clauses</td>
</tr>
<tr>
<td>byte-code</td>
<td>medium</td>
<td>yes</td>
<td>consulted predicates</td>
</tr>
<tr>
<td>native-code</td>
<td>fast</td>
<td>no</td>
<td>compiled predicates</td>
</tr>
</tbody>
</table>

2.4.2 Compilation scheme

Native-code compilation: a Prolog source is compiled in several stages to produce an object file that is linked to the GNU Prolog libraries to produce an executable. The Prolog source is first compiled to obtain a WAM [8] file. For a detailed study of the WAM the interested reader can refer to “Warren’s Abstract Machine: A Tutorial Reconstruction”[11] [1]. The WAM file is translated to a machine-independent language specifically designed for GNU Prolog. This language is close to a (universal) assembly language and is based on a very reduced instruction set. For this reason this language is called mini-assembly (MA). The mini-assembly file is then mapped to the assembly language of the target machine. This assembly file is assembled to give rise to an object file which is then linked with the GNU Prolog libraries to provide an executable. The compiler also takes into account Finite Domain constraint definition files. It translates them to C and invoke the C compiler to obtain object files. The following figure presents this compilation scheme:

Obviously all intermediate stages are hidden to the user who simply invokes the compiler on his Prolog file(s) (plus other files: C, ...) and obtains an executable. However, it is also possible to stop the compiler at any given stage. This can be useful, for instance, to see the WAM code produced (perhaps when learning the WAM). Finally it is possible to give any kind of file to the compiler which will insert it in the compilation chain at the stage corresponding to its type. The type of a file is determined using the suffix of its file name. The following table presents all recognized types/suffixes:

<table>
<thead>
<tr>
<th>Suffix of the file</th>
<th>Type of the file</th>
<th>Handled by</th>
</tr>
</thead>
<tbody>
<tr>
<td>.pl, .pro</td>
<td>Prolog source file</td>
<td>p12wam</td>
</tr>
<tr>
<td>.wam</td>
<td>WAM source file</td>
<td>wam2wam</td>
</tr>
<tr>
<td>.ma</td>
<td>Mini-assembly source file</td>
<td>ma2asm</td>
</tr>
<tr>
<td>.s</td>
<td>Assembly source file</td>
<td>the assembler</td>
</tr>
<tr>
<td>.c, .C, .CC, .cc, .cxx, .c++, .cpp</td>
<td>C or C++ source file</td>
<td>the C compiler</td>
</tr>
<tr>
<td>.fd</td>
<td>Finite Domain constraint source file</td>
<td>fd2c</td>
</tr>
<tr>
<td>any other suffix (.o, .a, ...)</td>
<td>any other type (object, library,...)</td>
<td>the linker (C linker)</td>
</tr>
</tbody>
</table>

**Byte-code compilation:** the same compiler can be used to compile a source Prolog file for byte-code. In that case the Prolog to WAM compiler is invoked using a specific option and produces a WAM for byte-code source file (suffixed .wbc) that can be later loaded using `load/1` (section 6.23.2, page 121). Note that this is exactly what `consult/1` (section 6.23.1, page 120) does as explained above (section 2.2.3, page 6).
2.4.3 Using the compiler

The GNU Prolog compiler is a command-line compiler similar in spirit to a Unix C compiler like gcc. 
To invoke the compiler use the gplc command as follows:

```
% gplc [OPTION]... FILE...  (the % symbol is the operating system shell prompt)
```

The arguments of gplc are file names that are dispatched in the compilation scheme depending on the type determined from their suffix as was explained previously (section 2.4.2, page 10). All object files are then linked to produce an executable. Note however that GNU Prolog has no module facility (since there is not yet an ISO reference for Prolog modules) thus a predicate defined in a Prolog file is visible from any other predicate defined in any other file. GNU Prolog allows the user to split a big Prolog source into several files but does not offer any way to hide a predicate from others.

The simplest way to obtain an executable from a Prolog source file prog.pl is to use:

```
% gplc prog.pl
```

This will produce an native executable called prog which can be executed as follows:

```
% prog
```

However, there are several options that can be used to control the compilation:

**General options:**

- `-o FILE, --output FILE` use `FILE` as the name of the output file
- `-W, --wam-for-native` stop after producing WAM files(s)
- `-w, --wam-for-byte-code` stop after producing WAM for byte-code file(s) (force --no-call-c)
- `-M, --mini-assembly` stop after producing mini-assembly files(s)
- `-S, --assembly` stop after producing assembly files (s)
- `-F, --fd-to-c` stop after producing C files(s) from FD constraint definition file(s)
- `-c, --object` stop after producing object files(s)
- `--temp-dir PATH` use `PATH` as directory for temporary files
- `--no-del-temp` do not delete temporary files
- `--no-decode-hexa` do not decode hexadecimal predicate names
- `--verbose` print executed commands
- `--help` print a help and exit
- `--version` print version number and exit

**Prolog to WAM compiler options:**

- `--pl-state FILE` read `FILE` to set the initial Prolog state
- `--no-inline` do not inline predicates
- `--no-reorder` do not reorder predicate arguments
- `--no-reg-opt` do not optimize registers
- `--min-reg-opt` minimally optimize registers
- `--no-opt-last-subterm` do not optimize last subterm compilation
- `--fast-math` use fast mathematical mode (assume integer arithmetic)
- `--keep-void-inst` keep void WAM instructions in the output file
- `--no-susp-warn` do not show warnings for suspicious predicates
- `--no-singl-warn` do not show warnings for named singleton variables
- `--no-redef-error` no show errors for built-in predicate redefinitions
- `--no-call-c` do not allow the use of `fd_tell`, `$call.C`, ...
- `--compile-msg` print a compile message
- `--statistics` print statistics information

**WAM to mini-assembly translator options:**
--comment include comments in the output file

Mini-assembly to assembly translator options:
--comment include comments in the output file

C compiler options:
--c-compiler FILE use FILE as C compiler
-C OPTION pass OPTION to the C compiler

Assembler options:
-A OPTION pass OPTION to the assembler

Linker options:
--local-size N set default local stack size to N Kb
--global-size N set default global stack size to N Kb
--trail-size N set default trail stack size to N Kb
--cstr-size N set default constraint stack size to N Kb
--fixed-sizes do not consult environment variables at run-time (use default sizes)
--no-top-level do not link the top-level (force --no-debugger)
--no-debugger do not link the Prolog/WAM debugger
--min-pl-bips link only used Prolog built-in predicates
--min-fd-bips link only used FD solver built-in predicates
--min-bips shorthand for: --no-top-level --min-pl-bips --min-fd-bips
--min-size shorthand for: --min-bips --strip
--no-fd-lib do not look for the FD library (maintenance only)
-s, --strip strip the executable
-L OPTION Pass OPTION to the linker

It is possible to only give the prefix of an option if there is no ambiguity.

The name of the output file is controlled via the -o FILE option. If present the output file produced will be named FILE. If not specified, the output file name depends on the last stage reached by the compiler.

If the link is not done the output file name(s) is the input file name(s) with the suffix associated to the last stage. If the link is done, the name of the executable is the name (without suffix) of the first file name encountered in the command-line. Note that if the link is not done -o should be used if only one file name is given as argument.

By default the compiler runs in the native-code compilation scheme. To generate a WAM file for byte-code use the --wam-for-byte-code option. The resulting file can then be loaded using load/1 (section 6.23.2, page 121).

To execute the Prolog to WAM compiler in a given read environment (operator definitions, character conversion table,...) use --pl-state FILE. The state file should be produced by write.pl.state.file/1 (section 6.22.5, page 120).

By default the Prolog to WAM compiler inlines calls to some deterministic built-in predicates (e.g. arg/3 and functor/3). Namely a call to such a predicate will not yield a classical predicate call but a simple C function call (which is obviously faster). It is possible to avoid this using --no-inline.

Another optimization performed by the Prolog to WAM compiler is unification reordering. The arguments of a predicate are reordered to optimize unification. This can be deactivated using --no-reorder. The compiler also optimizes the unification/loading of nested compound terms. More precisely, the compiler emits optimized instructions when the last subterm of a compound term is itself a compound term (e.g.
lists). This can be deactivated using --no-opt-last-subterm.

By default the Prolog to WAM compiler fully optimizes the allocation of registers to decrease both the number of instruction produced and the number of used registers. A good allocation will generate many \textit{void instructions} that are removed from the produced file except if \texttt{--keep-void-inst} is specified. To prevent any optimization use \texttt{--no-reg-opt} while \texttt{--min-reg-opt} forces the compiler to only perform simple register optimizations.

The Prolog to WAM compiler emits an error when a control construct or a built-in predicate is redefined. This can be avoided using \texttt{--no-redef-error}. The compiler also emits warnings for suspicious predicate definitions like \texttt{-/2} since this often corresponds to an earlier syntax error (e.g. \_ instead of \_). This can be deactivated by specifying \texttt{--no-susp-warn}. Finally, the compiler warns when a singleton variable has a name (i.e. not the generic anonymous name \_). This can be deactivated specifying \texttt{--no-singl-warn}.

Predicate names are encoded with an hexadecimal representation. This is explained in more detail later (section 2.4.6, page 16). By default the error messages from the linker (e.g. multiple definitions for a given predicate, reference to an undefined predicate,...) are filtered to replace any hexadecimal representation by the real predicate name. Specifying the \texttt{--no-decode-hex} prevents \texttt{gplc} from filtering linker output messages and hexadecimal representations are then shown.

When producing an executable it is possible to specify default stack sizes (using \texttt{--STACK_NAME-size}) and to prevent it from consulting environment variables (using \texttt{--fixed-sizes}) as was explained above (section 2.3, page 9). By default the produced executable will include the top-level, the Prolog/WAM debugger and all Prolog and FD built-in predicates. It is possible to avoid linking the top-level (section 2.2, page 4) by specifying \texttt{--no-top-level}. In this case, at least one initialization/1 directive (section 5.1.12, page 33) should be defined. The option \texttt{--no-debugger} does not link the debugger. To include only used built-in predicates that are actually used the options \texttt{--no-pl-bips} and/or \texttt{--no-fd-bips} can be specified. For the smallest executable all these options should be specified. This can be abbreviated by using the shorthand option \texttt{--min-bips}. By default, executables are not \textit{stripped}, i.e. their symbol table is not removed. This table is only useful for the C debugger (e.g. when interfacing Prolog and C). To remove the symbol table (and then to reduce the size of the final executable) use \texttt{--strip}. Finally \texttt{--min-size} is a shortcut for \texttt{--min-bips} and \texttt{--strip}, i.e. the produced executable is as small as possible.

Example: compile and link two Prolog sources \texttt{prog1.pl} and \texttt{prog2.pl}. The resulting executable will be named \texttt{prog1} (since \texttt{-o} is not specified):

\begin{verbatim}
% gplc prog1.pl prog2.pl
\end{verbatim}

Example: compile the Prolog file \texttt{prog.pl} to study basic WAM code. The resulting file will be named \texttt{prog.wam}:

\begin{verbatim}
% gplc -w --no-inline --no-reorder --keep-void-inst prog.pl
\end{verbatim}

Example: compile the Prolog file \texttt{prog.pl} and its C interface file \texttt{utils.c} to provide an autonomous executable called \texttt{mycommand}. The executable is not stripped to allow the use of the C debugger:

\begin{verbatim}
% gplc -o mycommand prog.pl utils.c
\end{verbatim}

Example: detail all steps to compile the Prolog file \texttt{prog.pl} (the resulting executable is stripped). All intermediate files are produced (\texttt{prog.wam}, \texttt{prog.ma}, \texttt{prog.s}, \texttt{prog.o} and the executable \texttt{prog}):

\begin{verbatim}
% gplc -w prog.pl
% gplc -M --comment prog.wam
% gplc -S --comment prog.ma
% gplc -c prog.s
% gplc -o prog -s prog.o
\end{verbatim}
2.4.4 Running an executable

In this section we explain what happens when running an executable produced by the GNU Prolog native-code compiler. The default main function first starts the Prolog engine. This function collects all linked objects (issued from the compilation of Prolog files) and initializes them. The initialization of a Prolog object file consists in adding to appropriate tables new atoms, new predicates and executing its system directives. A system directive is generated by the Prolog to WAM compiler to reflect a (user) directive executed at compile-time such as op/3 (section 5.1.9, page 32). Indeed, when the compiler encounters such a directive it immediately executes it and also generates a system directive to execute it at the start of the executable. When all system directives have been executed the Prolog engine executes all initialization directives defined with initialization/1 (section 5.1.12, page 33). If several initialization directives appear in the same file they are executed in the order of appearance. If several initialization directives appear in different files the order in which they are executed is machine-dependant. However, on most machines the order will be the order in which the associated files have been linked. When all initialization directives have been executed the default main function looks for the GNU Prolog top-level. If present (i.e. it has been linked) it is called otherwise the program simply ends. Note that if the top-level is not linked and if there is no initialization directive the program is useless since it simply ends without doing any work. The default main function detects such a behavior and emits a warning message.

Example: compile an empty file prog.pl without linking the top-level and execute it:

    % gplc --no-top-level prog.pl
    % prog
    Warning: no initial goal executed
    use a directive :- initialization(Goal)
    or remove the link option --no-top-level (or --min-bips or --min-size)

2.4.5 Generating a new interactive interpreter

In this section we show how to define a new top-level extending the GNU Prolog interactive interpreter with new predicate definitions. The obtained top-level can then be considered as an enriched version of the basic GNU Prolog top-level (section 2.2, page 4). Indeed, each added predicate can be viewed as a predefined predicate just like any other built-in predicate. This can be achieved by compiling these predicates and including the top-level at link-time.

The real question is: why would we include some predicates in a new top-level instead of simply consulting them under the GNU Prolog top-level? There are two reasons for this:

- the predicate cannot be consulted. This is the case of a predicate calling foreign code, like a predicate interfacing with C (section 8, page 165) or a predicate defining a new FD constraint.

- the performance of the predicate is crucial. Since it is compiled to native-code such a predicate will be executed very quickly. Consulting will load it as byte-code. The gain is much more noticeable if the program is run under the debugger. The included version will not be affected by the debugger while the consulted version will be several times slower. Obviously, a predicate should be included in a new top-level only when it is itself debugged since it is difficult to debug native-code.

To define a new top-level simply compile the set of desired predicates and linking them with the GNU Prolog top-level (this is the default) using gplc (section 2.4.3, page 12).

Example: let us define a new top-level called my_top_level including all predicates defined in prog.pl:

    % gplc -o my_top_level prog.pl

By the way, note that if prog.pl is an empty Prolog file the previous command will simply create a new interactive interpreter similar to the GNU Prolog top-level.
Example: as before where some predicates of progl.pl call C functions defined in utils.c:

\[
% \text{gplc -o my\_top\_level progl.pl utils.c}
\]

In conclusion, defining a particular top-level is nothing else but a particular case of the native-code compilation. It is simple to do and very useful in practice.

2.4.6 The hexadecimal predicate name encoding

When the GNU Prolog compiler compiles a Prolog source to an object file it has to associate a symbol to each predicate name. However, the syntax of symbols is restricted to identifiers: string containing only letters, digits or underscore characters. On the other hand, predicate names (i.e. atoms) can contain any character with quotes if necessary (e.g. 'x+y=z' is a valid predicate name). The compiler has then to encode predicate names respecting the syntax of identifiers. To achieve this, GNU Prolog uses an hexadecimal representation where each predicate name is translated to a symbol beginning with an x followed by the hexadecimal notation of the code of each character of the name.

Example: 'x+y=z' will be encoded as 782B793D7A since 78 is the hexadecimal representation of the code of x, 2B of the code of +, etc.

Since Prolog allows the user to define several predicates with the same name but with a different arity GNU Prolog encodes predicate indicators (predicate name followed by the arity). The symbol associated to the predicate name is then followed by an underscore and by the decimal notation of the arity.

Example: 'x+y=z'/3 will be encoded as 782B793D7A.3.

So, from the mini-assembly stage, each predicate indicator is replaced by its hexadecimal encoding. The knowledge of this encoding is normally not of interest for the user, i.e. the Prolog programmer. For this reason the GNU Prolog compiler hides this encoding. When an error occurs on a predicate (undefined predicate, predicate with multiple definitions,...) the compiler has to decode the symbol associated to the predicate indicator. For this gplc filters each message emitted by the linker to locate and decode eventual predicate indicators. This filtering can be deactivated specifying --no-decode-hexa when invoking gplc (section 2.4.3, page 12).

This filter is provided as an utility that can be invoked using the hexplic command as follows:

\[
% \text{hexplic [OPTION]... FILE...} \quad (the \% symbol is the operating system shell prompt)
\]

Options:

\begin{itemize}
\item [--encode] encoding mode (default mode is decoding)
\item [--relax] decode also predicate names (not only predicate indicators)
\item [--printf FORMAT] pass encoded/decoded string to C printf(3) with FORMAT
\item [--aux-father] decode an auxiliary predicate as its father
\item [--aux-father2] decode an auxiliary predicate as its father + auxiliary number
\item [--cmd-line] encode/decode each argument of the command-line
\item [-H] same as: --cmd-line --encode
\item [-P] same as: --cmd-line --relax
\item [--help] print a help and exit
\item [--version] print version number and exit
\end{itemize}

It is possible to give a prefix of an option if there is no ambiguity.

Without arguments hexplic runs in decoding mode reading its standard input and decoding each symbol corresponding to a predicate indicator. To use hexplic in the encoding mode the --encode option must be specified. By default hexplic only decodes predicate indicators, this can be relaxed using --relax
to also take into account simple predicate names (the arity can be omitted). It is possible to format the output of an encoded/decoded string using `--printf FORMAT` in that case each string \( S \) is passed to the C `printf(3)` function as `printf(FORMAT, S)`.

Auxiliary predicates are generated by the Prolog to WAM compiler when simplifying some control constructs like \( ';/2 \) present in the body of a clause. They are of the form \( 'S\textsc{name}/\textsc{arity}$_{\text{aux}N} \) where \( \textsc{name/arity} \) is the predicate indicator of the simplified (i.e. father) predicate and \( N \) is a sequential number (a predicate can give rise to several auxiliary predicates). It is possible to force \texttt{hexgplc} to decode an auxiliary predicate as its father predicate indicator using `--aux-father` or as its father predicate indicator followed by the sequential number using `--aux-father2`.

If no file is specified, \texttt{hexgplc} processes its standard input otherwise each file is treated sequentially. Specifying the `--cmd-line` option informs \texttt{hexgplc} that each argument is not a file name but a string that must be encoded (or decoded). This is useful to encode/decode a particular string. For this reason the option `\(-H \)` (encode to hexadecimal) and `\(-P \)` (decode to Prolog) are provided as shorthand. Then, to obtain the hexadecimal representation of a predicate \( P \) use:

\[
\text{% hexgplc } -H \ P
\]

Example:

\[
\text{% hexgplc } -H \ 'x+y=z' \\
X782B793D7A
\]
3 Debugging

3.1 Introduction

The GNU Prolog debugger provides information concerning the control flow of the program. The debugger can be fully used on consulted predicates (i.e. byte-code). For native compiled code only the calls/exit are traced, no internal behavior is shown. Under the debugger it is possible to exhaustively trace the execution or to set spy-points to only debug a specific part of the program. Spy-points allow the user to indicate on which predicates the debugger has to stop to allow the user to interact with it. The debugger uses the “procedure box control flow model”, also called the Byrd Box model since it is due to Lawrence Byrd.

3.2 The procedure box model

The procedure box model of Prolog execution provides a simple way to show the control flow. This model is very popular and has been adopted in many Prolog systems (e.g. SICStus Prolog, Quintus Prolog, ...). A good introduction is the chapter 8 of “Programming in Prolog” of Clocksin & Mellish [2]. The debugger executes a program step by step tracing an invocation to a predicate (call) and the return from this predicate due to either a success (exit) or a failure (fail). When a failure occurs the execution backtracks to the last predicate with an alternative clause. The predicate is then re-invoked (redo). Another source of change of the control flow is due to exceptions. When an exception is raised from a predicate (exception) by throw/1 (section 5.2.4, page 35) the control is given back to the most recent predicate that has defined a handler to recover this exception using catch/3 (section 5.2.4, page 35). The procedure box model shows these different changes in the control flow, as illustrated here:

Each arrow corresponds to a port. An arrow to the box indicates that the control is given to this predicate while an arrow from the box indicates that the control is given back from the procedure. This model visualizes the control flow through these five ports and the connections between the boxes associated to subgoals. Finally, it should be clear that a box is associated to one invocation of a given predicate. In particular, a recursive predicate will give raise to a box for each invocation of the predicate with different entries/exit in the control flow. Since this might get confusing for the user, the debugger associates to each box a unique identifier (i.e. the invocation number).

3.3 Debugging predicates

3.3.1 Running and stopping the debugger

trace/0 activates the debugger. The next invocation of a predicate will be traced.

debug/0 activates the debugger. The next invocation of a predicate on which a spy-point has been set will be traced.

It is important to understand that the information associated to the control flow is only available when the debugger is on. For efficiency reasons, when the debugger is off the information concerning the control
3.3 Debugging predicates

flow (i.e. the boxes) is not retained. So, if the debugger is activated in the middle of a computation (by a call to debug/0 or trace/0 in the program or after the interrupt key sequence (Ctrl-C) by choosing trace or debug), information prior to this point is not available.

debugging/0: prints onto the terminal information about the current debugging state (whether the debugger is switched on, what are the leashed ports, spy-points defined, ...).

notrace/0 or nodebug/0 switches the debugger off.

wam_debug/0 invokes the sub-debugger devoted to the WAM data structures (section 3.6, page 21). It can be also invoked using the \( \texttt{w} \) debugger command (section 3.5, page 20).

3.3.2 Leashing ports

leash(\texttt{Ports}) requests the debugger to prompt the user, as he creeps through the program, for every port defined in the \texttt{Ports} list. Each element of \texttt{Ports} is an atom in call, exit, redo, fail, exception. \texttt{Ports} can also be an atom defining a shorthand:

- full: equivalent to \texttt{[call, exit, redo, fail, exception]}
- half: equivalent to \texttt{[call, redo]}
- loose: equivalent to \texttt{[call]}
- none: equivalent to \texttt{[]}
- tight: equivalent to \texttt{[call, redo, fail, exception]}

When an unleashed port is encountered the debugger continues to show the associated goal but does not stop the execution to prompt the user.

3.3.3 Spy-points

When dealing with big sources it is not very practical to creep through the entire program. It is preferable to define a set of spy-points on interesting predicates to be prompted when the debugger reaches one of these predicates. Spy-points can be added either using \texttt{spy/1} (or \texttt{spypoint,condition/3}) or dynamically when prompted by the debugger using the + (or *) debugger command (section 3.5, page 20). The current mode of leashing does not affect spy-points in the sense that user interaction is requested on every port.

\texttt{spy} (\texttt{PredSpec}) sets a spy-point on all the predicates given by \texttt{PredSpec}. \texttt{PredSpec} defines one or several predicates and has one of the following forms:

- \texttt{[PredSpec1, PredSpec2, ...]}: set a spy-point for each element of the list.
- \texttt{Name}: set a spy-point for any predicate whose name is \texttt{Name} (whatever the arity).
- \texttt{Name/Arity}: set a spy-point for the predicate whose name is \texttt{Name} and arity is \texttt{Arity}.
- \texttt{Name/A1-A2}: set a spy-point for the each predicate whose name is \texttt{Name} and arity is between \texttt{A1} and \texttt{A2}.

It is not possible to set a spy-point on an undefined predicate.

The following predicate is used to remove one or several spy-points:

\texttt{nospy} (\texttt{PredSpec}) removes the spy-points from the specified predicates.
nospyall/0 removes all spy-points:

It is also possible to define conditional spy-points.

\texttt{spypoint\_condition(Goal, Port, Test)} sets a conditional spy-point on the predicate for Goal. When the debugger reaches a conditional spy-point it only shows the associated goal if the following conditions are verified:

- the actual goal unifies with Goal.
- the actual port unifies with Port.
- the Prolog goal Test succeeds.

### 3.4 Debugging messages

We here described which information is displayed by the debugger when it shows a goal. The basic format is as follows:

\begin{verbatim}
S N M Port: Goal ?
\end{verbatim}

\begin{itemize}
  \item \texttt{S} is a spy-point indicator; if there is a spy-point on the current goal the + symbol is displayed else a space is displayed. \texttt{N} is the invocation number. This unique number can be used to correlate the trace messages for the various ports, since it is unique for every invocation. \texttt{M} is an index number which represents the number of direct ancestors of the goal (i.e. the current depth of the goal). \texttt{Port} specifies the particular port (\texttt{call, exit, fail, redo, exception}). \texttt{Goal} is the current goal (it is then possible to inspect its current instantiation) which is displayed using \texttt{write\_term/3 with quoted(true)} and \texttt{max\_depth(D)} options (section 6.14.6, page 83). Initially \texttt{D} (the print depth) is set to 10 but can be redefined using the \texttt{<} debugger command (section 3.5, page 20). The ? symbol is displayed when the debugger is waiting a command from the user. (i.e. \texttt{Port} is a leashed port). If the port is unleashed, this symbol is not displayed and the debugger continues the execution displaying the next goal.
\end{itemize}

### 3.5 Debugger commands

When the debugger reaches a leashed port it shows the current goal followed by the ? symbol At this point there are many commands available. Typing RETURN will creep into the program. Continuing to creep will show all the control flow. The debugger shows every port for every predicate encountered during the execution. It is possible to select the ports at which the debugger will prompt the user using the built-in predicate \texttt{leash/1} (section 3.3.2, page 19). Each command is only one character long:
3.6 The WAM debugger

In some cases it is interesting to have access to the WAM data structures. This sub-debugger allows
the user to inspect/modify the contents of any stack or register of the WAM. The WAM debugger is
invoked using the built-in predicate `wam_debug/0` (section 3.3.1, page 18) or the `w` debugger command
(section 3.5, page 20). The following table presents the specific commands of the WAM debugger:

<table>
<thead>
<tr>
<th>Command</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET or c</td>
<td>creep</td>
<td>single-step to the next port</td>
</tr>
<tr>
<td>l</td>
<td>leap</td>
<td>continue the execution only stopping when a goal with a spy-point is reached</td>
</tr>
<tr>
<td>s</td>
<td>skip</td>
<td>skip over the entire execution of the current goal. No message will be shown until control returns</td>
</tr>
<tr>
<td>G</td>
<td>go to</td>
<td>ask for an invocation number and continue the execution until a port is reached for that invocation number</td>
</tr>
<tr>
<td>r</td>
<td>retry</td>
<td>try to restart the invocation of the current goal by failing until reaching the invocation of the goal. The state of execution is the same as when the goal was initially invoked (except when using side-effect predicates)</td>
</tr>
<tr>
<td>f</td>
<td>fail</td>
<td>force the current goal to fail immediately</td>
</tr>
<tr>
<td>w</td>
<td>write</td>
<td>show the current goal using <code>write/2</code> (section 6.14.6, page 83)</td>
</tr>
<tr>
<td>d</td>
<td>display</td>
<td>show the current goal using <code>display/2</code> (section 6.14.6, page 83)</td>
</tr>
<tr>
<td>p</td>
<td>print</td>
<td>show the current goal using <code>print/2</code> (section 6.14.6, page 83)</td>
</tr>
<tr>
<td>e</td>
<td>exception</td>
<td>show the pending exception. Only applicable to an <code>exception</code> port</td>
</tr>
<tr>
<td>g</td>
<td>ancestors</td>
<td>show the list of ancestors of the current goal</td>
</tr>
<tr>
<td>A</td>
<td>alternatives</td>
<td>show the list of ancestors of the current goal combined with choice-points</td>
</tr>
<tr>
<td>u</td>
<td>unify</td>
<td>ask for a term and unify the current goal with this term. This is convenient for getting a specific solution. Only available at a call port</td>
</tr>
<tr>
<td>.</td>
<td>father file</td>
<td>show the Prolog file name and the line number where the current predicate is defined</td>
</tr>
<tr>
<td>n</td>
<td>no debug</td>
<td>switch the debugger off. Same as <code>nodebug/0</code> (section 3.3.1, page 18)</td>
</tr>
<tr>
<td>=</td>
<td>debugging</td>
<td>show debugger information. Same as <code>debugging/0</code> (section 3.3.1, page 18)</td>
</tr>
<tr>
<td>+</td>
<td>spy this</td>
<td>set a spy-point on the current goal. Uses <code>spy/1</code> (section 3.3.3, page 19)</td>
</tr>
<tr>
<td>-</td>
<td>nospy this</td>
<td>remove a spy-point on the current goal. Uses <code>nospy/1</code> (section 3.3.3, page 19)</td>
</tr>
<tr>
<td>*</td>
<td>spy conditionally</td>
<td>ask for a term <code>Goal, Port, Test</code> (terminated by a dot) and set a conditional spy-point on the current predicate. Goal and the current goal must have the same predicate indicator. Uses <code>spypoint_condition/3</code> (section 3.3.3, page 19)</td>
</tr>
<tr>
<td>L</td>
<td>listing</td>
<td>list the clauses associated to the current predicate. Uses <code>listing/1</code> (section 6.23.3, page 122)</td>
</tr>
<tr>
<td>a</td>
<td>abort</td>
<td>abort the current execution. Same as <code>abort/0</code> (section 6.18.1, page 99)</td>
</tr>
<tr>
<td>b</td>
<td>break</td>
<td>invoke a recursive top-level. Same as <code>break/0</code> (section 6.18.1, page 99)</td>
</tr>
<tr>
<td>@</td>
<td>execute goal</td>
<td>ask for a goal and execute it</td>
</tr>
<tr>
<td>&lt;</td>
<td>set print depth</td>
<td>ask for an integer and set the print depth to this value (~1 for no depth limit)</td>
</tr>
<tr>
<td>h or ?</td>
<td>help</td>
<td>display a summary of available commands</td>
</tr>
<tr>
<td>w</td>
<td>WAM debugger</td>
<td>invoke the low-level WAM debugger (section 3.6, page 21)</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>write $A$ [$N$]</td>
<td>write $N$ terms starting at the address $A$ using write/1 (section 6.14.6, page 83)</td>
<td></td>
</tr>
<tr>
<td>data $A$ [$N$]</td>
<td>display $N$ words starting at the address $A$</td>
<td></td>
</tr>
<tr>
<td>modify $A$ [$N$]</td>
<td>display and modify $N$ words starting at the address $A$</td>
<td></td>
</tr>
<tr>
<td>where $SA$</td>
<td>display the real address corresponding to $SA$</td>
<td></td>
</tr>
<tr>
<td>what $RA$</td>
<td>display what corresponds to the real address $RA$</td>
<td></td>
</tr>
<tr>
<td>deref $A$</td>
<td>display the dereferenced word starting at the address $A$</td>
<td></td>
</tr>
<tr>
<td>envir [$SA$]</td>
<td>display the contents of the environment located at $SA$ (or the current one)</td>
<td></td>
</tr>
<tr>
<td>backtrack [$SA$]</td>
<td>display the contents of the choice-point located at $SA$ (or the current one)</td>
<td></td>
</tr>
<tr>
<td>backtrack all</td>
<td>display all choice-points</td>
<td></td>
</tr>
<tr>
<td>quit</td>
<td>quit the WAM debugger</td>
<td></td>
</tr>
<tr>
<td>help</td>
<td>display a summary of available commands</td>
<td></td>
</tr>
</tbody>
</table>

In the above table the following conventions apply:

- elements between [ and ] are optional.
- $N$ is an optional integer (defaults to 1).
- $A$ is a WAM address, its syntax is: \texttt{BANK\_NAME} [ [ $N$ ] ], i.e. a bank name possibly followed by an index (defaults to 0). \texttt{BANK\_NAME} is either:
  - \texttt{reg}: WAM general register (stack pointers, continuation, ...).
  - \texttt{x}: WAM X register (temporary variables, i.e. arguments).
  - \texttt{y}: WAM Y register (permanent variables).
  - \texttt{STACK\_NAME}: WAM stack (\texttt{STACK\_NAME} in local, global, trail, cstr).
- $SA$ is a WAM stack address, i.e. \texttt{STACK\_NAME} [ [ $N$ ] ] (special case of WAM addresses).
- $RA$ is a real address, its syntax if the syntax of C integers (in particular the notation 0x... is recognized).

It is possible to only use the first letters of a commands and bank names when there is no ambiguity. Also the square brackets [ ] enclosing the index of a bank name can be omitted. For instance the following command (showing the contents of 25 consecutive words of the global stack from the index 3): \texttt{data global[3]} 25 can be abbreviated as: \texttt{d g 3 25}.
4 Format of definitions

4.1 General format

The definition of control constructs, directives and built-in predicates is presented as follows:

Templates

Specifies the types of the arguments and which of them shall be instantiated (mode). Types and modes are described later (section 4.2, page 23).

Description

Describes the behavior (in the absence of any error conditions). It is explicitly mentioned when a built-in predicate is re-executable on backtracking. Predefined operators involved in the definition are also mentioned.

Errors

Details the error conditions. Possible errors are detailed later (section 4.3, page 25). For directives, this part is omitted.

Portability

Specifies whether the definition conforms to the ISO standard or is a GNU Prolog extension.

4.2 Types and modes

The templates part defines, for each argument of the concerned built-in predicate, its mode and type. The mode specifies whether or not the argument must be instantiated when the built-in predicate is called. The mode is encoded with a symbol just before the type. Possible modes are:

- `+`: the argument must be instantiated.
- `-`: the argument must be a variable (will be instantiated if the built-in predicate succeeds).
- `?`: the argument can be instantiated or a variable.

The type of an argument is defined by the following table:
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_list</td>
<td>a list whose the type of each element is TYPE</td>
</tr>
<tr>
<td>TYPE1_or_TYPE2</td>
<td>a term whose type is either TYPE1 or TYPE2</td>
</tr>
<tr>
<td>atom</td>
<td>an atom</td>
</tr>
<tr>
<td>atom_property</td>
<td>an atom property (section 6.19.12, page 107)</td>
</tr>
<tr>
<td>boolean</td>
<td>the atom true or false</td>
</tr>
<tr>
<td>byte</td>
<td>an integer ≥ 0 and ≤ 255</td>
</tr>
<tr>
<td>callable_term</td>
<td>an atom or a compound term</td>
</tr>
<tr>
<td>character</td>
<td>a single character atom</td>
</tr>
<tr>
<td>character_code</td>
<td>an integer ≥ 1 and ≤ 255</td>
</tr>
<tr>
<td>clause</td>
<td>a clause (fact or rule)</td>
</tr>
<tr>
<td>close_option</td>
<td>a close option (section 6.10.7, page 59)</td>
</tr>
<tr>
<td>compound_term</td>
<td>a compound term</td>
</tr>
<tr>
<td>evaluate</td>
<td>an arithmetic expression (section 6.6.1, page 45)</td>
</tr>
<tr>
<td>fd_bool_evaluable</td>
<td>a boolean FD expression (section 7.7.1, page 156)</td>
</tr>
<tr>
<td>fd_labeling_option</td>
<td>an FD labeling option (section 7.9.1, page 162)</td>
</tr>
<tr>
<td>fd_evaluable</td>
<td>an arithmetic FD expression (section 7.6.1, page 153)</td>
</tr>
<tr>
<td>fd_variable</td>
<td>an FD variable</td>
</tr>
<tr>
<td>flag</td>
<td>a Prolog flag (section 6.22.1, page 117)</td>
</tr>
<tr>
<td>float</td>
<td>a floating point number</td>
</tr>
<tr>
<td>head</td>
<td>a head of a clause (atom or compound term)</td>
</tr>
<tr>
<td>integer</td>
<td>an integer</td>
</tr>
<tr>
<td>in_byte</td>
<td>an integer ≥ 0 and ≤ 255 or -1 (for the end-of-file)</td>
</tr>
<tr>
<td>in_character</td>
<td>a single character atom or the atom end_of_file (for the end-of-file)</td>
</tr>
<tr>
<td>in_character_code</td>
<td>an integer ≥ 1 and ≤ 255 or -1 (for the end-of-file)</td>
</tr>
<tr>
<td>io_mode</td>
<td>an atom in: read, write or append</td>
</tr>
<tr>
<td>list</td>
<td>the empty list [] or a non-empty list [·]</td>
</tr>
<tr>
<td>nonvar</td>
<td>any term that is not a variable</td>
</tr>
<tr>
<td>number</td>
<td>an integer or a floating point number</td>
</tr>
<tr>
<td>operatorSpecifier</td>
<td>an operator specifier (section 6.14.10, page 87)</td>
</tr>
<tr>
<td>os_file_property</td>
<td>an operating system file property (section 6.27.11, page 132)</td>
</tr>
<tr>
<td>predicate_indicator</td>
<td>a term Name/Arity where Name is an atom and Arity an integer ≥ 0. A callable term can be given if the strict_iso Prolog flag is switched off (section 6.22.1, page 117)</td>
</tr>
<tr>
<td>predicate_property</td>
<td>a predicate property (section 6.8.2, page 52)</td>
</tr>
<tr>
<td>read_option</td>
<td>a read option (section 6.14.1, page 79)</td>
</tr>
<tr>
<td>socket_address</td>
<td>a term of the form ’AF_UNIX’(A) or ’AF_INET’(A,N) where A is an atom and N an integer</td>
</tr>
<tr>
<td>socket_domain</td>
<td>an atom in: ’AF_UNIX’ or ’AF_INET’</td>
</tr>
<tr>
<td>source_sink</td>
<td>an atom identifying a source or a sink</td>
</tr>
<tr>
<td>stream</td>
<td>a stream-term: a term of the form ’$stream’(N) where N is an integer ≥ 0</td>
</tr>
<tr>
<td>stream_option</td>
<td>a stream option (section 6.10.6, page 58)</td>
</tr>
<tr>
<td>stream_or_alias</td>
<td>a stream-term or an alias (atom)</td>
</tr>
<tr>
<td>stream_position</td>
<td>a stream position: a term ’$stream_position’(I1, I2, I3, I4) where I1, I2, I3 and I4 are integers</td>
</tr>
<tr>
<td>stream_property</td>
<td>a stream property (section 6.10.10, page 61)</td>
</tr>
<tr>
<td>stream_seek_method</td>
<td>an atom in: bof, current or eof</td>
</tr>
<tr>
<td>term</td>
<td>any term</td>
</tr>
<tr>
<td>var_binding_option</td>
<td>a variable binding option (section 6.5.3, page 44)</td>
</tr>
<tr>
<td>write_option</td>
<td>a write option (section 6.14.6, page 83)</td>
</tr>
</tbody>
</table>
4.3 Errors

4.3.1 General format and error context

When an error occurs an exception of the form: error(ErrorTerm, Caller) is raised. ErrorTerm is a term specifying the error (detailed in next sections) and Caller is a term specifying the context of the error. The context is either the predicate indicator of the last invoked built-in predicate or an atom giving general context information.

Using exceptions allows the user both to recover an error using catch/3 (section 5.2.4, page 35) and to raise an error using throw/1 (section 5.2.4, page 35).

To illustrate how to write a error cases, let us write a predicate my_pred(X) where X must be an integer:

```prolog
my_pred(X) :-
   ( nonvar(X) ->
     true;
     throw(error(instantiation_error, my_pred/1)),
   ),
   ( integer(X) ->
     true;
     throw(error(type_error(integer, X), my_pred/1))
   ),
   ...
```

To help the user to write these error cases, a set of system predicates is provided to raise errors. These predicates are of the form `$pl_err_...` and they all refer to the implicit error context. The predicates set_bip_name/2 (section 6.22.3, page 119) and current_bip_name/2 (section 6.22.4, page 119) are provided to set and recover the name and the arity associated to this context (an arity < 0 means that only the atom corresponding to the functor is significant). Using these system predicates the user could define the above predicate as follow:

```prolog
my_pred(X) :-
   set_bip_name(my_pred, 1),
   ( nonvar(X) ->
     true;
     '$pl_err_instantiation'
   ),
   ( integer(X) ->
     true;
     '$pl_err_type'(integer, X)
   ),
   ...
```

The following sections detail each kind of errors (and associated system predicates).

4.3.2 Instantiation error

An instantiation error occurs when an argument or one of its components is variable while an instantiated argument was expected. ErrorTerm has the following form: instantiation_error.

The system predicate `$pl_err_instantiation` raises this error in the current error context (section 4.3.1, page 25).
4.3.3 Type error

A type error occurs when the type of an argument or one of its components is not the expected type (but not a variable). ErrorTerm has the following form: type_error(Type, Culprit) where Type is the expected type and Culprit the argument which caused the error. Type is one of:

- atom
- atomic
- boolean
- byte
- callable
- character
- compound
- evaluable
- fd_bool_evaluable
- fd_evaluable
- fd_variable
- float
- in_byte
- in_character
- integer
- list
- number
- predicate_indicator
- variable

The system predicate `$pl_errno_type`(Type, Culprit) raises this error in the current error context (section 4.3.1, page 25).

4.3.4 Domain error

A domain error occurs when the type of an argument is correct but its value is outside the expected domain. ErrorTerm has the following form: domain_error(Domain, Culprit) where Domain is the expected domain and Culprit the argument which caused the error. Domain is one of:

- atom_property
- buffering_mode
- character_code_list
- close_option
- date_time
- eof_action
- fd_labeling_option
- flag_value
- format_control_sequence
- g_array_index
- io_mode
- non_empty_list
- not_less_than_zero
- operator_priority
- operator_specifier
- os_file_permission
- os_file_property
- os_path
- predicate_property
- prolog_flag
- read_option
- selectable_item
- socket_address
- socket_domain
- source_sink
- statistics_key
- statistics_value
- stream
- stream_option
- stream_or_alias
- stream_position
- stream_property
- stream_seek_method
- stream_type
- term_stream_or_alias
- var_binding_option
- write_option

The system predicate `$pl_errno_domain`(Domain, Culprit) raises this error in the current error context (section 4.3.1, page 25).
4.3.5 Existence error

An existence error occurs when an object on which an operation is to be performed does not exist. ErrorTerm has the following form: existence_error(Object, Culprit) where Object is the type of the object and Culprit the argument which caused the error. Object is one of:

- procedure
- source_sink
- stream

The system predicate 'pl_err_existence'(Object, Culprit) raises this error in the current error context (section 4.3.1, page 25).

4.3.6 Permission error

A permission error occurs when an attempt to perform a prohibited operation is made. ErrorTerm has the following form: permission_error(Operation, Permission, Culprit) where Operation is the operation which caused the error, Permission the type of the tried permission and Culprit the argument which caused the error. Operation is one of:

- access
- add_alias
- close
- create
- input
- modify
- open
- output
- reposition

and Permission is one of:

- binary_stream
- past_end_of_stream
- static_procedure
- flag
- private_procedure
- stream
- operator
- source_sink
- text_stream

The system predicate 'pl_err_permission'(Operation, Permission, Culprit) raises this error in the current error context (section 4.3.1, page 25).

4.3.7 Representation error

A representation error occurs when an implementation limit has been breached. ErrorTerm has the following form: representation_error(Limit) where Limit is the name of the reached limit. Limit is one of:

- character
- max_arith
- too_many_variables
- character_code
- max_integer
- in_character_code
- min_integer

The errors max_integer and min_integer are not currently implemented.

The system predicate 'pl_err_representation'(Limit) raises this error in the current error context (section 4.3.1, page 25).
4.3.8 Evaluation error

An evaluation error occurs when an arithmetic expression gives rise to an exceptional value. ErrorTerm has the following form: evaluation_error(Err) where Err is the name of the error. Err is one of:

- float_overflow
- int_overflow
- undefined
- zero_divisor
- underflow

The errors float_overflow, int_overflow, undefined and underflow are not currently implemented.

The system predicate '$pl.err.evaluation'(Err) raises this error in the current error context (section 4.3.1, page 25).

4.3.9 Resource error

A resource error occurs when GNU Prolog does not have enough resources. ErrorTerm has the following form: resource_error(Res) where Res is the name of the resource. Res is one of:

- print_object_not_linked
- too_big_fd_constraint
- too_many_open_streams

The system predicate '$pl.err.resource'(Res) raises this error in the current error context (section 4.3.1, page 25).

4.3.10 Syntax error

A syntax error occurs when a sequence of character does not conform to the syntax of terms. ErrorTerm has the following form: syntax_error(Err) where Err is an atom explaining the error.

The system predicate '$pl.err.syntax'(Err) raises this error in the current error context (section 4.3.1, page 25).

4.3.11 System error

A system error can occur at any stage. A system error is generally associated to an external component (e.g. operating system). ErrorTerm has the following form: system_error(Err) where Err is an atom explaining the error. This is an extension to ISO which only defines system_error without arguments.

The system predicate '$pl.err.system'(Err) raises this error in the current error context (section 4.3.1, page 25).
5 Prolog directives and control constructs

5.1 Prolog directives

5.1.1 Introduction

Prolog directives are annotations inserted in Prolog source files for the compiler. A Prolog directive is used to specify:

- the properties of some procedures defined in the source file.
- the format and the syntax for read-terms in the source file (using changeable Prolog flags).
- included source files.
- a goal to be executed at run-time.

5.1.2 dynamic/1

Templates

\[
dynamic(+\text{predicate\_indicator}) \\
dynamic(+\text{predicate\_indicator\_list}) \\
dynamic(+\text{predicate\_indicator\_sequence})
\]

Description

dynamic(Pred) specifies that the procedure whose predicate indicator is Pred is a dynamic procedure. This directive makes it possible to alter the definition of Pred by adding or removing clauses. For more information refer to the section about dynamic clause management (section 6.7.1, page 48).

This directive shall precede the definition of Pred in the source file.

If there is no clause for Pred in the source file, Pred exists however as an empty predicate (this means that current\_predicate(Pred) succeeds).

In order to allow multiple definitions, Pred can also be a list of predicate indicators or a sequence of predicate indicators using ‘,‘/2 as separator.

Portability

ISO directive.

5.1.3 public/1

Templates

\[
\text{public(+predicate\_indicator)} \\
\text{public(+predicate\_indicator\_list)} \\
\text{public(+predicate\_indicator\_sequence)}
\]

Description
public(Pred) specifies that the procedure whose predicate indicator is Pred is a public procedure. This directive makes it possible to inspect the clauses of Pred. For more information refer to the section about dynamic clause management (section 6.7.1, page 48).

This directive shall precede the definition of Pred in the source file. Since a dynamic procedure is also public. It is useless (but correct) to define a public directive for a predicate already declared as dynamic.

In order to allow multiple definitions, Pred can also be a list of predicate indicators or a sequence of predicate indicators using ‘,’ /2 as separator.

Portability

GNU Prolog directive. The ISO reference does not define any directive to declare a predicate public but it does distinguish public predicates. It is worth noting that in most Prolog systems the public/1 directive is as a visibility declaration. Indeed, declaring a predicate as public makes it visible from any predicate defined in any other file (otherwise the predicate is only visible from predicates defined in the same source file as itself). When a module system is incorporated in GNU Prolog a more general visibility declaration shall be provided conforming to the ISO reference.

5.1.4 multfile/1

Templates

multfile(+predicate_indicator)
multfile(+predicate_indicator_list)
multfile(+predicate_indicator_sequence)

Description

multfile(Pred) is not supported by GNU Prolog. When such a directive is encountered it is simply ignored. All clauses for a given predicate must reside in a single file.

Portability

ISO directive. Not supported.

5.1.5 discontiguous/1

Templates

discontiguous(+predicate_indicator)
discontiguous(+predicate_indicator_list)
discontiguous(+predicate_indicator_sequence)

Description

discontiguous(Pred) specifies that the procedure whose predicate indicator is Pred is a discontiguous procedure. Namely, the clauses defining Pred are not restricted to be consecutive but can appear anywhere in the source file.

This directive shall precede the definition of Pred in the source file.

In order to allow multiple definitions, Pred can also be a list of predicate indicators or a sequence of predicate indicators using ’,’ /2 as separator.
5.1 Prolog directives

Portability

ISO directive. The ISO reference document states that if there is no clause for \texttt{Pred} in the source file, \texttt{Pred} exists however as an empty predicate (i.e. \texttt{current
class}(Pred) will succeed). This is not the case for GNU Prolog.

5.1.6 \texttt{built-in/0}, \texttt{built-in/1}, \texttt{built-in_fd/0}, \texttt{built-in_fd/1}

Templates

\begin{verbatim}
built-in
built-in(+predicate_indicator)
built-in(+predicate_indicator_list)
built-in(+predicate_indicator_sequence)
built-in_fd
built-in_fd(+predicate_indicator)
built-in_fd(+predicate_indicator_list)
built-in_fd(+predicate_indicator_sequence)
\end{verbatim}

Description

\texttt{built-in} specifies that the procedures defined from now have the \texttt{built-in} property (section 6.8.2, page 52).

\texttt{built-in}(Pred) is similar to \texttt{built-in/0} but only affects the procedure whose predicate indicator is \texttt{Pred}.

This directive shall precede the definition of \texttt{Pred} in the source file.

In order to allow multiple definitions, \texttt{Pred} can also be a list of predicate indicators or a sequence of predicate indicators using "," as separator.

\texttt{built-in_fd} (resp. \texttt{built-in_fd}(Pred)) is similar to \texttt{built-in} (resp. \texttt{built-in}(Pred)) but sets the \texttt{built-in_fd} predicate property (section 6.8.2, page 52).

Portability

GNU Prolog directives.

5.1.7 \texttt{include/1}

Templates

\begin{verbatim}
include(+atom)
\end{verbatim}

Description

\texttt{include(File)} specifies that the content of the Prolog source \texttt{File} shall be inserted. The resulting Prolog text is identical to the Prolog text obtained by replacing the directive by the content of the Prolog source \texttt{File}.

See \texttt{absolute_file_name/2} for information about the syntax of \texttt{File} (section 6.2.1, page 125).

Portability
ISO directive.

5.1.8 ensure_loaded/1

Templates

ensure_loaded(+atom)

Description

ensure_loaded(File) is not supported by GNU Prolog. When such a directive is encountered it is simply ignored.

Portability

ISO directive. Not supported.

5.1.9 op/3

Templates

op(+integer, +operator_specifier, +atom_or_atom_list)

Description

op(Priority, OpSpecifier, Operator) alters the operator table. This directive is executed as soon as it is encountered by calling the built-in predicate op/3 (section 6.14.10, page 87). A system directive is also generated to reflect the effect of this directive at run-time (section 2.4.4, page 15).

Portability

ISO directive.

5.1.10 char_conversion/2

Templates

char_conversion(+character, +character)

Description

char_conversion(InChar, OutChar) alters the character-conversion mapping. This directive is executed as soon as it is encountered by a call to the built-in predicate char_conversion/2 (section 6.14.12, page 89). A system directive is also generated to reflect the effect of this directive at run-time (section 2.4.4, page 15).

Portability

ISO directive.

5.1.11 set_prolog_flag/2

Templates
5.2 Prolog control constructs

set_prolog_flag(+flag, +term)

Description

`set_prolog_flag(Flag, Value)` sets the value of the Prolog flag `Flag` to `Value`. This directive is executed as soon as it is encountered by a call to the built-in predicate `set_prolog_flag/2` (section 6.22.1, page 117). A system directive is also generated to reflect the effect of this directive at run-time (section 2.4.4, page 15).

Portability

ISO directive.

5.1.12 initialization/1

Templates

`initialization(+callable_term)`

Description

`initialization(Goal)` adds `Goal` to the set of goal which shall be executed at run-time. A user directive is generated to execute `Goal` at run-time. If several initialization directives appear in the same file they are executed in the order of apparition (section 2.4.4, page 15).

Portability

ISO directive.

5.1.13 foreign/2, foreign/1

Templates

`foreign(+callable_term, +foreign_option_list)`

`foreign(+callable_term)`

Description

`foreign(Template, Options)` defines an interface predicate whose prototype is `Template` according to the options given by `Options`. Refer to the foreign code interface for more information (section 8.1, page 165).

`foreign(Template)` is equivalent to `foreign(Template, []).`

Portability

GNU Prolog directive.

5.2 Prolog control constructs

5.2.1 true/0, fail/0, !/0

Templates
true
fail !

Description

ture always succeeds.
fail always fails (enforces backtracking).
! always succeeds and the for side-effect of removing all choice-points created since the invocation of the predicate activating it.

Errors

None.

Portability

ISO control constructs.

5.2.2 (',')/2 - conjunction, (;)/2 - disjunction, (->)/2 - if-then

Templates

','(callable_term, callable_term)
;(callable_term, callable_term)
->(callable_term, callable_term)

Description

Goal1 ; Goal2 first creates a choice-point and executes Goal1. On backtracking Goal2 is executed.

Goal1 -> Goal2 first executes Goal1 and, in case of success, removes all choice-points created by Goal1 and executes Goal2. This control construct acts like an if-then (Goal1 is the test part and Goal2 the then part). Note that if Goal1 fails ->/2 fails also. ->/2 is often combined with ;/2 to define an if-then-else as follows: Goal1 -> Goal2 ; Goal3. Note that Goal1 -> Goal2 is the first argument of the (;)/2 and Goal13 (the else part) is the second argument. Such an if-then-else control construct first creates a choice-point for the else-part (intuitively associated to ;/2) and then executes Goal1. In case of success, all choice-points created by Goal1 together with the choice-point for the else-part are removed and Goal2 is executed. If Goal1 fails then Goal3 is executed.

',', ; and -> are predefined infix operators (section 6.14.10, page 87).

Errors

<table>
<thead>
<tr>
<th>Goal1 or Goal2 is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal1 is neither a variable nor a callable term</td>
<td>type_error(callable, Goal1)</td>
</tr>
<tr>
<td>Goal2 is neither a variable nor a callable term</td>
<td>type_error(callable, Goal2)</td>
</tr>
<tr>
<td>The predicate indicator Pred of Goal1 or Goal2 does not correspond to an existing procedure and the value of the unknown Prolog flag is error (section 6.22.1, page 117)</td>
<td>existence_error(procedure, Pred)</td>
</tr>
</tbody>
</table>
5.2 Prolog control constructs

Portability

ISO control constructs.

5.2.3 call/1

Templates

call(+callable, term)

Description

call(Goal) executes Goal. call/1 succeeds if Goal represents a goal which is true. When Goal contains a cut symbol ! (section 5.2.1, page 33) as a subgoal, the effect of ! does not extend outside Goal.

Errors

<table>
<thead>
<tr>
<th>Goal is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal is neither a variable nor a callable term</td>
<td>type_error(callable, Goal)</td>
</tr>
<tr>
<td>The predicate indicator Pred of Goal does not correspond to an existing procedure and the value of the unknown Prolog flag is error (section 6.22.1, page 117)</td>
<td>existence_error(procedure, Pred)</td>
</tr>
</tbody>
</table>

Portability

ISO control construct.

5.2.4 catch/3, throw/1

Templates

catch(?callable, term, ?term, ?term)

throw(+nonvar)

Description

catch(Goal, Catcher, Recovery) is similar to call(Goal) (section 5.2.3, page 35). If this succeeds or fails, so does the call to catch/3. If however, during the execution of Goal, there is a call to throw(Ball), the current flow of control is interrupted, and control returns to a call of catch/3 that is being executed. This can happen in one of two ways:

- implicitly, when an error condition for a built-in predicate is satisfied.
- explicitly, when the program executes a call of throw/1 because the program wishes to abandon the current processing, and instead to take an alternative action.

throw(Ball) causes the normal flow of control to be transferred back to an existing call of catch/3. When a call to throw(Ball) happens, Ball is copied and the stack is unwound back to the call to catch/3, whereupon the copy of Ball is unified with Catcher. If this unification succeeds, then catch/3 executes the goal Recovery using call/1 (section 5.2.3, page 35) in order to determine the success or failure of catch/3. Otherwise, in case the unification fails, the stack keeps unwinding, looking for an earlier invocation of catch/3. Ball may be any non-variable term.

Errors
<table>
<thead>
<tr>
<th>Goal is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal is neither a variable nor a callable term</td>
<td>type_error(callable, Goal)</td>
</tr>
<tr>
<td>The predicate indicator Pred of Goal does not correspond to an existing procedure and the value of the unknown Prolog flag is error (section 6.22.1, page 117)</td>
<td>existence_error(procedure, Pred)</td>
</tr>
<tr>
<td>Ball is a variable</td>
<td>instantiation_error</td>
</tr>
</tbody>
</table>

If Ball does not unify with the Catcher argument of any call of catch/3, a system error message is displayed and throw/1 fails.

When catch/3 calls Recovery it uses call/1 (section 5.2.3, page 35), an instantiation_error, a type_error or an existence_error can then occur depending on Recovery.

**Portability**

ISO control constructs.
6 Prolog built-in predicates

6.1 Type testing

6.1.1 var/1, nonvar/1, atom/1, integer/1, float/1, number/1, atomic/1, compound/1, callable/1, list/1, partial_list/1, list_or_partial_list/1

Templates

\[
\begin{align*}
\text{var(?term)} & \quad \text{atomic(?term)} \\
\text{nonvar(?term)} & \quad \text{compound(?term)} \\
\text{atom(?term)} & \quad \text{callable(?term)} \\
\text{integer(?term)} & \quad \text{list(?term)} \\
\text{float(?term)} & \quad \text{partial_list(?term)} \\
\text{number(?term)} & \quad \text{list_or_partial_list(?term)}
\end{align*}
\]

Description

\text{var(Term)}\ succeeds if \text{Term} is currently uninstantiated (which therefore has not been bound to anything, except possibly another uninstantiated variable).

\text{nonvar(Term)}\ succeeds if \text{Term} is currently instantiated (opposite of \text{var/1}).

\text{atom(Term)}\ succeeds if \text{Term} is currently instantiated to an atom.

\text{integer(Term)}\ succeeds if \text{Term} is currently instantiated to an integer.

\text{float(Term)}\ succeeds if \text{Term} is currently instantiated to a floating point number.

\text{number(Term)}\ succeeds if \text{Term} is currently instantiated to an integer or a floating point number.

\text{atomic(Term)}\ succeeds if \text{Term} is currently instantiated to an atom, an integer or a floating point number.

\text{compound(Term)}\ succeeds if \text{Term} is currently instantiated to a compound term, i.e. a term of arity > 0 (a list or a structure).

\text{callable(Term)}\ succeeds if \text{Term} is currently instantiated to a callable term, i.e. an atom or a compound term.

\text{list(Term)}\ succeeds if \text{Term} is currently instantiated to a list, i.e. the atom [] (empty list) or a term with principal functor \text{""}/2 and with second argument (the tail) a list.

\text{partial_list(Term)}\ succeeds if \text{Term} is currently instantiated to a partial list, i.e. a variable or a term whose the main functor is \text{""}/2 and the second argument (the tail) is a partial list.

\text{list_or_partial_list(Term)}\ succeeds if \text{Term} is currently instantiated to a list or a partial list.

Errors

None.

Portability

\text{var/1, nonvar/1, atom/1, integer/1, float/1, number/1, atomic/1, compound/1 and callable/1} are ISO predicates.
list/1, partial_list/1 and list_or_partial_list/1 are GNU Prolog predicates.

6.2 Term unification

6.2.1 (=)/2 - Prolog unification

Templates

```prolog
= (?term, ?term)
```

Description

Term1 = Term2 unifies Term1 and Term2. No occurs check is done, i.e. this predicate does not check if a variable is unified with a compound term containing this variable (this can lead to an infinite loop).

= is a predefined infix operator (section 6.14.10, page 87).

Errors

None.

Portability

ISO predicate.

6.2.2 unify_with_occurs_check/2

Templates

```prolog
unify_with_occurs_check(?term, ?term)
```

Description

`unify_with_occurs_check(Term1, Term2)` unifies Term1 and Term2. The occurs check test is done (i.e. the unification fails if a variable is unified with a compound term containing this variable).

Errors

None.

Portability

ISO predicate.

6.2.3 (\=)/2 - not Prolog unifiable

Templates

```prolog
\=(?term, ?term)
```

Description
6.3 Term comparison

Term1 \= Term2 succeeds if Term1 and Term2 are not unifiable (no occurs check is done).

\= is a predefined infix operator (section 6.14.10, page 87).

Errors

None.

Portability

ISO predicate.

6.3.1 Standard total ordering of terms

The built-in predicates described in this section allows the user to compare Prolog terms. Prolog terms are totally ordered according to the standard total ordering of terms which is as follows (from the smallest term to the greatest):

- variables, oldest first.
- finite domain variables (section 7.1.1, page 148), oldest first.
- floating point numbers, in numeric order.
- integers, in numeric order.
- atoms, in alphabetical (i.e. character code) order.
- compound terms, ordered first by arity, then by the name of the principal functor and by the arguments in left-to-right order.

A list is treated as a compound term (whose principal functor is \(\cdot\cdot/2\)).

The portability of the order of variables is not guaranteed (in the ISO reference the order of variables is system dependent).

6.3.2 \(==/2\) - term identical, \(\\neq/2\) - term not identical,
\(<=/2\) - term less than, \(=\neq/2\) - term less than or equal to,
\(>=/2\) - term greater than, \(>=\neq/2\) - term greater than or equal to

Templates

```
==(?term, ?term)  \neq(?term, ?term)
\=(?term, ?term)  \neq(?term, ?term)
<=(?term, ?term)  \geq(?term, ?term)
```  

Description

These predicates compare two terms according to the standard total ordering of terms (section 6.3.1, page 39).

Term1 == Term2 succeeds if Term1 and Term2 are equal.
Term1 \istency Term2 succeeds if Term1 and Term2 are different.

Term1 \istency Term2 succeeds if Term1 is less than Term2.

Term1 \istency Term2 succeeds if Term1 is less than or equal to Term2.

Term1 \istency Term2 succeeds if Term1 is greater than Term2.

Term1 \istency Term2 succeeds if Term1 is greater than or equal to Term2.

\istency, \istency, \istency, \istency and \istency are predefined infix operators (section 6.14.10, page 87).

Errors

None.

Portability

ISO predicates.

6.3.3 compare/3

Templates

compare(?atom, +term, +term)

Description

compare(Result, Term1, Term2) compares Term1 and Term2 according to the standard (section 6.3.1, page 39) and unifies Result with:

- the atom < if Term1 is less than Term2.
- the atom = if Term1 and Term2 are equal.
- the atom > if Term1 is greater than Term2.

Errors

| Result is neither a variable nor an atom | type_error(atom, Result) |

Portability

GNU Prolog predicate.

6.4 Term processing

6.4.1 functor/3

Templates

functor(+nonvar, ?atomic, ?integer)
functor(-nonvar, +atomic, +integer)
### 6.4 Term processing

#### Description

functor(Term, Name, Arity) succeeds if the principal functor of Term is Name and its arity is Arity.
This predicate can be used in two ways:

- **Term** is not a variable: extract the name (an atom or a number if Term is a number) and the arity of Term (if Term is atomic Arity = 0).
- **Term** is a variable: unify Term with a general term whose principal functor is given by Name and arity is given by Arity.

#### Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term and Name are both variables</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Term and Arity are both variables</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Term is a variable and Name is neither a variable nor an atomic term</td>
<td>type_error(atomic, Name)</td>
</tr>
<tr>
<td>Term is a variable and Arity is neither a variable nor an integer</td>
<td>type_error(integer, Arity)</td>
</tr>
<tr>
<td>Term is a variable, Name is a constant but not an atom and Arity is an integer &gt; 0</td>
<td>type_error(atomic, Name)</td>
</tr>
<tr>
<td>Term is a variable and Arity is an integer &gt; max arity flag (section 6.22.1, page 117)</td>
<td>representation_error(max,arity)</td>
</tr>
<tr>
<td>Term is a variable and Arity is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, Arity)</td>
</tr>
</tbody>
</table>

#### Portability

ISO predicate.

#### 6.4.2 arg/3

#### Templates

arg(+integer, +compound, Term, ?term)

#### Description

arg(N, Term, Arg) succeeds if the Nth argument of Term is Arg.

#### Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>N is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Term is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>N is neither a variable nor an integer</td>
<td>type_error(integer, N)</td>
</tr>
<tr>
<td>Term is neither a variable nor a compound term</td>
<td>type_error(compound, Term)</td>
</tr>
<tr>
<td>N is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, N)</td>
</tr>
</tbody>
</table>

#### Portability

ISO predicate.

#### 6.4.3 (=..)/2 - univ

#### Templates
=.(+nonvar, ?list)  
=.(+nonvar, +list)

Description
Term =.. List succeeds if List is a list whose head is the atom corresponding to the principal functor of Term and whose tail is a list of the arguments of Term.

=.. is a predefined infix operator (section 6.14.10, page 87).

Errors

| Term is a variable and List is a partial list | instantiation_error |
| List is neither a partial list nor a list | type_error(list, List) |
| Term is a variable and List is a list whose head is a variable | instantiation_error |
| List is a list whose head H is neither an atom nor a variable and whose tail is not the empty list | type_error(atomic, H) |
| List is a list whose head H is a compound term and whose tail is the empty list | type_error(atomic, H) |
| Term is a variable and List is the empty list | domain_error(non_empty_list, []) |
| Term is a variable and the tail of List has a length > max arity flag (section 6.22.1, page 117) | representation_error(max arity) |

Portability
ISO predicate.

6.4.4 copy_term/2

Templates

\[ \text{copy_term(?term, ?term)} \]

Description
\( \text{copy_term(Term1, Term2)} \) succeeds if Term2 unifies with a term \( T \) which is a renamed copy of Term1.

Errors
None.

Portability
ISO predicate.

6.4.5 setarg/4, setarg/3

Templates

\[ \text{setarg(+integer, +compound_term, +term, +boolean)} \]
\[ \text{setarg(+integer, +compound_term, +term)} \]
Description

setarg(N, Term, NewValue, Undo) replaces destructively the Nth argument of Term with NewValue. This assignment is undone on backtracking if Undo = true. This should only be used if there is no further use of the old value of the replaced argument. If Undo = false then NewValue must be either an atom or an integer.

setarg(N, Term, NewValue) is equivalent to setarg(N, Term, NewValue, true).

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>N is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>N is neither a variable nor an integer</td>
<td>type_error(integer, N)</td>
</tr>
<tr>
<td>N is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, N)</td>
</tr>
<tr>
<td>Term is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Term is neither a variable nor a compound term</td>
<td>type_error(compound, Term)</td>
</tr>
<tr>
<td>NewValue is neither an atom nor an integer and</td>
<td>type_error(atomic, NewValue)</td>
</tr>
<tr>
<td>Undo = false</td>
<td></td>
</tr>
<tr>
<td>Undo is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Undo is neither a variable nor a boolean</td>
<td>type_error(boolean, Undo)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.5 Variable naming/numbering

6.5.1 name_singleton_vars/1

Templates

name_singleton_vars(?term)

Description

name_singleton_vars(Term) binds each singleton variable appearing in Term with a term of the form 'VARNAME'(' '). Such a term can be output by write_term/3 as a variable name (section 6.14.6, page 83).

Errors

None.

Portability

GNU Prolog predicates.

6.5.2 name_query_vars/2

Templates

name_query_vars(+list, ?list)
Description

name_query_vars(List, Rest) for each element of List of the form Name = Var where Name is an atom and Var a variable, binds Var with the term '$VARNAME'(Name). Such a term can be output by write_term/3 as a variable name (section 6.14.6, page 83). Rest is unified with the list of elements of List that have not given rise to a binding. This predicate is provided as a way to name the variable lists obtained returned by read_term/3 with variable_names(List) or singletons(List) options (section 6.14.1, page 79).

Errors

| List is a partial list | instantiation_error |
| List is neither a partial list nor a list | type_error(list, List) |
| Rest is neither a partial list nor a list | type_error(list, Rest) |

Portability

GNU Prolog predicate.

6.5.3 bind_variables/2, numvars/3, numvars/1

Templates

bind_variables(?term, +var,binding option_list)
numvars(?term, +var, integer, ?integer)
nnumvars(?term)

Description

bind_variables(Term, Options) binds each variable appearing in Term according to the options given by Options.

Variable binding options: Options is a list of variable binding options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- **numvars**: specifies that each variable appearing in Term should be bound to a term of the form '$VAR'(N) where N is an integer. Such a term can be output by write_term/3 as a variable name (section 6.14.6, page 83). This is the default.

- **namevars**: specifies that each variables appearing in Term shall be bound to a term of the form '$VARNAME'(Name) where Name is the atom that would be output by write_term/3 seeing a term of the '$VAR'(N) where N is an integer. Such a term can be output by write_term/3 as a variable name (section 6.14.6, page 83). This is the alternative to numvars.

- **from(From)**: the first integer N to use for number/name variables of Term is From. The default value is 0.

- **next(Next)**: when bind_variables/2 succeeds, Next is unified with the (last integer N)+1 used to bind the variables of Term.

- **exclude(List)**: collects all variable names appearing in List to avoid a clash when binding a variable of Term. Precisely a number N ≥ From will not be used to bind a variable of Term if:
  - there is a sub-term of List of the form '$VAR'(N) or '$VARNAME'(Name) where Name is the constant that would be output by write_term/3 seeing a term of the '$VAR'(N).
  - an element of List is of the form Name = Var where Name is an atom that would be output by write_term/3 on seeing a term of the from '$VAR'(N). This case allows for lists returned by read_term/3 (with variable_names(List) or singletons(List) options) (section 6.14.1, page 79) and by name_query_vars/2 (section 6.5.2, page 43).
numbervars(Term, From, Next) is equivalent to bind_variables(Term, [from(From), next(Next)]), i.e. each variable of Term is bound to 'VAR'(N) where From \leq N < Next.

numbervars(Term) is equivalent to numbervars(Term, 0, _).

Errors

| Options is a partial list or a list with an element E which is a variable | instantiation_error |
| Options is neither a partial list nor a list | type_error(list, Options) |
| an element E of the Options list is neither a variable nor a variable binding option | domain_error(var_binding_option, E) |
| From is a variable | instantiation_error |
| From is neither a variable nor an integer | type_error(integer, From) |
| Next is neither a variable nor an integer | type_error(integer, Next) |
| List is a partial list | instantiation_error |
| List is neither a partial list nor a list | type_error(list, List) |

Portability

GNU Prolog predicates.

6.5.4 term_ref/2

Templates

\[
\begin{align*}
\text{term_ref(+term, ?integer)} \\
\text{term_ref(?term, +integer)}
\end{align*}
\]

Description

term_ref(Term, Ref) succeeds if the internal reference of Term is Ref. This predicate can be used either to obtain the internal reference of a term or to obtain the term associated to a given reference. Note that two identical terms can have different internal references. A good way to use this predicate is to first record the internal reference of a given term and to later re-obtain the term via this reference.

Errors

| Term and Ref are both variables | instantiation_error |
| Ref is neither a variable nor an integer | type_error(integer, Ref) |
| Ref is an integer < 0 | domain_error(not_less_than_zero, Ref) |

Portability

GNU Prolog predicate.

6.6 Arithmetic

6.6.1 Evaluation of an arithmetic expression

An arithmetic expression is a Prolog term built from numbers, variables, and functors (or operators) that represent arithmetic functions. When an expression is evaluated each variable must be bound to a non-variable expression. An expression evaluates to a number, which may be an integer or a floating
point number. The following table details the components of an arithmetic expression, how they are evaluated, the types expected/returned and if they are ISO or an extension:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result = eval(Expression)</th>
<th>Signature</th>
<th>ISO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>must be bound to a non-variable expression E. The result is eval(E)</td>
<td>IF → IF</td>
<td>Y</td>
</tr>
<tr>
<td>integer number</td>
<td>this number</td>
<td>I → I</td>
<td>Y</td>
</tr>
<tr>
<td>floating point number</td>
<td>this number</td>
<td>F → F</td>
<td>Y</td>
</tr>
<tr>
<td>+ E</td>
<td>eval(E)</td>
<td>IF → IF</td>
<td>N</td>
</tr>
<tr>
<td>- E</td>
<td>- eval(E)</td>
<td>IF → IF</td>
<td>Y</td>
</tr>
<tr>
<td>inc(E)</td>
<td>eval(E) + 1</td>
<td>IF → IF</td>
<td>N</td>
</tr>
<tr>
<td>dec(E)</td>
<td>eval(E) - 1</td>
<td>IF → IF</td>
<td>N</td>
</tr>
<tr>
<td>E1 + E2</td>
<td>eval(E1) + eval(E2)</td>
<td>IF, IF → IF</td>
<td>Y</td>
</tr>
<tr>
<td>E1 - E2</td>
<td>eval(E1) - eval(E2)</td>
<td>IF, IF → IF</td>
<td>Y</td>
</tr>
<tr>
<td>E1 * E2</td>
<td>eval(E1) * eval(E2)</td>
<td>IF, IF → IF</td>
<td>Y</td>
</tr>
<tr>
<td>E1 / E2</td>
<td>eval(E1) / eval(E2)</td>
<td>IF, IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>E1 // E2</td>
<td>rnd(eval(E1) / eval(E2))</td>
<td>I, I → I</td>
<td>Y</td>
</tr>
<tr>
<td>E1 rem E2</td>
<td>eval(E1) - ( rndi(eval(E1) / eval(E2)) * eval(E2) )</td>
<td>I, I → I</td>
<td>Y</td>
</tr>
<tr>
<td>E1 mod E2</td>
<td>eval(E1) - ( eval(E1) / eval(E2) * eval(E2) )</td>
<td>I, I → I</td>
<td>Y</td>
</tr>
<tr>
<td>E1 \ E2</td>
<td>eval(E1) bitwise_and eval(E2)</td>
<td>I, I → I</td>
<td>Y</td>
</tr>
<tr>
<td>E1 ^ E2</td>
<td>eval(E1) bitwise_xor eval(E2)</td>
<td>I, I → I</td>
<td>Y</td>
</tr>
<tr>
<td>\ E</td>
<td>bitwise_not eval(E)</td>
<td>I → I</td>
<td>Y</td>
</tr>
<tr>
<td>E1 &lt;&lt; E2</td>
<td>eval(E1) integer_shift_left eval(E2)</td>
<td>I, I → I</td>
<td>Y</td>
</tr>
<tr>
<td>E1 &gt;&gt; E2</td>
<td>eval(E1) integer_shift_right eval(E2)</td>
<td>I, I → I</td>
<td>Y</td>
</tr>
<tr>
<td>abs(E)</td>
<td>absolute value of eval(E)</td>
<td>IF → IF</td>
<td>Y</td>
</tr>
<tr>
<td>sign(E)</td>
<td>sign of eval(E) (-1 if &lt; 0, 0 if = 0, +1 if &gt; 0)</td>
<td>IF → IF</td>
<td>Y</td>
</tr>
<tr>
<td>E1 ** E2</td>
<td>eval(E1) raised to the power of eval(E2)</td>
<td>IF, IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>sqrt(E)</td>
<td>square root of eval(E)</td>
<td>IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>atan(E)</td>
<td>arc tangent of eval(E)</td>
<td>IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>cos(E)</td>
<td>cosine of eval(E)</td>
<td>IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>sin(E)</td>
<td>sine of eval(E)</td>
<td>IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>exp(E)</td>
<td>e raised to the power of eval(E)</td>
<td>IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>log(E)</td>
<td>natural logarithms of eval(E)</td>
<td>IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>float(E)</td>
<td>the floating point number equal to eval(E)</td>
<td>IF → F</td>
<td>Y</td>
</tr>
<tr>
<td>ceiling(E)</td>
<td>rounds eval(E) upward to the nearest integer</td>
<td>F → I</td>
<td>Y</td>
</tr>
<tr>
<td>floor(E)</td>
<td>rounds eval(E) downward to the nearest integer</td>
<td>F → I</td>
<td>Y</td>
</tr>
<tr>
<td>round(E)</td>
<td>rounds eval(E) to the nearest integer</td>
<td>F → I</td>
<td>Y</td>
</tr>
<tr>
<td>truncate(E)</td>
<td>the integer value of eval(E)</td>
<td>F → I</td>
<td>Y</td>
</tr>
<tr>
<td>float_fractional_part(E)</td>
<td>the float equal to the fractional part of eval(E)</td>
<td>F → F</td>
<td>Y</td>
</tr>
<tr>
<td>float_integer_part(E)</td>
<td>the float equal to the integer part of eval(E)</td>
<td>F → F</td>
<td>Y</td>
</tr>
</tbody>
</table>

The meaning of the signature field is as follows:

- **I → I**: unary function, the operand must be an integer and the result is an integer.
- **F → F**: unary function, the operand must be a floating point number and the result is a floating point number.
- **F → I**: unary function, the operand must be a floating point number and the result is an integer.
- **IF → F**: unary function, the operand can be an integer or a floating point number and the result is a floating point number.
- **IF → IF**: unary function, the operand can be an integer or a floating point number and the result has the same type as the operand.
- **I, I → I**: binary function: each operand must be an integer and the result is an integer.
• IF, IF → IF: binary function: each operand can be an integer or a floating point number and the result is a floating point number if at least one operand is a floating point number, an integer otherwise.

is, +, -, *, //, /, rem, and mod are predefined infix operators. + and - are predefined prefix operators (section 6.14.10, page 87).

**Integer division rounding function**: the integer division rounding function \( \text{rnd}(x) \) rounds the floating point number \( x \) to an integer. There are two possible definitions (depending on the target machine) for this function which differ on negative numbers:

- \( \text{rnd}(x) = \text{integer part of } x \), e.g. \( \text{rnd}(-1.5) = -1 \) (round toward 0)
- \( \text{rnd}(x) = [x] \), e.g. \( \text{rnd}(-1.5) = -2 \) (round toward \(-\infty\))

The definition of this function determines the precise definition of the integer division \( \langle//\rangle/2 \) and of the integer remainder \( \langle\text{rem}\rangle/2 \). Rounding toward zero is the most common case. In any case it is possible to test the value (toward zero or down) of the integer rounding function Prolog flag to determine which function being used (section 6.22.1, page 117).

**Fast mathematical mode**: in order to speed-up integer computations, the GNU Prolog compiler can generate faster code when invoked with the \(--\text{fast-math}\) option (section 2.4.3, page 12). In this mode only integer operations are allowed and a variable in an expression must be bound at evaluation time to an integer. No type checking is done.

**Errors**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>a sub-expression E is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>a sub-expression E is neither a number nor an evaluable functor</td>
<td>type_error(evaluable, E)</td>
</tr>
<tr>
<td>a sub-expression E is a floating point number while an integer is expected</td>
<td>type_error(integer, E)</td>
</tr>
<tr>
<td>a sub-expression E is an integer while a floating point number is expected</td>
<td>type_error(float, E)</td>
</tr>
<tr>
<td>a division by zero occurs</td>
<td>evaluation_error(zero_divisor)</td>
</tr>
</tbody>
</table>

**Portability**

Refer to the above table to determine which evaluable functors are ISO and which are GNU Prolog extensions. For efficiency reasons, GNU Prolog does not detect the following ISO arithmetic errors: float_overflow, int_overflow, int_underflow, and undefined.

### 6.6.2 (is)/2 - evaluate expression

**Templates**

\[
is(?nonvar, +evaluable)
\]

**Description**

Result is Expression succeeds if Result can be unified with eval(Expression). Refer to the evaluation of an arithmetic expression for the definition of the eval function (section 6.6.1, page 45).

`is` is a predefined infix operator (section 6.14.10, page 87).

**Errors**
Refer to the evaluation of an arithmetic expression for possible errors (section 6.6.1, page 45).

Portability

ISO predicate.

6.6.3 \( (=\) /2 - arithmetic equal, \( =\) \( =\) /2 - arithmetic not equal, \\
\( <\) /2 - arithmetic less than, \( <=\) /2 - arithmetic less than or equal to, \\
\( >\) /2 - arithmetic greater than, \( >=\) /2 - arithmetic greater than or equal to\

Templates

\[
\begin{align*}
\texttt{=}(\textit{+evaluable, +evaluable}) & \quad \texttt{=} (\textit{+evaluable, +evaluable}) \\
\texttt{\=\=}(\textit{+evaluable, +evaluable}) & \quad \texttt{\=\=} (\textit{+evaluable, +evaluable}) \\
\texttt{\<}(\textit{+evaluable, +evaluable}) & \quad \texttt{\>} (\textit{+evaluable, +evaluable}) \\
\texttt{\<\<}(\textit{+evaluable, +evaluable}) & \quad \texttt{\>} (\textit{+evaluable, +evaluable})
\end{align*}
\]

Description

\( \texttt{Expr1 \:=\: \texttt{Expr2}} \) succeeds if \( \texttt{eval(Expr1)} = \texttt{eval(Expr2)} \).

\( \texttt{Expr1 \=\=} \texttt{Expr2} \) succeeds if \( \texttt{eval(Expr1)} \neq \texttt{eval(Expr2)} \).

\( \texttt{Expr1 \<} \texttt{Expr2} \) succeeds if \( \texttt{eval(Expr1)} < \texttt{eval(Expr2)} \).

\( \texttt{Expr1 \<\<} \texttt{Expr2} \) succeeds if \( \texttt{eval(Expr1)} \leq \texttt{eval(Expr2)} \).

\( \texttt{Expr1 \>} \texttt{Expr2} \) succeeds if \( \texttt{eval(Expr1)} > \texttt{eval(Expr2)} \).

\( \texttt{Expr1 \>} \texttt{Expr2} \) succeeds if \( \texttt{eval(Expr1)} \geq \texttt{eval(Expr2)} \).

Refer to the evaluation of an arithmetic expression for the definition of the \( \texttt{eval} \) function (section 6.6.1, page 45).

\( \texttt{\=\=}, \texttt{\=\=}, \texttt{\<}, \texttt{\=} \), and \( \texttt{\>} \) are predefined infix operators (section 6.14.10, page 87).

Errors

Refer to the evaluation of an arithmetic expression for possible errors (section 6.6.1, page 45).

Portability

ISO predicates.

6.7 Dynamic clause management

6.7.1 Introduction

Static and dynamic procedures: a procedure is either dynamic or static. All built-in predicates are static. A user-defined procedure is static by default unless a \texttt{dynamic/1} directive precedes its definition (section 5.1.2, page 29). Adding a clause to a non-existent procedure creates a dynamic procedure. The clauses of a dynamic procedure can be altered (e.g. using \texttt{asserta/1}), the clauses of a static procedure cannot be altered.
6.7 Dynamic clause management

**Private and public procedures**: each procedure is either public or private. A dynamic procedure is always public. Each built-in predicate is private, and a static user-defined procedure is private by default unless a public/1 directive precedes its definition (section 5.1.3, page 29). If a dynamic declaration exists it is unnecessary to add a public declaration since a dynamic procedure is also public. A clause of a public procedure can be inspected (e.g. using clause/2), a clause of a private procedure cannot be inspected.

**A logical database update view**: any change in the database that occurs as the result of executing a goal (e.g. when a sub-goal is a call of assertz/1 or retract/1) only affects subsequent activations. The change does not affect any activation that is currently being executed. Thus the database is frozen during the execution of a goal, and the list of clauses defining a predication is fixed at the moment of its execution.

### 6.7.2 asserta/1, assertz/1

**Templates**

```
asserta(+clause)
assertz(+clause)
```

**Description**

asserta(Clause) first converts the term `Clause` to a clause and then adds it to the current internal database. The predicate concerned must be dynamic (section 6.7.1, page 48) or undefined and the clause is inserted before the first clause of the predicate. If the predicated is undefined it is created as a dynamic procedure.

assertz(Clause) acts like asserta/1 except that the clause is added at the end of all existing clauses of the concerned predicate.

**Converting a term `Clause` to a clause `Clause1`:**

- extract the head and the body of `Clause`: either `Clause = (Head :- Body)` or `Clause = Head` and `Body = true`.
- `Head` must be a callable term (or else the conversion fails).
- convert `Body` to a body clause (i.e. a goal) `Body1`.
- the converted clause `Clause1 = (Head :- Body1)`.

**Converting a term `T` to a goal:**

- if `T` is a variable it is replaced by the term `call(T)`.
- if `T` is a control construct `(’,’)/2, (;)/2 or (->)/2` each argument of the control construct is recursively converted to a goal.
- if `T` is a callable term it remains unchanged.
- otherwise the conversion fails (`T` is neither a variable nor a callable term).

**Errors**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Head is neither a variable nor a callable term</td>
<td><code>type_error(callable, Head)</code></td>
</tr>
<tr>
<td>Body cannot be converted to a goal</td>
<td><code>type_error(callable, Body)</code></td>
</tr>
<tr>
<td>The predicate indicator <code>Pred</code> of <code>Head</code> is that of a static procedure</td>
<td><code>permission_error(modify, static_procedure, Pred)</code></td>
</tr>
</tbody>
</table>
Portability

ISO predicates.

6.7.3 retract/1

Templates

retract(+clause)

Description

retract(Clause) erases the first clause of the database that unifies with Clause. The concerned predicate must be a dynamic procedure (section 6.7.1, page 48). Removing all clauses of a procedure does not erase the procedure definition. To achieve this use abolish/1 (section 6.7.6, page 51). retract/1 is re-executable on backtracking.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Head is neither a variable nor a callable term</td>
<td>type_error(callable, Head)</td>
</tr>
<tr>
<td>The predicate indicator Pred of Head is that of a static procedure</td>
<td>permission_error(modify, static_procedure, Pred)</td>
</tr>
</tbody>
</table>

Portability

ISO predicate. In the ISO reference, the operation associated to the permission_error is access while it is modify in GNU Prolog. This seems to be an error of the ISO reference since for asserta/1 (which is similar in spirit to retract/1) the operation is also modify.

6.7.4 retractall/1

Templates

retractall(+head)

Description

retractall(Head) erases all clauses whose head unifies with Head. The concerned predicate must be a dynamic procedure (section 6.7.1, page 48). The procedure definition is not removed so that it is found by current_predicate/1 (section 6.8.1, page 52). abolish/1 should be used to remove the procedure (section 6.7.6, page 51).

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Head is not a callable term</td>
<td>type_error(callable, Head)</td>
</tr>
<tr>
<td>The predicate indicator Pred of Head is that of a static procedure</td>
<td>permission_error(modify, static_procedure, Pred)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.
6.7 Dynamic clause management

6.7.5 clause/2

Templates

\texttt{clause(+head, ?callable\_term)}

Description

\texttt{clause(Head, Body)} succeeds if there exists a clause in the database that unifies with \texttt{Head :- Body}. The predicate in question must be a public procedure (section 6.7.1, page 48). Clauses are delivered from the first to the last. This predicate is re-executable on backtracking.

Errors

\begin{array}{|l|l|}
\hline
\text{Head is a variable} & \text{instantiation\_error} \\
\text{Head is neither a variable nor a callable term} & \text{type\_error(calling, Head)} \\
\text{The predicate indicator \texttt{Pred} of \texttt{Head} is that of a private procedure} & \text{permission\_error(access, private\_procedure, Pred)} \\
\text{Body is neither a variable nor a callable term} & \text{type\_error(calling, Body)} \\
\hline
\end{array}

Portability

ISO predicate.

6.7.6 abolish/1

Templates

\texttt{abolish(+predicate\_indicator)}

Description

\texttt{abolish(Pred)} removes from the database the procedure whose predicate indicator is \texttt{Pred}. The concerned predicate must be a dynamic procedure (section 6.7.1, page 48).

Errors

\begin{array}{|l|l|}
\hline
\text{Pred is a variable} & \text{instantiation\_error} \\
\text{Pred is a term \texttt{Name/Arity} and either \texttt{Name} or \texttt{Arity} is a variable} & \text{instantiation\_error} \\
\text{Pred is neither a variable nor a predicate indicator} & \text{type\_error(predicate\_indicator, Pred)} \\
\text{Pred is a term \texttt{Name/Arity} and \texttt{Arity} is neither a variable nor an integer} & \text{type\_error(integer, Arity)} \\
\text{Pred is a term \texttt{Name/Arity} and \texttt{Name} is neither a variable nor an atom} & \text{type\_error(atom, Name)} \\
\text{Pred is a term \texttt{Name/Arity} and \texttt{Arity} is an integer} & \text{domain\_error(not\_less\_than\_zero, Arity)} \\
\text{Pred is a term \texttt{Name/Arity} and \texttt{Arity} is an integer} & \text{representation\_error(max\_arity)} \\
\text{The predicate indicator \texttt{Pred} is that of a static procedure} & \text{permission\_error(modify, static\_procedure, Pred)} \\
\hline
\end{array}

Portability
ISO predicate.

6.8 Predicate information

6.8.1 current_predicate/1

Templates

\texttt{current_predicate(?predicate\_indicator)}

\textbf{Description}

current\_predicate(Pred) succeeds if there exists a predicate indicator of a defined procedure that unifies with Pred. All user defined procedures are found, whether static or dynamic. Internal system procedures whose name begins with \\
'\$' are not found. A user-defined procedure is found even when it has no clauses. A user-defined procedure is not found if it has been abolished. To conform to the ISO reference, built-in predicates are not found except if the \texttt{strict\_iso} Prolog flag is switched off (section 6.22.1, page 117). This predicate is re-executable on backtracking.

\textbf{Errors}

<table>
<thead>
<tr>
<th>Pred is neither a variable nor a predicate indicator</th>
<th>\texttt{type_error(predicate_indicator, Pred)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pred is a term Name/Arity and Arity is neither a variable nor an integer</td>
<td>\texttt{type_error(integer, Arity)}</td>
</tr>
<tr>
<td>Pred is a term Name/Arity and Name is neither a variable nor an atom</td>
<td>\texttt{type_error(atom, Name)}</td>
</tr>
<tr>
<td>Pred is a term Name/Arity and Arity is an integer &lt; 0</td>
<td>\texttt{domain_error(not_less_than_zero, Arity)}</td>
</tr>
<tr>
<td>Pred is a term Name/Arity and Arity is an integer &gt; max_arity flag (section 6.22.1, page 117)</td>
<td>\texttt{representation_error(max_arity)}</td>
</tr>
</tbody>
</table>

\textbf{Portability}

ISO predicate.

6.8.2 predicate\_property/2

\textbf{Templates}

\texttt{predicate\_property(?predicate\_indicator, ?predicate\_property)}

\textbf{Description}

c\texttt{predicate\_property(Pred, Property)} succeeds if \texttt{current\_predicate(Pred)} succeeds (section 6.8.1, page 52) and if Property unifies with one of the properties of the procedure. This predicate is re-executable on backtracking.

\textbf{Predicate properties:}

- \texttt{static}: if the procedure is static.
- \texttt{dynamic}: if the procedure is dynamic.
6.9 All solutions

6.9.1 Introduction

It is sometimes useful to collect all solutions for a goal. This can be done by repeatedly backtracking and gradually building the list of solutions. The following built-in predicates are provided to automate this process.

The built-in predicates described in this section invoke call/1 (section 5.2.3, page 35) on the argument Goal. When efficiency is crucial and Goal is complex it is better to define an auxiliary predicate which can then be compiled, and have Goal call this predicate.

6.9.2 findall/3

Templates
findall(?term, +callable_term, ?list)

Description

findall(Template, Goal, Instances) succeeds if Instances unifies with the list of values to which a variable X not occurring in Template or Goal would be instantiated by successive re-executions of call(Goal), X = Template after systematic replacement of all variables in X by new variables. Thus, the order of the list Instances corresponds to the order in which the proofs are found.

Errors

| Goal is a variable                         | instantiation_error |
| Goal is neither a variable nor a callable term | type_error(callable, Goal) |
| The predicate indicator Pred of Goal does not correspond to an existing procedure and the value of the unknown Prolog flag is error (section 6.22.1, page 117) | existence_error(procedure, Pred) |
| Instances is neither a partial list nor a list | type_error(list, Instances) |

Portability

ISO predicate.

6.9.3 bagof/3, setof/3

Templates

```
bagof(?term, +callable_term, ?list)
setof(?term, +callable_term, ?list)
```

Description

bagof(Template, Goal, Instances) assembles a list the set of solutions of Goal for each different instantiation of the free variables in Goal. The elements of each list are in order of solution, but the order in which each list is found is undefined. This predicate is re-executable on backtracking.

**Free variable set:** bagof/3 groups the solutions of Goal according to the free variables in Goal. This set corresponds to all variables occurring in Goal but not in Template. It is sometimes useful to exclude some additional variables of Goal. For that, bagof/3 recognizes a goal of the form \( T \text{goals} \) and exclude all variables occurring in \( T \) from the free variable set. \((\cdot)\)/2 can be viewed as an *existential quantifier* (the logical reading of \( \exists X \text{goal} \) being “there exists an \( X \) such that Goal is true”). The use of this existential qualifier is superfluous outside bagof/3 (and setof/3) and then is not recognized.

\((\cdot)\)/2 is a predefined infix operator (section 6.14.10, page 87).

setof(Template, Goal, Instances) is equivalent to bagof(Template,Goal,I), sort(I,Instances). Each list is then a sorted list (duplicate elements are removed).

From the implementation point of view setof/3 is as fast as bagof/3. Both predicates use an in-place (i.e. destructive) sort (section 6.20.12, page 112) and require the same amount of memory.

Errors
6.10 Streams

6.10.1 Introduction

A stream provides a logical view of a source/sink.

**Sources and sinks:** a program can output results to a sink or input data from a source. A source/sink may be a file (regular file, terminal, device,...), a constant term, a pipe, a socket,...

**Associating a stream to a source/sink:** to manipulate a source/sink it must be associated to a stream. This provides a logical and uniform view of the source/sink whatever its type. Once this association has been established, i.e. a stream has been created, all subsequent references to the source/sink are made by referring to the stream. A stream is unidirectional: it is either an input stream or an output stream. For a classical file, the association is done by opening the file (whose name is specified as an atom) with the `open/4` (section 6.10.6, page 58). GNU Prolog makes it possible to treat a Prolog constant term as a source/sink and provides built-in predicates to associate a stream to such a term (section 6.11, page 69). GNU Prolog provides operating system interface predicates defining pipes between GNU Prolog and child processes with streams associated to these pipes, e.g. `popen/3` (section 6.27.21, page 137). Similarly, socket interface predicates associate streams to a socket to allow the communication, e.g. `socket_connect/4` (section 6.28.5, page 143).

**Stream-term:** a stream-term identifies a stream during a call of an input/output built-in predicate. It is created as a result of associating a stream to a source/sink (section above). A stream-term is a compound term of the form `

**Stream aliases:** any stream may be associated with a stream alias which is an atom which may be used to refer to that stream. The association automatically ends when the stream is closed. A particular alias only refers to at most one stream at any one time. However, more than one alias can be associated to a stream. All built-in predicates which have a stream-term as an input argument also accept a stream alias as that argument. However, built-in predicates which return a stream-term do not accept a stream alias.

**Standard streams:** two streams are predefined and open during the execution of every goal: the standard input stream which has the alias `user_input` and the standard output stream which has the alias `user_output`. A goal which attempts to close either standard stream succeeds, but does not close the stream.

**Current streams:** during execution there is a current input stream and a current output stream. By default, the current input and output streams are the standard input and output streams, but the built-in predicates `set_input/1` (section 6.10.4, page 57) and `set_output/1` (section 6.10.5, page 57) can be used to change them. When the current input stream is closed, the standard input stream becomes the current input stream. When the current output stream is closed, the standard output stream becomes...
the current output stream.

**Text streams and binary streams:** a text stream is a sequence of characters. A text stream is also regarded as a sequence of lines where each line is a possibly empty sequence of characters followed by a new line character. GNU Prolog may add or remove space characters at the ends of lines in order to conform to the conventions for representing text streams in the operating system. A binary stream is a sequence of bytes. Only a few built-in predicates can deal with binary streams, e.g. `get_byte/2` (section 6.13, page 76).

**Stream positions:** the stream position of a stream identifies an absolute position of the source/sink to which the stream is connected and defines where in the source/sink the next input or output will take place. A stream position is a ground term of the form `'$stream_position'(I1, I2, I3, I4)` where I1, I2, I3 and I4 are integers. Stream positions are used to reposition a stream (when possible) using for instance `set_stream_position/2` (section 6.10.13, page 63).

**The position end of stream:** when all data of a stream S has been input S has a stream position end-of-stream. At this stream position a goal to input more data will return a specific value to indicate that end of stream has been reached (e.g. -1 for `get_code/2` or `end_of_file` for `get_char/2`, ...). When this terminating value has been input, the stream has a stream position past-end-of-stream.

**Buffering mode:** input/output on a stream can be buffered (line-buffered or block-buffered) or not buffered at all. The buffering mode can be specified at open time or using `set_stream_buffering/2` (section 6.10.24, page 68). Line buffering is used on output streams, output data are only written to the sink when a new-line character is output (or at the close time). Block buffering is used on input or output. On input streams, when an input is requested on the source, if the buffer is empty, all available characters are read (within the limits of the size of the buffer), subsequent reads will first use the characters in the buffer. On output streams, output data are stored in the buffer and only when the buffer is full is it physically written on the sink. Thus, an output to a buffered stream may not be sent immediately to the sink connected to that stream. When it is necessary to be certain that output has been delivered, the built-in predicate `flush_output/1` (section 6.10.8, page 60) should be used. Finally, it is also possible to use non-buffered streams, in that case input/output are directly done on the connected source/sink. This can be useful for communication purposes (e.g. sockets) or when a precise control is needed, e.g. `select/5` (section 6.27.23, page 139).

### 6.10.2 current_input/1

**Templates**

`current_input(?stream)`

**Description**

`current_input(Stream)` unifies `Stream` with the stream-term identifying the current input stream.

**Errors**

| Stream is neither a variable nor a stream | domain_error(stream, Stream) |

**Portability**

ISO predicate.
6.10 Streams

6.10.3 current_output/1

Templates

```clojure
current_output(Stream)
```

Description
current_output(Stream) unifies Stream with the stream-term identifying the current output stream.

Errors

| Stream is neither a variable nor a stream | domain_error(stream, Stream) |

Portability

ISO predicate.

6.10.4 set_input/1

Templates

```clojure
set_input(+stream_or_alias)
```

Description

set_input(Stream) sets the current input stream to be the stream associated with the stream-term or alias Stream.

Errors

| Stream is a variable | instantiation_error |
| Stream is neither a variable nor a stream-term or alias | domain_error(stream_or_alias, Stream) |
| Stream is not associated with an open stream | existence_error(stream, Stream) |
| Stream is an output stream | permission_error(input, stream, Stream) |

Portability

ISO predicate.

6.10.5 set_output/1

Templates

```clojure
set_output(+stream_or_alias)
```

Description

set_output(Stream) sets the current output stream to be the stream associated with the stream-term or alias Stream.

Errors
| SorA is a variable | instantiation_error |
| SorA is neither a variable nor a stream-term or alias | domain_error(stream_or_alias, SorA) |
| SorA is not associated with an open stream | existence_error(stream, SorA) |
| SorA is an input stream | permission_error(output, stream, SorA) |

Portability

ISO predicate.

6.10.6 open/4, open/3

Templates

open(+source_sink, +io_mode, -stream, +stream_option_list)
open(+source_sink, +io_mode, -stream)

Description

`open(SourceSink, Mode, Stream, Options)` opens the source/sink `SourceSink` for input or output as indicated by `Mode` and the list of stream-options `Options` and unifies `Stream` with the stream-term which is associated to this stream. See `absolute_file_name/2` for information about the syntax of `SourceSink` (section 6.26.1, page 125).

Input/output modes: `Mode` is an atom which defines the input/output operations that may be performed the stream. Possible modes are:

- **read**: the source/sink is a source and must already exist. Input starts at the beginning of the source.

- **write**: the source/sink is a sink. If the sink already exists then it is emptied else an empty sink is created. Output starts at the beginning of that sink.

- **append**: the source/sink is a sink. If the sink does not exist it is created. Output starts at the end of that sink.

Stream options: `Options` is a list of stream options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- **type(text/binary)**: specifies whether the stream is a text stream or a binary stream. The default value is text.

- **reposition(true/false)**: specifies whether it is possible to reposition the stream. The default value is true except if the stream cannot be repositioned (e.g. a terminal).

- **eof_action(error/eof_code/reset)**: specifies the effect of attempting to input from a stream whose stream position is past-end-of-stream:
  - **error**: a `permission_error` is raised signifying that no more input exists in this stream.
  - **eof_code**: the result of input is as if the stream position is end-of-stream.
  - **reset**: the stream position is reset so that it is not past-end-of-stream, and another attempt is made to input from it (this is useful when inputting from a terminal).
  The default value is `eof_code`.

- **alias(Alias)**: specifies that the atom `Alias` is to be an alias for the stream. By default no alias is attached to the stream.

- **buffering(none/line/block)**: specifies which type of buffering is used by input/output operations on this stream:
- **none**: no buffering.
- **line**: output operations buffer data emitted until a new-line occurs
- **block**: input/output operations buffer data until a given number (implementation dependent) of characters/bytes have been treated.

The default value is **line** for a terminal (TTY), **block** otherwise.

```
open(SourceSink, Mode, Stream, Options) is equivalent to open(SourceSink, Mode, Stream, []).
```

### Errors

<table>
<thead>
<tr>
<th>SourceSink is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Options is a partial list or a list with an element E which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Mode is neither a variable nor an atom</td>
<td>type_error(atom, Mode)</td>
</tr>
<tr>
<td>Options is neither a partial list nor a list</td>
<td>type_error(list, Options)</td>
</tr>
<tr>
<td>Stream is not a variable</td>
<td>type_error(variable, Stream)</td>
</tr>
<tr>
<td>SourceSink is neither a variable nor a source/sink</td>
<td>domain_error(source_sink, SourceSink)</td>
</tr>
<tr>
<td>Mode is an atom but not an input/output mode</td>
<td>domain_error(io_mode, Mode)</td>
</tr>
<tr>
<td>an element E of the Options list is neither a variable nor a stream-option</td>
<td>domain_error(stream_option, E)</td>
</tr>
<tr>
<td>the source/sink specified by SourceSink does not exist</td>
<td>existence_error(source_sink, SourceSink)</td>
</tr>
<tr>
<td>the source/sink specified by SourceSink cannot be opened</td>
<td>permission_error(open, source_sink, SourceSink)</td>
</tr>
<tr>
<td>an element E of the Options list is alias(A) and A is already associated with an open stream</td>
<td>permission_error(open, source_sink, alias(A))</td>
</tr>
<tr>
<td>an element E of the Options list is reposition(true) and it is not possible to reposition this stream</td>
<td>permission_error(open, source_sink, reposition(true))</td>
</tr>
</tbody>
</table>

### Portability

ISO predicates. The **buffering/1** stream option is a GNU Prolog extension.

#### 6.10.7 close/2, close/1

### Templates

```
close(+stream_or_alias, +close_option_list)
close(+stream_or_alias)
```

### Description

close(Stream) closes the stream associated with the stream-term or alias Stream. If Stream is the standard input stream or the standard output stream close/2 simply succeeds else the associated source/sink is physically closed. If Stream is the current input stream the current input stream becomes the standard input stream user_input. If Stream is the current output stream the current output stream becomes the standard output stream user_output.

**Close options**: Options is a list of close options. For the moment only one option is available:

- **force(true/false)**: with **false**, if an error occurs when trying to close the source/sink, the stream is not closed and an error (**system_error** or **resource_error**) is raised (but close/2 succeeds).
With true, if an error occurs it is ignored and the stream is closed. The purpose of force/1 option is to allow an error handling routine to do its best to reclaim resources. The default value is false. close(SorA) is equivalent to close(SorA, []).

Errors

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options is a partial list or a list with an element E which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Options is neither a partial list nor a list</td>
<td>type_error(list, Options)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>an element E of the Options list is neither a variable nor a close-option</td>
<td>domain_error(close_option, E)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA needs a special close (section 6.11, page 69)</td>
<td>system_error(needs_special_close)</td>
</tr>
</tbody>
</table>

Portability

ISO predicates. The system_error(needs_special_close) is a GNU Prolog extension.

6.10.8 flush_output/1, flush_output/0

Templates

    flush_output(+stream_or_alias)
    flush_output

Description

flush_output(SorA) sends any buffered output characters/bytes to the stream.

flush_output/0 applies to the current output stream.

Errors

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an input stream</td>
<td>permission_error(output, stream, SorA)</td>
</tr>
</tbody>
</table>

Portability

ISO predicates.

6.10.9 current_stream/1

Templates

    current_stream(?stream)

Description
current_stream(Stream) succeeds if there exists a stream-term that unifies with Stream. This predicate is re-executable on backtracking.

Errors

| Stream is neither a variable nor a stream-term | domain_error(stream, Stream) |

Portability

GNU Prolog predicate.

6.10.10 stream_property/2

Templates

stream_property(?stream, ?stream_property)

Description

stream_property(Stream, Property) succeeds if current_stream(Stream) succeeds (section 6.10.9, page 60) and if Property unifies with one of the properties of the stream. This predicate is re-executable on backtracking.

Stream properties:

- file_name(F): the name of the connected source/sink.
- mode(M): M is the open mode (read, write, append).
- input: if it is an input stream.
- output: if it is an output stream.
- alias(A): A is an alias of the stream.
- type(T): T is the type of the stream (text, binary).
- reposition(R): R is the reposition boolean (true, false).
- eof_action(A): A is the end-of-file action (error, eof_code, reset).
- buffering(B): B is the buffering mode (none, line, block).
- end_of_stream(E): E is the current end-of-stream status (not, at, past). If the stream position is end-of-stream then E is unified with at else if the stream position is past-end-of-stream then E is unified with past else E is unified with not.
- position(P): P is the stream-position term associated to the current position.

Errors

| Stream is a variable | instantiation_error |
| Stream is neither a variable nor a stream-term | domain_error(stream, Stream) |
| Property is neither a variable nor a stream property | domain_error(stream_property, Property) |
| Property = file_name(E), mode(E), alias(E), end_of_stream(E), eof_action(E), reposition(E), type(E) or buffering(E) and E is neither a variable nor an atom | type_error(atom, E) |
Portability
ISO predicate. The buffering/1 property is a GNU Prolog extension.

6.10.11 at_end_of_stream/1, at_end_of_stream/0

Templates

\[
\text{at_end_of_stream(+stream_or_alias)} \\
\text{at_end_of_stream}
\]

Description

at_end_of_stream(SorA) succeeds if the stream associated with stream-term or alias SorA has a stream position end-of-stream or past-end-of-stream. This predicate can be defined using stream_property/2 (section 6.10.10, page 61).

at_end_of_stream/0 applies to the current input stream.

Errors

\[
\begin{array}{|l|l|}
\hline
\text{SorA is a variable} & \text{instantiation_error} \\
\text{SorA is neither a variable nor a stream-term or alias} & \text{domain_error(stream_or_alias, SorA)} \\
\text{SorA is not associated with an open stream} & \text{existence_error(stream, SorA)} \\
\text{SorA is an output stream} & \text{permission_error(input, stream, SorA)} \\
\hline
\end{array}
\]

Portability
ISO predicates. The permission_error(input, stream, SorA) is a GNU Prolog extension.

6.10.12 stream_position/2

Templates

\[
\text{stream_position(+stream_or_alias, ?stream_position)}
\]

Description

stream_position(SorA, Position) succeeds unifying Position with the stream-position term associated to the current position of the stream-term or alias SorA. This predicate can be defined using stream_property/2 (section 6.10.10, page 61).

Errors

\[
\begin{array}{|l|l|}
\hline
\text{SorA is a variable} & \text{instantiation_error} \\
\text{SorA is neither a variable nor a stream-term or alias} & \text{domain_error(stream_or_alias, SorA)} \\
\text{Position is neither a variable nor a stream-position term} & \text{domain_error(stream_position, Position)} \\
\text{SorA is not associated with an open stream} & \text{existence_error(stream, SorA)} \\
\hline
\end{array}
\]

Portability
GNU Prolog predicate.
6.10 Streams

6.10.13 set_stream_position/2

Templates

\[
\text{set\_stream\_position}(\text{+stream\_or\_alias}, \text{+stream\_position})
\]

Description

\text{set\_stream\_position}(\text{SorA, Position}) \text{ sets the position of the stream associated with the stream-term or alias SorA to Position. Position should have previously been returned by stream\_property/2 (section 6.10.10, page 61) or by stream\_position/2 (section 6.10.12, page 62).}

Errors

<table>
<thead>
<tr>
<th>Error condition</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{SorA} is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Position is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>\text{SorA} is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>Position is neither a variable nor a stream-position term</td>
<td>domain_error(stream_position, Position)</td>
</tr>
<tr>
<td>\text{SorA} is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>\text{SorA} has stream property reposition(false)</td>
<td>permission_error(reposition, stream, SorA)</td>
</tr>
</tbody>
</table>

Portability

ISO predicate.

6.10.14 seek/4

Templates

\[
\text{seek}(\text{+stream\_or\_alias}, \text{+stream\_seek\_method}, \text{+integer}, \text{?integer})
\]

Description

\text{seek(SorA, Whence, Offset, NewOffset)} \text{ sets the position of the stream associated with the stream-term or alias SorA to Offset according to Whence and unifies NewOffset with the new offset from the beginning of the file. seek/4 can only be used on binary streams. Whence is an atom from:

- \text{bof}: the position is set relatively to the begin of the file (Offset should be \( \geq 0 \)).
- \text{current}: the position is set relatively to the current position (Offset can be \( \geq 0 \) or \( \leq 0 \)).
- \text{eof}: the position is set relatively to the end of the file (Offset should be \( \leq 0 \)).

This predicate is an interface to the C Unix function \text{1seek(2)}.}

Errors
<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whence is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Offset is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>Whence is neither a variable nor an atom</td>
<td>type_error(atom, Whence)</td>
</tr>
<tr>
<td>Whence is an atom but not a valid stream seek method</td>
<td>domain_error(stream_seek_method, Whence)</td>
</tr>
<tr>
<td>Offset is neither a variable nor an integer</td>
<td>type_error(integer, Offset)</td>
</tr>
<tr>
<td>NewOffset is neither a variable nor an integer</td>
<td>type_error(integer, NewOffset)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA has stream property reposition (false)</td>
<td>permission_error(reposition, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a text stream</td>
<td>permission_error(reposition, text_stream, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.10.15 \texttt{character\_count/2}

**Templates**

\texttt{character\_count(+stream\_or\_alias, ?integer)}

**Description**

\texttt{character\_count(SorA, Count)} unifies Count with the number of characters/bytes read/written on the stream associated with stream-term or alias SorA.

**Errors**

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count is neither a variable nor an integer</td>
<td>type_error(integer, Count)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.10.16 \texttt{line\_count/2}

**Templates**

\texttt{line\_count(+stream\_or\_alias, ?integer)}

**Description**

\texttt{line\_count(SorA, Count)} unifies Count with the number of lines read/written on the stream associated with the stream-term or alias SorA. This predicate can only be used on text streams.
6.10 Streams

Errors

| SorA is a variable | instantiation_error |
| Count is neither a variable nor an integer | type_error(integer, Count) |
| SorA is neither a variable nor a stream-term or alias | domain_error(stream_or_alias, SorA) |
| SorA is not associated with an open stream | existence_error(stream, SorA) |
| SorA is associated with a binary stream | permission_error(access, binary_stream, SorA) |

Portability

GNU Prolog predicate.

6.10.17 line_position/2

Templates

    line_position(+stream_or_alias, ?integer)

Description

line_position(SorA, Count) unifies Count with the number of characters read/written on the current line of the stream associated with the stream-term or alias SorA. This predicate can only be used on text streams.

Errors

| SorA is a variable | instantiation_error |
| Count is neither a variable nor an integer | type_error(integer, Count) |
| SorA is neither a variable nor a stream-term or alias | domain_error(stream_or_alias, SorA) |
| SorA is not associated with an open stream | existence_error(stream, SorA) |
| SorA is associated with a binary stream | permission_error(access, binary_stream, SorA) |

Portability

GNU Prolog predicate.

6.10.18 stream_line_column/3

Templates

    stream_line_column(+stream_or_alias, ?integer, ?integer)

Description

stream_line_column(SorA, Line, Column) unifies Line (resp. Column) with the current line number (resp. column number) of the stream associated with the stream-term or alias SorA. This predicate can only be used on text streams. Note that Line corresponds to the value returned by line_count/2 + 1 (section 6.10.16, page 64) and Column to the value returned by line_position/2 + 1 (section 6.10.17, page 65).

Errors
<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line is neither a variable nor an integer</td>
<td>type_error(integer, Line)</td>
</tr>
<tr>
<td>Column is neither a variable nor an integer</td>
<td>type_error(integer, Column)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a binary stream</td>
<td>permission_error(reposition, binary_stream, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.10.19 set_stream_line_column/3

**Templates**

```prolog
set_stream_line_column(+stream_or_alias, +integer, +integer)
```

**Description**

The `set_stream_line_column/3` predicate sets the stream position of the stream associated with the stream-term or alias `SorA` according to the line number `Line` and the column number `Column`. This predicate can only be used on text streams. It first repositions the stream to the beginning of the file and then reads character by character until the required position is reached.

**Errors**

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Column is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Line is neither a variable nor an integer</td>
<td>type_error(integer, Line)</td>
</tr>
<tr>
<td>Column is neither a variable nor an integer</td>
<td>type_error(integer, Column)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a binary stream</td>
<td>permission_error(reposition, binary_stream, SorA)</td>
</tr>
<tr>
<td>SorA has stream property reposition(false)</td>
<td>permission_error(reposition, stream, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.10.20 add_stream_alias/2

**Templates**

```prolog
add_stream_alias(+stream_or_alias, +atom)
```

**Description**

The `add_stream_alias/2` predicate adds `Alias` as a new alias to the stream associated with the stream-term or alias `SorA`.
6.10 Streams

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SorA is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Alias is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Alias is neither a variable nor an atom</td>
<td>type_error(Atom, Alias)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(Stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(Stream, SorA)</td>
</tr>
<tr>
<td>Alias is already associated with an open stream</td>
<td>permission_error(add_alias, Source_Sink, alias(Alias))</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.10.21 current_alias/2

Templates

```prolog
current_alias(?stream, ?atom)
```

Description

current_alias(Stream, Alias) succeeds if current_stream(Stream) succeeds (section 6.10.9, page 60) and if Alias unifies with one of the aliases of the stream. It can be defined using stream_property/2 (section 6.10.10, page 61). This predicate is re-executable on backtracking.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream is neither a variable nor a stream-term</td>
<td>domain_error(Stream, Stream)</td>
</tr>
<tr>
<td>Alias is neither a variable nor an atom</td>
<td>type_error(Atom, Alias)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.10.22 set_stream_type/2

Templates

```prolog
set_stream_type(+stream_or_alias, +atom)
```

Description

set_stream_type(SorA, Type) updates the type associated with stream-term or alias SorA. The value of Type is an atom in text or binary as for open/4 (section 6.10.6, page 58). The type of a stream can only be changed before any input/output operation is executed.

Errors
### Portability

GNU Prolog predicate.

#### 6.10.23 set_stream_eof_action/2

**Templates**

\[
\text{set\_stream\_eof\_action}(+\text{stream}_\text{or}_\text{alias}, +\text{atom})
\]

**Description**

`set_stream_eof_action(SorA, Action)` updates the `eof_action` option associated with the stream-term or alias `SorA`. The value of `Action` is one of the atoms `error`, `eof_code`, `reset` as for `open/4` (section 6.10.6, page 58).

**Errors**

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Action is neither a variable nor a valid <code>eof_action</code></td>
<td>domain_error(<code>eof_action</code>, <code>Action</code>)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(<code>stream_or_alias</code>, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(<code>stream</code>, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(<code>modify</code>, <code>stream</code>, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

#### 6.10.24 set_stream_buffering/2

**Templates**

\[
\text{set\_stream\_buffering}(+\text{stream}_\text{or}_\text{alias}, +\text{atom})
\]

**Description**

`set_stream_buffering(SorA, Buffering)` updates the buffering mode associated with the stream-term or alias `SorA`. The value of `Buffering` is one of the atoms `none`, `line` or `block` as for `open/4` (section 6.10.6, page 58).

**Errors**

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffering is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Buffering is neither a variable nor a valid <code>buffering</code></td>
<td>domain_error(<code>buffering</code>, Buffering)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(<code>stream_or_alias</code>, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(<code>stream</code>, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(<code>modify</code>, <code>stream</code>, SorA)</td>
</tr>
<tr>
<td>SorA is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Buffering is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Buffering is neither a variable nor a valid buffering mode</td>
<td>domain_error(buffering_mode, Buffering)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.11 Constant term streams

6.11.1 Introduction

Constant term streams allow the user to consider a constant term (atom, character list or character code list) as a source/sink by associating to them a stream. Reading from a constant term stream will deliver the characters of the constant term as if they had been read from a standard file. Characters written on a constant term stream are stored to form the final constant term when the stream is closed. The built-in predicates described in this section allow the user to open and close a constant term stream for input or output. However, very often, a constant term stream is created to be only read or written once and then closed. To avoid the creation and the destruction of such a stream, GNU Prolog offers several built-in predicates to perform single input/output from/to constant terms (section 6.15, page 91).

6.11.2 open_input_atom_stream/2, open_input_chars_stream/2, open_input_codes_stream/2

Templates

open_input_atom_stream(+atom, -stream)
open_input_chars_stream(+character_list, -stream)
open_input_codes_stream(+character_code_list, -stream)

Description

open_input_atom_stream(Atom, Stream) unifies Stream with the stream-term which is associated to a new input text-stream whose data are the characters of Atom.

open_input_chars_stream(Chars, Stream) is similar to open_input_atom_stream/2 except that data are the content of the character list Chars.

open_input_codes_stream(Codes, Stream) is similar to open_input_atom_stream/2 except that data are the content of the character code list Codes.

Errors
<table>
<thead>
<tr>
<th>Stream is not a variable</th>
<th>type_error(variable, Stream)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Chars is a partial list or a list with an element E which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Codes is a partial list or a list with an element E which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Atom is neither a variable nor an atom</td>
<td>type_error(atom, Atom)</td>
</tr>
<tr>
<td>Chars is neither a partial list nor a list</td>
<td>type_error(list, Chars)</td>
</tr>
<tr>
<td>Codes is neither a partial list nor a list</td>
<td>type_error(list, Codes)</td>
</tr>
<tr>
<td>an element E of the Chars list is neither a variable nor a character</td>
<td>type_error(character, E)</td>
</tr>
<tr>
<td>an element E of the Codes list is neither a variable nor an integer</td>
<td>type_error(integer, E)</td>
</tr>
<tr>
<td>an element E of the Codes list is an integer but not a character code</td>
<td>representation_error(character_code)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

6.11.3 close_input_atom_stream/1, close_input_chars_stream/1, close_input_codes_stream/1

**Templates**

- close_input_atom_stream(+stream_or_alias)
- close_input_chars_stream(+stream_or_alias)
- close_input_codes_stream(+stream_or_alias)

**Description**

`close_input_atom_stream(SorA)` closes the constant term stream associated with the stream-term or alias `SorA`. `SorA` must be a stream open with `open_input_atom_stream/2` (section 6.11.1, page 69).

`close_input_chars_stream(SorA)` acts similarly for a character list stream.

`close_input_codes_stream(SorA)` acts similarly for a character code list stream.

**Errors**

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(close, stream, SorA)</td>
</tr>
<tr>
<td>SorA is a stream-term or alias but does not refer to a constant term stream.</td>
<td>domain_error(term_stream_or_alias, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.
6.11.4 \texttt{open\_output\_atom\_stream}/1, \texttt{open\_output\_chars\_stream}/1, \\
\texttt{open\_output\_codes\_stream}/1

**Templates**

\begin{verbatim}
open_output_atom_stream(-stream)
open_output_chars_stream(-stream)
open_output_codes_stream(-stream)
\end{verbatim}

**Description**

\texttt{open\_output\_atom\_stream(Stream)} unifies \texttt{Stream} with the stream-term which is associated to a new output text-stream. All characters written to this stream are collected and will be returned as an atom when the stream is closed by \texttt{close\_output\_atom\_stream}/2 (section 6.11.5, page 71).

\texttt{open\_output\_chars\_stream(Stream)} is similar to \texttt{open\_output\_atom\_stream}/1 except that the result will be a character list.

\texttt{open\_output\_codes\_stream(Stream)} is similar to \texttt{open\_output\_atom\_stream}/1 except that the result will be a character code list.

**Errors**

\begin{center}
\begin{tabular}{|l|l|}
\hline
\texttt{Stream is not a variable} & \texttt{type\_error(variable, Stream)} \\
\hline
\end{tabular}
\end{center}

**Portability**

GNU Prolog predicates.

6.11.5 \texttt{close\_output\_atom\_stream}/2, \texttt{close\_output\_chars\_stream}/2, \\
\texttt{close\_output\_codes\_stream}/2

**Templates**

\begin{verbatim}
close_output_atom_stream(+stream_or_alias, ?atom)
close_output_chars_stream(+stream_or_alias, ?character_list)
close_output_codes_stream(+stream_or_alias, ?character_code_list)
\end{verbatim}

**Description**

\texttt{close\_output\_atom\_stream(SorA, Atom)} closes the constant term stream associated with the stream-term or alias \texttt{SorA}. \texttt{SorA} must be associated to a stream open with \texttt{open\_output\_atom\_stream}/1 (section 6.11.4, page 71). \texttt{Atom} is unified with an atom formed with all characters written on the stream.

\texttt{close\_output\_chars\_stream(SorA, Chars)} acts similarly for a character list stream.

\texttt{close\_output\_codes\_stream(SorA, Codes)} acts similarly for a character code list stream.

**Errors**
<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom is neither a variable nor an atom</td>
<td>type_error(atom, Atom)</td>
</tr>
<tr>
<td>Chars is neither a partial list nor a list</td>
<td>type_error(list, Chars)</td>
</tr>
<tr>
<td>Codes is neither a partial list nor a list</td>
<td>type_error(list, Codes)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an input stream</td>
<td>permission_error(close, stream, SorA)</td>
</tr>
<tr>
<td>SorA is a stream-term or alias but does not refer to a constant term stream</td>
<td>domain_error(term_stream_or_alias, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

### 6.12 Character input/output

These built-in predicates enable a single character or character code to be input from and output to a text stream. The atom `end_of_file` is returned as character to indicate the end-of-file. -1 is returned as character code to indicate the end-of-file.

#### 6.12.1 get_char/2, get_char/1, get_code/1, get_code/2

**Templates**

```prolog
get_char(+stream_or_alias, ?in_character)
get_char(?in_character)
get_code(+stream_or_alias, ?in_character_code)
get_code(?in_character_code)
```

**Description**

*get_char*(SorA, Char) succeeds if Char unifies with the next character read from the stream associated with the stream-term or alias SorA.

*get_code*/2 is similar to *get_char*/2 but deals with character codes.

*get_char*/1 and *get_code*/1 apply to the current input stream.

**Errors**

Portability

ISO predicates.

6.12.2 get_code_no_echo/2, get_code_no_echo/1

Templates

get_code_no_echo(+stream_or_alias, ?integer)
get_code_no_echo(?integer)

Description

get_code_no_echo/2 is similar to get_code/2 (section 6.12.1, page 72) except that if the associated source is a terminal the input is not echoed. This is possible only if the linedit facility has been installed (section 2.2.5, page 8) otherwise get_code_no_echo/2 is similar to get_code/2. Any integer can be returned (not only an in-character code) to allow for function key reading.

get_code_no_echo/1 applies to the current input stream.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SorA is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Code is neither a variable nor an in-character</td>
<td>type_error(integer, Code)</td>
</tr>
<tr>
<td>SorA is neither a variable nor an integer</td>
<td>type_error(integer, Code)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>or alias</td>
<td></td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(input, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a binary stream</td>
<td>permission_error(input, binary_stream, SorA)</td>
</tr>
<tr>
<td>SorA has stream properties</td>
<td>permission_error(input, past_end_of_stream, SorA)</td>
</tr>
<tr>
<td>end_of_stream(past) and eof_action(error)</td>
<td></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.
6.12.3 peek_char/2, peek_char/1, peek_code/1, peek_code/2

Templates

\[
\begin{align*}
\text{peek_char} & (\text{+stream_or_alias}, \text{?in_character}) \\
\text{peek_char} & (\text{?in_character}) \\
\text{peek_code} & (\text{+stream_or_alias}, \text{?in_character_code}) \\
\text{peek_code} & (\text{?in_character_code})
\end{align*}
\]

Description

\text{peek_char(SorA, Char)} succeeds if \text{Char} unifies with the next character that will be read from the stream associated with the stream-term or alias \text{SorA}. The character is not read.

\text{peek_code/2} is similar to \text{peek_char/2} but deals with character codes.

\text{peek_char/1} and \text{peek_code/1} apply to the current input stream.

Errors

\begin{table}
\begin{center}
\begin{tabular}{|l|l|}
\hline
\text{SorA is a variable} & \text{instantiation_error} \\
\text{Char is neither a variable nor an in-character} & \text{type_error(in_character, Char)} \\
\text{Code is neither a variable nor an integer} & \text{type_error(integer, Code)} \\
\text{SorA is neither a variable nor a stream-term or alias} & \text{domain_error(stream_or_alias, SorA)} \\
\text{SorA is not associated with an open stream} & \text{existence_error(stream, SorA)} \\
\text{SorA is an output stream} & \text{permission_error(input, stream, SorA)} \\
\text{SorA is associated with a binary stream} & \text{permission_error(input, binary_stream, SorA)} \\
\text{SorA has stream properties} & \text{permission_error(input, past_end_of_stream, SorA)} \\
\text{end_of_stream(past) and eof_action(error)} & \text{representation_error(character)} \\
\text{The entity input from the stream is not a character} & \text{representation_error(in_character_code)} \\
\text{Code is an integer but not an in-character code} & \text{representation_error(in_character_code)} \\
\hline
\end{tabular}
\end{center}
\end{table}

Portability

ISO predicates.

6.12.4 unget_char/2, unget_char/1, unget_code/2, unget_code/1

Templates

\[
\begin{align*}
\text{unget_char} & (\text{+stream_or_alias}, \text{+character}) \\
\text{unget_char} & (\text{+character}) \\
\text{unget_code} & (\text{+stream_or_alias}, \text{+character_code}) \\
\text{unget_code} & (\text{+character_code})
\end{align*}
\]

Description

\text{unget_char(SorA, Char)} pushes back \text{Char} onto the stream associated with the stream-term or alias \text{SorA}. \text{Char} will be the next character read by \text{get_char/2}. The maximum number of characters that can be cumulatively pushed back is given by the \text{max_unget} Prolog flag (section 6.22.1, page 117).

\text{unget_code/2} is similar to \text{unget_char/2} but deals with character codes.
unget_char/1 and unget_code/1 apply to the current input stream.

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SorA is a variable</code></td>
<td>instantiation_error</td>
</tr>
<tr>
<td><code>Char is a variable</code></td>
<td>instantiation_error</td>
</tr>
<tr>
<td><code>Code is a variable</code></td>
<td>instantiation_error</td>
</tr>
<tr>
<td><code>Char is neither a variable nor a character</code></td>
<td>type_error(character, Char)</td>
</tr>
<tr>
<td><code>Code is neither a variable nor an integer</code></td>
<td>type_error(integer, Code)</td>
</tr>
<tr>
<td><code>SorA is neither a variable nor a stream-term or alias</code></td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td><code>SorA is not associated with an open stream</code></td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td><code>SorA is an output stream</code></td>
<td>permission_error(input, stream, SorA)</td>
</tr>
<tr>
<td><code>SorA is associated with a binary stream</code></td>
<td>permission_error(input, binary_stream, SorA)</td>
</tr>
<tr>
<td><code>Code is an integer but not a character code</code></td>
<td>representation_error(character, code)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.12.5 put_char/2, put_char/1, put_code/1, put_code/2, nl/1, nl/0

Templates

```
put_char(+stream_or_alias, +character)
put_char(+character)
put_code(+stream_or_alias, +character_code)
put_code(+character_code)
nl(+stream_or_alias)
nl
```

Description

`put_char(SorA, Char)` writes `Char` onto the stream associated with the stream-term or alias `SorA`.

`put_code/2` is similar to `put_char/2` but deals with character codes.

`nl(SorA)` writes a new-line character onto the stream associated with the stream-term or alias `SorA`. This is equivalent to `put_char(SorA, \"\n\")`.

`put_char/1, put_code/1` and `nl/0` apply to the current output stream.

Errors
$\text{Sor}A$ is a variable | instantiation_error  
Char is a variable | instantiation_error  
Code is a variable | instantiation_error  
Char is neither a variable nor a character | type_error(character, Char)  
Code is neither a variable nor an integer | type_error(integer, Code)  
$\text{Sor}A$ is neither a variable nor a stream-term or alias | domain_error(stream_or_alias, $\text{Sor}A$)  
$\text{Sor}A$ is not associated with an open stream | existence_error(stream, $\text{Sor}A$)  
$\text{Sor}A$ is an input stream | permission_error(output, stream, $\text{Sor}A$)  
$\text{Sor}A$ is associated with a binary stream | permission_error(output, binary_stream, $\text{Sor}A$)  
Code is an integer but not a character code | representation_error(character_code)

**Portability**

ISO predicates.

### 6.13 Byte input/output

These built-in predicates enable a single byte to be input from and output to a binary stream. -1 is returned to indicate the end-of-file.

#### 6.13.1 get_byte/2, get_byte/1

**Templates**

\[
\text{get_byte}(+\text{stream_or_alias}, \text{?in_byte})
\]

\[
\text{get_byte}(?\text{in_byte})
\]

**Description**

$\text{get_byte(Sor}A, \text{ Byte})$ succeeds if $\text{Byte}$ unifies with the next byte read from the stream associated with the stream-term or alias $\text{Sor}A$.

$\text{get_byte/1}$ applies to the current input stream.

**Errors**

| $\text{Sor}A$ is a variable | instantiation_error  
| $\text{Sor}A$ is neither a variable nor an in-byte | type_error(in_byte, Byte)  
| $\text{Sor}A$ is neither a variable nor a stream-term or alias | domain_error(stream_or_alias, $\text{Sor}A$)  
| $\text{Sor}A$ is not associated with an open stream | existence_error(stream, $\text{Sor}A$)  
| $\text{Sor}A$ is an output stream | permission_error(input, stream, $\text{Sor}A$)  
| $\text{Sor}A$ is associated with a text stream | permission_error(input, text_stream, $\text{Sor}A$)  
| $\text{Sor}A$ has stream properties | permission_error(input, past_end_of_stream, $\text{Sor}A$)  

**Portability**

ISO predicates.
6.13.2 peak_byte/2, peak_byte/1

Templates

peak_byte(+stream_or_alias, ?in_byte)
peak_byte(?in_byte)

Description

peak_byte(SorA, Byte) succeeds if Byte unifies with the next byte that will be read from the stream associated with the stream-term or alias SorA. The byte is not read.

peak_byte/1 applies to the current input stream.

Errors

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte is neither a variable nor an in-byte</td>
<td>type_error(in_byte, Byte)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(input, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a text stream</td>
<td>permission_error(input, text_stream, SorA)</td>
</tr>
<tr>
<td>SorA has stream properties</td>
<td>permission_error(input, past_end_of_stream, SorA)</td>
</tr>
<tr>
<td>end_of_stream(past) and eof_action(error)</td>
<td>permission_error(input, past_end_of_stream, SorA)</td>
</tr>
</tbody>
</table>

Portability

ISO predicates.

6.13.3 unget_byte/2, unget_byte/1

Templates

unget_byte(+stream_or_alias, +byte)
unget_byte(+byte)

Description

unget_byte(SorA, Byte) pushes back Byte onto the stream associated with the stream-term or alias SorA. Byte will be the next byte read by get_byte/2. The maximum number of bytes that can be successively pushed back is given by the max_unget Prolog flag (section 6.22.1, page 117).

unget_byte/1 applies to the current input stream.

Errors
<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Byte is neither a variable nor a byte</td>
<td>type_error(byte, Byte)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(input, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a text stream</td>
<td>permission_error(input, text_stream, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

### 6.13.4 put_byte/2, put_byte/1

**Templates**

\[
\text{put\_byte}(+\text{stream}\_or\_alias, +\text{byte}) \\
\text{put\_byte}(+\text{byte})
\]

**Description**

\text{put\_byte}(\text{SorA}, \text{Byte}) \text{ writes } \text{Byte} \text{ onto the stream associated with the stream-term or alias } \text{SorA}.

\text{put\_byte/1 \text{ applies to the current output stream.}}

**Errors**

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Byte is neither a variable nor a byte</td>
<td>type_error(byte, Byte)</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(output, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a text stream</td>
<td>permission_error(output, text_stream, SorA)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

### 6.14 Term input/output

These built-in predicates enable a Prolog term to be input from or output to a text stream. The atom \texttt{end\_of\_file} is returned as term to indicate the end-of-file. The syntax of such terms can also be altered by changing the operators (section 6.14.10, page 87), and making some characters equivalent to others (section 6.14.12, page 89) if the \texttt{char\_conversion} Prolog flag is \texttt{on} (section 6.22.1, page 117). Double quoted tokens will be returned as an atom or a character list or a character code list depending on the value of the \texttt{double\_quotes} Prolog flag (section 6.22.1, page 117).
6.14 Term input/output

6.14.1 read_term/3, read_term/2, read/2, read/1

Templates

read_term(+stream_or_alias, ?term, +read_option_list)
read_term(?term, +read_option_list)
read(+stream_or_alias, ?term)
read(?term)

Description

read_term(SorA, Term, Options) is true if Term unifies with the next term read from the stream associated with the stream-term or alias SorA according to the options given by Options.

Read options: Options is a list of read options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- **variables(VL):** VL is unified with the list of all variables of the input term, in left-to-right traversal order. Anonymous variables are included in the list VL.

- **variable_names(VNL):** VNL is unified with the list of pairs Name = Var where Var is a named variable of the term and Name is the atom associated to the name of Var. Anonymous variables are not included in the list VNL.

- **singletons(SL):** SL is unified with the list of pairs Name = Var where Var is a named variable which occurs only once in the term and Name is the atom associated to the name of Var. Anonymous variables are not included in the list SL.

- **syntax_error(error/warning/fail):** specifies the effect of a syntax error:
  - error: a syntax_error is raised.
  - warning: a warning message is displayed and the predicate fails.
  - fail: the predicate quietly fails.
  The default value is the value of the syntax_error Prolog flag (section 6.22.1, page 117).

- **end_of_term(dot/eof):** specifies the end-of-term delimiter: dot is the classical full-stop delimiter (a dot followed with a layout character), eof is the end-of-file delimiter. This option is useful for predicates like read_from_atom/2 (section 6.15.1, page 91) to avoid to add a terminal dot at the end of the atom. The default value is dot.

read(SorA, Term) is equivalent to read_term(SorA, Term, []).

read_term/2 and read/1 apply to the current input stream.

Errors
### Portability

ISO predicates. The ISO reference raises a `representation_error(Flag)` where Flag is `max arity`, `max integer`, or `min integer` when the read term breaches an implementation defined limit specified by Flag. GNU Prolog detects neither `min integer` nor `max integer` violation and treats a `max arity` violation as a syntax error. The read options `syntax_error/1` and `end_of_term/1` are GNU Prolog extensions.

### 6.14.2  read_atom/2, read_atom/1, read_integer/2, read_integer/1, read_number/2, read_number/1

**Templates**

```prolog
read_atom(+stream_or_alias, ?atom)
read_atom(?atom)
read_integer(+stream_or_alias, ?integer)
read_integer(?integer)
read_number(+stream_or_alias, ?number)
read_number(?number)
```

**Description**

`read_atom(SorA, Atom)` succeeds if `Atom` unifies with the next atom read from the stream associated with the stream-term or alias `SorA`.

`read_integer(SorA, Integer)` succeeds if `Integer` unifies with the next integer read from the stream associated with the stream-term or alias `SorA`.

`read_number(SorA, Number)` succeeds if `Number` unifies with the next number (integer or floating point number) read from the stream associated with the stream-term or alias `SorA`.

`read_atom/1, read_integer/1` and `read_number/1` apply to the current input stream.

**Errors**

<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options is a partial list or a list with an element E which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>Options is neither a partial list nor a list</td>
<td>type_error(list, Options)</td>
</tr>
<tr>
<td>an element E of the Options list is neither a variable nor a valid read option</td>
<td>domain_error(read_option, E)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(input, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a binary stream</td>
<td>permission_error(input, binary_stream, SorA)</td>
</tr>
<tr>
<td>SorA has stream properties end_of_stream(past) and eof_action(error)</td>
<td>permission_error(input, past_end_of_stream, SorA)</td>
</tr>
<tr>
<td>a syntax error occurs and the value of the syntax_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>syntax_error(atom explaining the error)</td>
</tr>
</tbody>
</table>
Portability

GNU Prolog predicates.

6.14.3 read_token/2, read_token/1

Templates

read_token(+stream_or_alias, ?nonvar)
read_token(?nonvar)

Description

read_token(SorA, Token) succeeds if Token unifies with the encoding of the next Prolog token read from the stream associated with stream-term or alias SorA.

Token encoding:

- var(A): a variable is read whose name is the atom A.
- an atom A: an atom A is read.
- integer N: an integer N is read.
- floating point number N: a floating point number N is read.
- string(A): a string (double quoted item) is read whose characters forms the atom A.
- punct(P): a punctuation character P is read (P is a one-character atom in ()[]{}), the atom full_stop or the atom end_of_file).
- back_quotes(A): a back quoted item is read whose characters forms the atom A.
- extended(A): an extended character A (an atom) is read.

As for read_term/3, the behavior of read_token/2 can be affected by some Prolog flags (section 6.14, page 78).

read_token/1 applies to the current input stream.

Errors
<table>
<thead>
<tr>
<th>SorA is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an output stream</td>
<td>permission_error(input, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a binary stream</td>
<td>permission_error(input, binary_stream, SorA)</td>
</tr>
<tr>
<td>SorA has stream properties</td>
<td>permission_error(input, past_end_of_stream, SorA)</td>
</tr>
<tr>
<td>end_of_stream(past) and eofAction(error)</td>
<td>syntax_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

### 6.14.4 syntax_error_info/4

**Templates**

```
syntax_error_info(?atom, ?integer, ?integer, ?atom)
```

**Description**

`syntax_error_info(FileName, Line, Column, Error)` returns the information associated to the last syntax error. `Line` is the line number of the error, `Column` is the column number of the error and `Error` is an atom explaining the error.

**Errors**

<table>
<thead>
<tr>
<th>Error</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileName is neither a variable nor an atom</td>
<td>type_error(atom, FileName)</td>
</tr>
<tr>
<td>Line is neither a variable nor an integer</td>
<td>type_error(integer, Line)</td>
</tr>
<tr>
<td>Column is neither a variable nor an integer</td>
<td>type_error(integer, Column)</td>
</tr>
<tr>
<td>Error is neither a variable nor an atom</td>
<td>type_error(atom, Error)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.14.5 last_read_start_line_column/2

**Templates**

```
last_read_start_line_column(?integer, ?integer)
```

**Description**

`last_read_start_line_column(Line, Column)` unifies `Line` and `Column` with the line number and the column number associated to the start of the last read predicate. This predicate can be used after calling one of the following predicates: `read_term/3, read_term/2, read/2, read/1` (section 6.14.1, page 79), `read_atom/2, read_atom/1, read_integer/2, read_integer/1, read_number/2, read_number/1` (section 6.14.2, page 80) or `read_token/2, read_token/1` (section 6.14.3, page 81).
6.14  Term input/output

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line is neither a variable nor an integer</td>
<td><code>type_error(integer, Line)</code></td>
</tr>
<tr>
<td>Column is neither a variable nor an integer</td>
<td><code>type_error(integer, Column)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.14.6  write_term/3, write_term/2, write/2, write/1, writeq/2, writeq/1,
        write_canonical/2, write_canonical/1, display/2, display/1, print/2,
        print/1

Templates

```
write_term(+stream_or_alias, ?term, +write_option_list)
write_term(?term, +write_option_list)
write(+stream_or_alias, ?term)
write(?term)
writeq(+stream_or_alias, ?term)
writeq(?term)
write_canonical(+stream_or_alias, ?term)
write_canonical(?term)
display(+stream_or_alias, ?term)
display(?term)
print(+stream_or_alias, ?term)
print(?term)
```

Description

`write_term(SorA, Term, Options)` writes Term to the stream associated with the stream-term or alias SorA according to the options given by Options.

Write options: Options is a list of write options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- `quoted(true/false)`: If true each atom and functor is quoted if this would be necessary for the term to be input by `read_term/3`. If false no extra quotes are written. The default value is false.
- `ignore_ops(true/false)`: If true each compound term is output in functional notation (neither operator notation nor list notation is used). If false operator and list notations are used. The default value is false.
- `numbervars(true/false)`: If true a term of the form `$VAR'(N)`, where N is an integer, is output as a variable name (see below). If false such a term is output normally (according to the other options). The default value is true.
- `namevars(true/false)`: If true a term of the form `$VARNAME'(Name)`, where Name is an atom, is output as a variable name (see below). If false such a term is output normally (according to the other options). The default value is true.
- `space_args(true/false)`: If true an extra space character is emitted after each comma separating the arguments of a compound term in functional notation or of a list. If false no extra space is emitted. The default value is false.
- `portrayed(true/false)`: If true and if there exists a predicate `portray/1`, `write_term/3` acts as follows: if Term is a variable it is simply written. If Term is non-variable then it is passed to
portray/1. If this succeeds then it is assumed that Term has been output. Otherwise write_term/3 outputs the principal functor of Term (Term itself if it is atomic) according to other options and recursively calls portray/1 on the components of Term (if it is a compound term). With ignore_ops(false) a list is first passed to portray/1 and only if this call fails each element of the list is passed to portray/1 (thus every sub-list is not passed). The default value is false.

- max_depth(N): controls the depth of output for compound terms. N is an integer specifying the depth. The output of a term whose depth is greater then N gives rise to the output of ... (3 dots). By default there is no depth limit.

Variable numbering: when the numvars(true) option is passed to write_term/3 any term of the form 'VAR'(N) where N is an integer is output as a variable name consisting of a capital letter possibly followed by an integer. The capital letter is the (I+1)th letter of the alphabet and the integer is J, where I = N mod 26 and J = N // 26. The integer J is omitted if it is zero. For example:

- 'VAR'(0) is written as A
- 'VAR'(1) is written as B
- 'VAR'(25) is written as Z
- 'VAR'(26) is written as A1
- 'VAR'(27) is written as B1

Variable naming: when the namevars(true) option is passed to write_term/3 any term of the form 'VARNAME'(Name) where Name is an atom is output as a variable name consisting of the characters Name. For example: 'VARNAME'('A') is written as A (even in the presence of the quoted(true) option).

write(SoAR, Term) is equivalent to write_term(SoAR, Term, []).

writeq(SoAR, Term) is equivalent to write_term(SoAR, Term, [quoted(true)]).

write_canonical(SoAR, Term) is equivalent to write_term(SoAR, Term, [quoted(true), ignore_ops(true), numvars(false)]).

display(SoAR, Term) is equivalent to write_term(SoAR, Term, [ignore_ops(true), numvars(false)]).

print(SoAR, Term) is equivalent to write_term(SoAR, Term, [numvars(false), portrayed(true)]).

write_term/2, write/1, writeq/1, write_canonical/1, display/1 and print/1 apply to the current output stream.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoAR is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Options is a partial list or a list with an element E which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Options is neither a partial list nor a list</td>
<td>type_error(list, Options)</td>
</tr>
<tr>
<td>SoAR is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SoAR)</td>
</tr>
<tr>
<td>an element E of the Options list is neither a variable nor a valid write-option</td>
<td>domain_error(write_option, E)</td>
</tr>
<tr>
<td>SoAR is not associated with an open stream</td>
<td>existence_error(stream, SoAR)</td>
</tr>
<tr>
<td>SoAR is an input stream</td>
<td>permission_error(output, stream, SoAR)</td>
</tr>
<tr>
<td>SoAR is associated with a binary stream</td>
<td>permission_error(output, binary_stream, SoAR)</td>
</tr>
</tbody>
</table>

Portability
ISO predicates except `display/1-2` and `print/1-2` that are GNU Prolog predicates. `namevars/1`, `space_args/1`, `portrayed/1` and `max_depth/1` options are GNU Prolog extensions.

### 6.14.7 `format/3, format/2`

**Templates**

```prolog
format(+stream_or_alias, +character_code_list_or_atom, +list)
format(+character_code_list_or_atom, +list)
```

**Description**

`format(SorA, Format, Arguments)` writes the `Format` string replacing each format control sequence F by the corresponding element of `Arguments` (formatted according to F) to the stream associated with the stream-term or alias `SorA`.

**Format control sequences**: the general format of a control sequence is `"NC"`. The character C determines the type of the control sequence. N is an optional numeric argument. An alternative form of N is `'*'`. `'*'` implies that the next argument `Arg` in `Arguments` should be used as a numeric argument in the control sequence. The use of C `printf()` formatting sequence (beginning by the character `%`) is also allowed. The following control sequences are available:

<table>
<thead>
<tr>
<th>Format sequence</th>
<th>type of the argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>~a</code></td>
<td>atom</td>
<td>print the atom without quoting</td>
</tr>
<tr>
<td><code>~Nc</code></td>
<td>character code</td>
<td>print the character associated to the code. N is the number of times to print the character (default: 1)</td>
</tr>
<tr>
<td><code>~Nf</code></td>
<td>float expression</td>
<td>pass the argument <code>Arg</code> and <code>N</code> to the C <code>printf()</code> function as: if <code>N</code> is not specified <code>printf(&quot;%f&quot;, Arg)</code> else <code>printf(&quot;%Nf&quot;, Arg)</code>. Similarly for <code>&quot;Nc&quot;, &quot;Ne&quot;, &quot;Ng&quot;</code> and <code>&quot;Ng&quot;</code></td>
</tr>
<tr>
<td><code>~Nd</code></td>
<td>integer expression</td>
<td>print the argument. N is the number of digits after the decimal point. If <code>N</code> is 0 no decimal point is printed (default: 0)</td>
</tr>
<tr>
<td><code>~ND</code></td>
<td>integer expression</td>
<td>identical to <code>~Nd</code> except that <code>,</code> separates groups of three digits to the left of the decimal point</td>
</tr>
<tr>
<td><code>~Nr</code></td>
<td>integer expression</td>
<td>print the argument according to the radix N. 2 ≤ N ≤ 36 (default: 8). The letters a-z denote digits &gt; 9</td>
</tr>
<tr>
<td><code>~NR</code></td>
<td>integer expression</td>
<td>identical to <code>~Nr</code> except that the letters A-Z denote digits &gt; 9</td>
</tr>
<tr>
<td><code>~Ns</code></td>
<td>character code list</td>
<td>print exactly N characters (default: the length of the list)</td>
</tr>
<tr>
<td><code>~NS</code></td>
<td>character code list</td>
<td>print exactly N characters (default: the length of the list)</td>
</tr>
<tr>
<td><code>~i</code></td>
<td>term</td>
<td>ignore the current argument</td>
</tr>
<tr>
<td><code>~k</code></td>
<td>term</td>
<td>pass the argument to <code>write_canonical/1</code> (section 6.14.6, page 83)</td>
</tr>
<tr>
<td><code>~p</code></td>
<td>term</td>
<td>pass the argument to <code>print/1</code> (section 6.14.6, page 83)</td>
</tr>
<tr>
<td><code>~q</code></td>
<td>term</td>
<td>pass the argument to <code>writeq/1</code> (section 6.14.6, page 83)</td>
</tr>
<tr>
<td><code>~w</code></td>
<td>term</td>
<td>pass the argument to <code>write/1</code> (section 6.14.6, page 83)</td>
</tr>
<tr>
<td><code>~</code></td>
<td>none</td>
<td>print the character <code>,</code></td>
</tr>
<tr>
<td><code>~Nn</code></td>
<td>none</td>
<td>print N new-line characters (default: 1)</td>
</tr>
<tr>
<td><code>~N</code></td>
<td>none</td>
<td>print a new-line character if not at the beginning of a line</td>
</tr>
<tr>
<td><code>?</code></td>
<td>atom</td>
<td>use the argument as a nested format string</td>
</tr>
<tr>
<td><code>%F</code></td>
<td>atom, integer or float expression</td>
<td>interface to the C function <code>printf(3)</code> for outputting atoms (C string), integers and floating point numbers. * are also allowed.</td>
</tr>
</tbody>
</table>

`format/2` applies to the current output stream.

**Errors**
| **Arguments** is a partial list | instantiation_error |
| Format is a partial list or a list with an element E which is a variable | instantiation_error |
| **Arguments** is a partial list | instantiation_error |
| Format is neither a partial list nor a list or an atom | type_error(list, Format) |
| **Arguments** is neither a partial list nor a list | type_error(list, Arguments) |
| an element E of the Format list is neither a variable nor a character code | representation_error(character_code, E) |
| SorA is neither a variable nor a stream-term or alias | domain_error(stream_or_alias, SorA) |
| an element E of Format is not a valid format control sequence | domain_error(format_control_sequence, E) |
| the **Arguments** list does not contain sufficient elements | domain_error(non_empty_list, []) |
| an element E of the **Arguments** list is a variable while a non-variable term was expected | instantiation_error |
| an element E of the **Arguments** list is neither variable nor an atom while an atom was expected | type_error(atom, E) |
| an element E of the **Arguments** cannot be evaluated as an arithmetic expression while an integer or a floating point number was expected | an arithmetic error (section 6.6.1, page 45) |
| an element E of the **Arguments** list is neither variable nor character code while a character code was expected | representation_error(character_code, E) |
| SorA is not associated with an open stream | existence_error(stream, SorA) |
| SorA is an input stream | permission_error(input, stream, SorA) |
| SorA is associated with a binary stream | permission_error(output, binary_stream, SorA) |

**Portability**

GNU Prolog predicates.

### 6.14.8 `portray_clause/2, portray_clause/1`

**Templates**

```
portray_clause(+stream_or_alias, +clause)
portray_clause(+clause)
```

**Description**

The `portray_clause/2, portray_clause/1` predicate pretty prints `Clause` to the stream associated with the stream-term or alias `SorA`. `portray_clause/2` uses the variable binding predicates `name_singleton_vars/1` (section 6.5.1, page 43) and `numbervars/1` (section 6.5.3, page 44). This predicate is used by `listing/1` (section 6.23.3, page 122).

`portray_clause/1` applies to the current output stream.

**Errors**
6.14  Term input/output

<table>
<thead>
<tr>
<th>Clause is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clause is neither a variable nor a callable term</td>
<td>type_error(callable, Clause)</td>
</tr>
<tr>
<td>SorA is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>SorA is neither a variable nor a stream-term or alias</td>
<td>domain_error(stream_or_alias, SorA)</td>
</tr>
<tr>
<td>SorA is not associated with an open stream</td>
<td>existence_error(stream, SorA)</td>
</tr>
<tr>
<td>SorA is an input stream</td>
<td>permission_error(output, stream, SorA)</td>
</tr>
<tr>
<td>SorA is associated with a binary stream</td>
<td>permission_error(output, binary_stream, SorA)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.14.9  get_print_stream/1

Templates

get_print_stream(-stream)

Description

get_print_stream(Stream) unifies Stream with the stream-term associated to the output stream used by print/2 (section 6.14.6, page 83). The purpose of this predicate is to allow a user-defined portray/1 predicate to identify the output stream in use.

Errors

| Stream is not a variable | type_error(variable, Stream) |

Portability

GNU Prolog predicate.

6.14.10  op/3

Templates

op(+integer, +operator_specifier, +atom_or_atom_list)

Description

op(Priority, OpSpecifier, Operator) alters the operator table. Operator is declared as an operator with properties defined by specifier OpSpecifier and Priority. Priority must be an integer ≥ 0 and ≤ 1200. If Priority is 0 then the operator properties of Operator (if any) are cancelled. Operator may also be a list of atoms in which case all of them are declared to be operators. In general, operators can be removed from the operator table and their priority or specifier can be changed. However, it is an error to attempt to change the ’,’ operator from its initial status. An atom can have multiple operator definitions (e.g. prefix and infix like +) however an atom cannot have both an infix and a postfix operator definitions.

Operator specifiers: the following specifiers are available:
<table>
<thead>
<tr>
<th>Specifier</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>fx</td>
<td>prefix</td>
<td>no</td>
</tr>
<tr>
<td>fy</td>
<td>prefix</td>
<td>yes</td>
</tr>
<tr>
<td>xf</td>
<td>postfix</td>
<td>no</td>
</tr>
<tr>
<td>yf</td>
<td>postfix</td>
<td>yes</td>
</tr>
<tr>
<td>xfx</td>
<td>infix</td>
<td>no</td>
</tr>
<tr>
<td>yfx</td>
<td>infix</td>
<td>left</td>
</tr>
<tr>
<td>xfy</td>
<td>infix</td>
<td>right</td>
</tr>
</tbody>
</table>

**Prolog predefined operators:**

<table>
<thead>
<tr>
<th>Priority</th>
<th>Specifier</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>xfx</td>
<td>:- --&gt;</td>
</tr>
<tr>
<td>1200</td>
<td>fx</td>
<td>:-</td>
</tr>
<tr>
<td>1100</td>
<td>xfy</td>
<td>;</td>
</tr>
<tr>
<td>1050</td>
<td>xfy</td>
<td>-&gt;</td>
</tr>
<tr>
<td>1000</td>
<td>xfy</td>
<td>,</td>
</tr>
<tr>
<td>900</td>
<td>fy</td>
<td>+</td>
</tr>
<tr>
<td>700</td>
<td>xfx</td>
<td>= = . . . == @ @&lt; @&gt; @&gt; is =:= == &lt; =&lt; &gt;&gt;=</td>
</tr>
<tr>
<td>500</td>
<td>yfx</td>
<td>+ - /\ /\</td>
</tr>
<tr>
<td>400</td>
<td>yfx</td>
<td>* / // rem mod &lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>200</td>
<td>xfy</td>
<td>** ^</td>
</tr>
<tr>
<td>200</td>
<td>fy</td>
<td>+ - \</td>
</tr>
</tbody>
</table>

**FD predefined operators:**

<table>
<thead>
<tr>
<th>Priority</th>
<th>Specifier</th>
<th>Operators</th>
</tr>
</thead>
</table>
| 750      | xfy       | #<=> #
<=> |
| 740      | xfy       | #==> #\==> |
| 730      | xfy       | # #\ #\# |
| 720      | yfx       | #\ #\# |
| 710      | fy        | #\ #\# |
| 700      | xfx       | #= #\= #< #=# #>= #=# #\= #< #=# #>= #=# |
| 500      | yfx       | + -      |
| 400      | yfx       | * / // rem |
| 200      | xfy       | **      |
| 200      | fy        | + -      |

**Errors**
Priority is a variable | instantiation_error
OpSpecifier is a variable | instantiation_error
Operator is a partial list or a list with an element E which is a variable | instantiation_error
Priority is neither a variable nor an integer | type_error(integer, Priority)
OpSpecifier is neither a variable nor an atom | type_error(atom, OpSpecifier)
Operator is neither a partial list nor a list nor an atom | type_error(list, Operator)
an element E of the Operator list is neither a variable nor an atom | type_error(atom, E)
Priority is an integer not \( \geq 0 \) and \( \leq 1200 \) | domain_error(operator_priority, Priority)
OpSpecifier is not a valid operator specifier | domain_error(operator_specifier, OpSpecifier)
Operator is ",," or an element of the Operator list is ",," | permission_error(modify, operator, ",," )
OpSpecifier is a specifier such that Operator would have a postfix and an infix definition. | permission_error(create, operator, Operator)

Portability

ISO predicate.

The ISO reference implies that if a program calls \texttt{current\_op/3}, then modifies an operator definition by calling \texttt{op/3} and backtracks into the call to \texttt{current\_op/3}, then the changes are guaranteed not to affect that \texttt{current\_op/3} goal. This is not guaranteed by GNU Prolog.

6.14.11 current\_op/3

Templates

\texttt{current\_op(?integer, ?operator\_specifier, ?atom)}

Description

current\_op(Priority, OpSpecifier, Operator) succeeds if Operator is an operator with properties defined by specifier OpSpecifier and Priority. This predicate is re-executable on backtracking.

Errors

Priority is neither a variable nor an operator priority | domain_error(operator_priority, Priority)
OpSpecifier is neither a variable nor an operator specifier | domain_error(operator_specifier, OpSpecifier)
Operator is neither a variable nor an atom | type_error(atom, Operator)

Portability

ISO predicate.

6.14.12 char\_conversion/2

Templates

\texttt{char\_conversion(+character, +character)}
Description

char_conversion(InChar, OutChar) alters the character- conversion mapping. This mapping is used by the following read predicates: read_term/3 (section 6.14.1, page 79), read_atom/2, read_integer/2, read_number/2 (section 6.14.2, page 80) and read_token/2 (section 6.14.3, page 81) to replace any occurrence of a character InChar by OutChar. However the conversion mechanism should have been previously activated by switching on the char_conversion Prolog flag (section 6.22.1, page 117). When InChar and OutChar are the same, the effect is to remove any conversion of a character InChar.

Note that the single character read predicates (e.g. get_char/2) never do character conversion. If such behavior is required, it must be explicitly done using current_char_conversion/2 (section 6.14.13, page 90).

Errors

<table>
<thead>
<tr>
<th>InChar is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutChar is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>InChar is neither a variable nor a character</td>
<td>type_error(character, InChar)</td>
</tr>
<tr>
<td>OutChar is neither a variable nor a character</td>
<td>type_error(character, OutChar)</td>
</tr>
</tbody>
</table>

Portability

ISO predicate. The type_error(character,...) is a GNU Prolog behavior, the ISO reference instead defines a representation_error(character) in this case. This seems to be an error of the ISO reference since, for many other built-in predicates accepting a character (e.g. char_code/2, put_char/2), a type_error is raised.

The ISO reference implies that if a program calls current_char_conversion/2, then modifies the character mapping by calling char_conversion/2, and backtracks into the call to current_char_conversion/2 then the changes are guaranteed not to affect that current_char_conversion/2 goal. This is not guaranteed by GNU Prolog.

6.14.13 current_char_conversion/2

Templates

current_char_conversion(?character, ?character)

Description

current_char_conversion(InChar, OutChar) succeeds if the conversion of InChar is OutChar according to the character-conversion mapping. In that case, InChar and OutChar are different. This predicate is re-executable on backtracking.

Errors

<table>
<thead>
<tr>
<th>InChar is neither a variable nor a character</th>
<th>type_error(character, InChar)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutChar is neither a variable nor a character</td>
<td>type_error(character, OutChar)</td>
</tr>
</tbody>
</table>

Portability

6.15 Input/output from/to constant terms

These built-in predicates enable a Prolog term to be input from or output to a Prolog constant term (atom, character list or character code list). All these predicates can be defined using constant term streams (section 6.11, page 69). They are however simpler to use.

6.15.1  \texttt{read_term_from_atom/3, read_from_atom/2, read_token_from_atom/2}

Templates

\begin{verbatim}
read_term_from_atom(+atom ?term, +read_option_list)
read_from_atom(+atom, ?term)
read_token_from_atom(+atom, ?nonvar)
\end{verbatim}

Description

Like \texttt{read_term/3, read/2} (section 6.14.1, page 79) and \texttt{read_token/2} (section 6.14.3, page 81) except that characters are not read from a text-stream but from \texttt{Atom}; the atom given as first argument.

Errors

\begin{itemize}
\item \texttt{Atom} is a variable \hspace{1cm} \texttt{instantiation_error}
\item \texttt{Atom} is neither a variable nor an atom \hspace{1cm} \texttt{type_error(Atom, Atom)}
\item see associated predicate errors \hspace{1cm} (section 6.14.1, page 79) and (section 6.14.3, page 81)
\end{itemize}

Portability

GNU Prolog predicates.

6.15.2  \texttt{read_term_from_chars/3, read_from_chars/2, read_token_from_chars/2}

Templates

\begin{verbatim}
read_term_from_chars(+character_list ?term, +read_option_list)
read_from_chars(+character_list, ?term)
read_token_from_chars(+character_list, ?nonvar)
\end{verbatim}

Description

Like \texttt{read_term/3, read/2} (section 6.14.1, page 79) and \texttt{read_token/2} (section 6.14.3, page 81) except that characters are not read from a text-stream but from \texttt{Chars}; the character list given as first argument.

Errors

\begin{itemize}
\item \texttt{Chars} is a partial list or a list with an element \texttt{E} which is a variable \hspace{1cm} \texttt{instantiation_error}
\item \texttt{Chars} is neither a partial list nor a list \hspace{1cm} \texttt{type_error(list, Chars)}
\item an element \texttt{E} of the \texttt{Chars} list is neither a variable nor a character \hspace{1cm} \texttt{type_error(character, E)}
\item see associated predicate errors \hspace{1cm} (section 6.14.1, page 79) and (section 6.14.3, page 81)
\end{itemize}

Portability
GNU Prolog predicates.

6.15.3 read_term_from_codes/3, read_from_codes/2, read_token_from_codes/2

Templates

\[
\text{read_term_from_codes}(\text{+character_code_list}, \text{?term}, \text{+read_option_list})
\]
\[
\text{read_from_codes}(\text{+character_code_list}, \text{?term})
\]
\[
\text{read_token_from_codes}(\text{+character_code_list}, \text{?nonvar})
\]

Description

Like \text{read_term}/3, \text{read}/2 (section 6.14.1, page 79) and \text{read_token}/2 (section 6.14.3, page 81) except that characters are not read from a text-stream but from Codes; the character code list given as first argument.

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codes is a partial list or a list with an element E which is a variable</td>
</tr>
<tr>
<td>Codes is neither a partial list nor a list</td>
</tr>
<tr>
<td>an element E of the Codes list is neither a variable nor an integer</td>
</tr>
<tr>
<td>an element E of the Codes list is an integer but not a character code</td>
</tr>
<tr>
<td>see associated predicate errors</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.15.4 write_term_to_atom/3, write_to_atom/2, writeq_to_atom/2,
\[
\text{write_canonical_to_atom}/2, \text{display_to_atom}/2, \text{print_to_atom}/2,
\]
\[
\text{format_to_atom}/3
\]

Templates

\[
\text{write_term_to_atom}(\text{?atom}, \text{?term}, \text{+write_option_list})
\]
\[
\text{write_to_atom}(\text{?atom}, \text{?term})
\]
\[
\text{writeq_to_atom}(\text{?atom}, \text{?term})
\]
\[
\text{write_canonical_to_atom}(\text{?atom}, \text{?term})
\]
\[
\text{display_to_atom}(\text{?atom}, \text{?term})
\]
\[
\text{print_to_atom}(\text{?atom}, \text{?term})
\]
\[
\text{format_to_atom}(\text{?atom}, \text{+character_code_list_or_atom}, \text{+list})
\]

Description

Similar to \text{write_term}/3, \text{write}/2, \text{writeq}/2, \text{write_canonical}/2, \text{display}/2, \text{print}/2 (section 6.14.6, page 83) and \text{format}/3 (section 6.14.7, page 85) except that characters are not written onto a text-stream but are collected as an atom which is then unified with the first argument \text{Atom}.

Errors
<table>
<thead>
<tr>
<th><strong>Atom</strong> is neither a variable nor an atom</th>
<th><strong>type_error(Atom, Atom)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>see associated predicate errors</td>
<td>(section 6.14.6, page 83) and (section 6.14.7, page 85)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

6.15.5  write_term_to_chars/3, write_to_chars/2, writeq_to_chars/2, write_canonical_to_chars/2, display_to_chars/2, print_to_chars/2, format_to_chars/3

**Templates**

<table>
<thead>
<tr>
<th>template</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_term_to_chars(?character_list, ?term, +write_option_list)</td>
<td></td>
</tr>
<tr>
<td>write_to_chars(?character_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>writeq_to_chars(?character_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>write_canonical_to_chars(?character_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>display_to_chars(?character_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>print_to_chars(?character_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>format_to_chars(?character_list, +character_code_list_or_atom, +list)</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Similar to write_term/3, write/2, writeq/2, write_canonical/2, display/2, print/2 (section 6.14.6, page 83) and format/3 (section 6.14.7, page 85) except that characters are not written onto a text-stream but are collected as a character list which is then unified with the first argument *Chars*.

**Errors**

<table>
<thead>
<tr>
<th><strong>Chars</strong> is neither a partial list nor a list</th>
<th><strong>type_error(Chars, List)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>see associated predicate errors</td>
<td>(section 6.14.6, page 83) and (section 6.14.7, page 85)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

6.15.6  write_term_to_codes/3, write_to_codes/2, writeq_to_codes/2, write_canonical_to_codes/2, display_to_codes/2, print_to_codes/2, format_to_codes/3

**Templates**

<table>
<thead>
<tr>
<th>template</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_term_to_codes(?character_code_list, ?term, +write_option_list)</td>
<td></td>
</tr>
<tr>
<td>write_to_codes(?character_code_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>writeq_to_codes(?character_code_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>write_canonical_to_codes(?character_code_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>display_to_codes(?character_code_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>print_to_codes(?character_code_list, ?term)</td>
<td></td>
</tr>
<tr>
<td>format_to_codes(?character_code_list, +character_code_list_or_atom, +list)</td>
<td></td>
</tr>
</tbody>
</table>

**Description**
Similar to write_term/3, write/2, writeq/2, write_canonical/2, display/2, print/2 (section 6.14.6, page 83) and format/3 (section 6.14.7, page 85) except that characters are not written onto a text-stream but are collected as a character code list which is then unified with the first argument Codes.

**Errors**

<table>
<thead>
<tr>
<th>Codes is neither a partial list nor a list</th>
<th>type_error(list, Codes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>see associated predicate errors</td>
<td>(section 6.14.6, page 83) and (section 6.14.7, page 85)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

### 6.16 DEC-10 compatibility input/output

#### 6.16.1 Introduction

The DEC-10 Prolog I/O predicates manipulate streams implicitly since they only refer to current input/output streams (section 6.10.1, page 55). The current input and output streams are initially set to user_input and user_output respectively. The predicate see/1 (resp. tell/1, append/1) can be used for setting the current input (resp. output) stream to newly opened streams for particular files. The predicate seen/0 (resp. told/0) close the current input (resp. output) stream, and resets it to the standard input (resp. output). The predicate seeing/1 (resp. telling/1) is used for retrieving the file name associated with the current input (resp. output) stream. The file name user stands for the standard input or output, depending on context (user_input and user_output can also be used). The DEC-10 Prolog I/O predicates are only provided for compatibility, they are now obsolete and their use is discouraged. The predicates for explicit stream manipulation should be used instead (section 6.10, page 55).

#### 6.16.2 see/1, tell/1, append/1

**Templates**

see(+source,sink)
tell(+source,sink)
append(+source,sink)

**Description**

see(FileName) sets the current input stream to FileName. If there is a stream opened by see/1 associated with the same FileName already, then it becomes the current input stream. Otherwise, FileName is opened for reading and becomes the current input stream.

tell(FileName) sets the current output stream to FileName. If there is a stream opened by tell/1 associated with the same FileName already, then it becomes the current output stream. Otherwise, FileName is opened for writing and becomes the current output stream.

append(FileName) like tell/1 but FileName is opened for writing + append.
A stream-term (obtained with any other built-in predicate) can also be provided as FileName to these predicates.

**Errors**

See errors associated to open/4 (section 6.10.6, page 58).

**Portability**

GNU Prolog predicates.

### 6.16.3 seeing/1, telling/1

**Templates**

seeing(?source_sink)

telling(?source_sink)

**Description**

seeing(FileName) succeeds if FileName unifies with the name of the current input file, if it was opened by see/1; else with the current input stream-term, if this is not user_input, otherwise with user.

telling(FileName) succeeds if FileName unifies with the name of the current output file, if it was opened by tell/1 or append/1; else with the current output stream-term, if this is not user_output, otherwise with user.

**Errors**

None.

**Portability**

GNU Prolog predicates.

### 6.16.4 seen/0, told/0

**Templates**

seen

told

**Description**

seen closes the current input, and resets it to user_input.

told closes the current output, and resets it to user_output.

**Errors**

None.

**Portability**

GNU Prolog predicates.
6.16.5 get0/1, get/1, skip/1

Templates

get0(?in_character_code)
get(?in_character_code)
skip(+character_code)

Description

get0(Code) succeeds if Code unifies with the next character code read from the current input stream. Thus it is equivalent to get_code(Code) (section 6.12.1, page 72).

get(Code) succeeds if Code unifies with the next character code read from the current input stream that is not a layout character.

skip(Code) skips just past the next character code Code from the current input stream.

Errors

See errors for get_code/2 (section 6.12.1, page 72).

Portability

 GNU Prolog predicates.

6.16.6 put/1, tab/1

Templates

put(+character_code)
tab(+evaluable)

Description

put(Code) writes the character whose code is Code onto the current output stream. It is equivalent to put_code(Code) (section 6.12.5, page 75).

tab(N) writes N spaces onto the current output stream. N may be an arithmetic expression.

Errors

See errors for put_code/2 (section 6.12.5, page 75) and for arithmetic expressions (section 6.6.1, page 45).

Portability

 GNU Prolog predicates.

6.17 Term expansion

6.17.1 Definite clause grammars

Definite clause grammars are a useful notation to express grammar rules. However the ISO reference does not include them, so they should be considered as a system dependent feature. Definite clause grammars
are an extension of context-free grammars. A grammar rule is of the form:

\[
\text{head} \rightarrow \text{body}.
\]

\(\rightarrow\) is a predefined infix operator (section 6.14.10, page 87).

Here are some features of definite clause grammars:

- a non-terminal symbol may be any callable term.
- a terminal symbol may be any Prolog term and is written as a list. The empty list represents an empty sequence of terminals.
- a sequence is expressed using the Prolog conjunction operator \( (',')/2 \).
- the head of a grammar rule consists of a non-terminal optionally followed by a sequence of terminals (i.e. a Prolog list).
- the body of a grammar rule consists of a sequence of non-terminals, terminals, predicate call, disjunction (using ;/2), if-then (using ->/2) or cut (using !).
- a predicate call must be enclosed in curly brackets (using \{ /1 ). This makes it possible to express an extra condition.

A grammar rule is nothing but a “syntactic sugar” for a Prolog clause. Each grammar rule accepts as input a list of terminals (tokens), parses a prefix of this list and gives as output the rest of this list (possibly enlarged). This rest is generally parsed later. So, each a grammar rule is translated into a Prolog clause that explicitly the manages the list. Two arguments are then added: the input list \( \text{Start} \) and the output list \( \text{Stop} \). For instance:

\[
p \rightarrow q.
\]

is translated into:

\[
p(\text{Start}, \text{End}) :- q(\text{Start}, \text{End}).
\]

Extra arguments can be provided and the body of the rule can contain several non-terminals. Example:

\[
p(X, Y) \rightarrow q(X),
\]

\[
r(X, Y),
\]

\[
s(Y).
\]

is translated into:

\[
p(X, Y, \text{Start}, \text{End}) :- q(X, \text{Start}, A),
\]

\[
r(X, Y, A, B),
\]

\[
s(Y, B, \text{End}).
\]

Terminals are translated using unification:

\[
\text{assign}(X,Y) \rightarrow \text{left}(X), [::], \text{right}(Y), [;].
\]

is translated into:

\[
\text{assign}(X,Y,\text{Start},\text{End}) :-
\]

\[
\text{left}(X, \text{Start}, A),
\]

\[
A=[::|[B],
\]

\[
\text{right}(Y, B, C),
\]

\[
C=[;|\text{End}].
\]
Terminals appearing on the left-hand side of a rule are connected to the output argument of the head.

It is possible to include a call to a prolog predicate enclosing it in curly brackets (to distinguish them from non-terminals):

\[
\text{assign}(X,Y) \rightarrow \text{left}(X), [::], \text{right}(Y0), \{ Y \text{ is } Y0 \}, [;].
\]

is translated into:

\[
\text{assign}(X,Y, \text{Start}, \text{End}) :- \\
\text{left}(X, \text{Start}, A), \\
A=[::|B], \\
\text{right}(Y0, B, C), \\
Y \text{ is } Y0, \\
C=[;|\text{End}].
\]

Cut, disjunction and if-then(-else) are translated literally (and do not need to be enclosed in curly brackets).

### 6.17.2 expand_term/2, term_expansion/2

**Templates**

\[
\text{expand}_\text{term}(\text{?term}, \text{?term}) \\
\text{term}_\text{expansion}(\text{?term}, \text{?term})
\]

**Description**

\text{expand}_\text{term}(\text{Term1}, \text{Term2}) \text{ succeeds if } \text{Term2 is a transformation of } \text{Term1}. \text{ The transformation steps are as follows:}

- if \text{Term1} is a variable, it is unified with \text{Term2}
- if \text{term_expansion}(\text{Term1}, \text{Term2}) \text{ succeeds } \text{Term2 is assumed to be the transformation of } \text{Term1}.
- if \text{Term1} is a DCG then \text{Term2 is its translation (section 6.17.1, page 96)}.
- otherwise \text{Term2 is unified with Term1}.

\text{term_expansion}(\text{Term1}, \text{Term2}) \text{ is a hook predicate allowing the user to define a specific transformation}.

The GNU Prolog compiler (section 2.4, page 10) automatically calls \text{expand_term/2} on each \text{Term1} read in. \text{ However, in the current release, only DCG transformation are done by the compiler (i.e. term_expansion/2 cannot be used). To use term_expansion/2, it is necessary to call expand_term/2 explicitly}.

**Errors**

None.

**Portability**

GNU Prolog predicate.

### 6.17.3 phrase/3, phrase/2

**Templates**
phrase(?term, ?list, ?list)
phrase(?term, ?list)

Description

phrase(Phrase, List, Remainder) succeeds if the list List is in the language defined by the grammar rule body Phrase. Remainder is what remains of the list after a phrase has been found.

phrase(Phrase, List) is equivalent to phrase(Phrase, List, []).

Errors

| List is neither a list nor a partial list | type_error(list, List) |
| Remainder is neither a list nor a partial list | type_error(list, Remainder) |

Portability

GNU Prolog predicates.

6.18 Logic, control and exceptions

6.18.1 abort/0, stop/0, top_level/0, break/0, halt/1, halt/0

Templates

abort
stop
top_level
break
halt(+integer)
halt

Description

abort aborts the current execution. If this execution was initiated under a top-level the control is given back to the top-level and the message {execution aborted} is displayed. Otherwise, e.g. execution started by a initialization/1 directive (section 5.1.12, page 33), abort/0 is equivalent to halt(1) (see below).

stop stops the current execution. If this execution was initiated under a top-level the control is given back to the top-level. Otherwise, stop/0 is equivalent to halt(0) (see below).

top_level starts a new recursive top-level (including the banner display). To end this new top-level simply type the end-of-file key sequence (Ctrl-D) or its term representation: end_of_file.

break invokes a recursive top-level (no banner is displayed). To end this new level simply type the end-of-file key sequence (Ctrl-D) or its term representation: end_of_file.

halt(Status) causes the GNU Prolog process to immediately exit back to the shell with the return code Status.

halt is equivalent to halt(0).

Errors
<table>
<thead>
<tr>
<th>Status is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status is neither a variable nor an integer</td>
<td>type_error(integer, Status)</td>
</tr>
</tbody>
</table>

**Portability**

halt/1 and halt/0 are ISO predicates. abort/0, stop/0, top_level/0 and break/0 are GNU Prolog predicates.

### 6.18.2 call/2, once/1, (\+)/1 - not provable

**Templates**

```prolog
call(+callable_term, ?boolean)
once(+callable_term)
\+(+callable_term)
```

**Description**

call(Goal, Deterministic) succeeds if call(Goal) succeeds and unifies Deterministic with true if Goal has not created any choice-points, with false otherwise.

once(Goal) succeeds if call(Goal) succeeds. However once/1 is not re-executable on backtracking since all alternatives of Goal are cut. once(Goal) is equivalent to call(Goal), !.

\+ Goal succeeds if call(Goal) fails and fails otherwise. This built-in predicate gives negation by failure.

\+ is a predefined prefix operator (section 6.14.10, page 87).

**Errors**

<table>
<thead>
<tr>
<th>Goal is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal is neither a variable nor a callable term</td>
<td>type_error(callable, Goal)</td>
</tr>
<tr>
<td>The predicate indicator Pred of Goal does not correspond to an existing procedure and the value of the unknown Prolog flag is error (section 6.22.1, page 117)</td>
<td>existence_error(procedure, Pred)</td>
</tr>
<tr>
<td>Deterministic is neither a variable nor a boolean</td>
<td>type_error(boolean, Deterministic)</td>
</tr>
</tbody>
</table>

**Portability**

once/1 and (\+)/1 are ISO predicates, call/2 is a GNU Prolog predicate.

### 6.18.3 repeat/0

**Templates**

```prolog
repeat
```

**Description**

repeat generates an infinite sequence of backtracking choices. The purpose is to repeatedly perform some action on elements which are somehow generated, e.g. by reading them from a stream, until some test becomes true. Repeat loops cannot contribute to the logic of the program. They are only meaningful
if the action involves side-effects. The only reason for using repeat loops instead of a more natural tail-
recursive formulation is efficiency: when the test fails back, the Prolog engine immediately reclaims any
working storage consumed since the call to `repeat/0`.

**Errors**

None.

**Portability**

ISO predicate.

### 6.18.4 for/3

**Templates**

\[
\text{for}(\text{?integer, +integer, +integer})
\]

**Description**

\[
\text{for}(	ext{Counter, Lower, Upper}) \text{ generates an sequence of backtracking choices instantiating } \\
\text{Counter to the values Lower, Lower+1,..., Upper.}
\]

**Errors**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter is neither a variable nor an integer</td>
<td><code>type_error(integer, Counter)</code></td>
</tr>
<tr>
<td>Lower is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>Lower is neither a variable nor an integer</td>
<td><code>type_error(integer, Lower)</code></td>
</tr>
<tr>
<td>Upper is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>Upper is neither a variable nor an integer</td>
<td><code>type_error(integer, Upper)</code></td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.19 Atomic term processing

These built-in predicates enable atomic terms to be processed as a sequence of characters and character
codes. Facilities exist to split and join atoms, to convert a single character to and from the corresponding
character code, and to convert a number to and from a list of characters and character codes.

### 6.19.1 `atom_length/2`

**Templates**

\[
\text{atom_length}(\text{+atom, ?integer})
\]

**Description**

\[
\text{atom_length(Atom, Length)} \text{ succeeds if Length unifies with the number of characters of the name of } \\
\text{Atom.}
\]

**Errors**
<table>
<thead>
<tr>
<th>Description</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Atom is neither a variable nor an atom</td>
<td>type_error(atom, Atom)</td>
</tr>
<tr>
<td>Length is neither a variable nor an integer</td>
<td>type_error(integer, Length)</td>
</tr>
<tr>
<td>Length is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, Length)</td>
</tr>
</tbody>
</table>

Portability

ISO predicate.

6.19.2 atom_concat/3

Templates

atom_concat(+atom, +atom, ?atom)
atom_concat(?atom, ?atom, +atom)

Description

atom_concat(Atom1, Atom2, Atom12) succeeds if the name of Atom12 is the concatenation of the name of Atom1 with the name of Atom1. This predicate is re-executable on backtracking (e.g. if Atom12 is instantiated and both Atom1 and Atom2 are variables).

Errors

<table>
<thead>
<tr>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom1 and Atom12 are variables</td>
</tr>
<tr>
<td>Atom2 and Atom12 are variables</td>
</tr>
<tr>
<td>Atom1 is neither a variable nor an atom</td>
</tr>
<tr>
<td>Atom2 is neither a variable nor an atom</td>
</tr>
<tr>
<td>Atom12 is neither a variable nor an atom</td>
</tr>
</tbody>
</table>

Portability

ISO predicate.

6.19.3 sub_atom/5

Templates


Description

sub_atom(Atom, Before, Length, After, SubAtom) succeeds if atom Atom can be split into three atoms, AtomL, SubAtom and AtomR such that Before is the number of characters of the name of AtomL, Length is the number of characters of the name of SubAtom and After is the number of characters of the name of AtomR. This predicate is re-executable on backtracking.

Errors
6.19 Atomic term processing

<table>
<thead>
<tr>
<th>Atom is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom is neither a variable nor an atom</td>
<td>type_error(Atom, Atom)</td>
</tr>
<tr>
<td>SubAtom is neither a variable nor an atom</td>
<td>type_error(Atom, SubAtom)</td>
</tr>
<tr>
<td>Before is neither a variable nor an integer</td>
<td>type_error(integer, Before)</td>
</tr>
<tr>
<td>Length is neither a variable nor an integer</td>
<td>type_error(integer, Length)</td>
</tr>
<tr>
<td>After is neither a variable nor an integer</td>
<td>type_error(integer, After)</td>
</tr>
<tr>
<td>Before is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, Before)</td>
</tr>
<tr>
<td>Length is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, Length)</td>
</tr>
<tr>
<td>After is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, After)</td>
</tr>
</tbody>
</table>

Portability
ISO predicate.

6.19.4 char_code/2

Templates

\[
\text{char} \_	ext{code}(+\text{character, ?character} \_	ext{code}) \\
\text{char} \_	ext{code}(-\text{character, +character} \_	ext{code})
\]

Description

\text{char} \_	ext{code}(\text{Char, Code}) succeeds if the character code for the one-char atom \text{Char} is \text{Code}.

Errors

<table>
<thead>
<tr>
<th>Char and Code are variables</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char is neither a variable nor a one-char atom</td>
<td>type_error(character, Char)</td>
</tr>
<tr>
<td>Code is neither a variable nor an integer</td>
<td>type_error(integer, Code)</td>
</tr>
<tr>
<td>Code is an integer but not a character code</td>
<td>representation_error(character, code)</td>
</tr>
</tbody>
</table>

Portability
ISO predicate.

6.19.5 \text{lower.upper}/2

Templates

\[
\text{lower} \_	ext{upper}(+\text{character, ?character}) \\
\text{lower} \_	ext{upper}(-\text{character, +character})
\]

Description

\text{lower} \_	ext{upper}(\text{Char1, Char2}) succeeds if \text{Char1} and \text{Char2} are one-char atoms and if \text{Char2} is the upper conversion of \text{Char1}. If \text{Char1} (resp. \text{Char2}) is a character that is not a lower (resp. upper) letter then \text{Char2} is equal to \text{Char1}.

Errors

<table>
<thead>
<tr>
<th>Char1 and Char2 are variables</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char1 is neither a variable nor a one-char atom</td>
<td>type_error(character, Char1)</td>
</tr>
<tr>
<td>Char2 is neither a variable nor a one-char atom</td>
<td>type_error(character, Char2)</td>
</tr>
</tbody>
</table>
Portability

GNU Prolog predicate.

6.19.6 atom_chars/2, atom_codes/2

Templates

atom_chars(+atom, ?character_list)
anom_chars(-atom, +character_list)
atom_codes(+atom, ?character_code_list)
atom_codes(-atom, +character_code_list)

Description

atom_chars(Atom, Chars) succeeds if Chars is the list of one-char atoms whose names are the successive characters of the name of Atom.

atom_codes(Atom, Codes) is similar to atom_chars/2 but deals with a list of character codes.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom is a variable and Chars (or Codes) is a partial list or a list with an element which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Atom is neither a variable nor an atom</td>
<td>type_error(atom, Atom)</td>
</tr>
<tr>
<td>Chars is neither a list nor a partial list</td>
<td>type_error(list, Chars)</td>
</tr>
<tr>
<td>Codes is neither a list nor a partial list</td>
<td>type_error(list, Codes)</td>
</tr>
<tr>
<td>Atom is a variable and an element E of the list Chars is neither a variable nor a one-char atom</td>
<td>type_error(character, E)</td>
</tr>
<tr>
<td>Atom is a variable and an element E of the list Codes is neither a variable nor an integer</td>
<td>type_error(integer, E)</td>
</tr>
<tr>
<td>Atom is a variable and an element E of the list Codes is an integer but not a character code</td>
<td>representation_error(character, code)</td>
</tr>
</tbody>
</table>

Portability

ISO predicates. The ISO reference only causes a `type_error(list, Chars)` if Atom is a variable and Chars is neither a list nor a partial list. GNU Prolog always checks if Chars is a list. Similarly for Codes. The `type_error(integer, E)` when an element E of the Codes is not an integer is a GNU Prolog extension. This seems to be an omission in the ISO reference since this error is detected for many other built-in predicates accepting a character code (e.g. `char_code/2, put_code/2`).

6.19.7 number_atom/2, number_chars/2, number_codes/2

Templates

number_atom(+number, ?atom)
number_atom(-number, +atom)
number_chars(+number, ?character_list)
number_chars(-number, +character_list)
number_codes(+number, ?character_code_list)
number_codes(-number, +character_code_list)
Description

`number_atom(Number, Atom)` succeeds if `Atom` is an atom whose name corresponds to the characters of `Number`.

`number_chars(Number, Chars)` is similar to `number_atom/2` but deals with a list of character codes.

`number_codes(Number, Codes)` is similar to `number_atom/2` but deals with a list of characters.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Number</code> and <code>Atom</code> are variables</td>
<td>instantiation_error</td>
</tr>
<tr>
<td><code>Number</code> is a variable and <code>Chars</code> (or <code>Codes</code>) is a partial list or a list with an element which is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td><code>Number</code> is neither a variable nor an number</td>
<td><code>type_error(number, Number)</code></td>
</tr>
<tr>
<td><code>Atom</code> is neither a variable nor an atom</td>
<td><code>type_error(atom, Atom)</code></td>
</tr>
<tr>
<td><code>Number</code> is a variable and <code>Chars</code> is neither a list nor a partial list</td>
<td><code>type_error(list, Chars)</code></td>
</tr>
<tr>
<td><code>Number</code> is a variable and <code>Codes</code> is neither a list nor a partial list</td>
<td><code>type_error(list, Codes)</code></td>
</tr>
<tr>
<td><code>Number</code> is a variable and an element <code>E</code> of the list <code>Chars</code> is neither a variable nor a one-char atom</td>
<td><code>type_error(character, E)</code></td>
</tr>
<tr>
<td><code>Number</code> is a variable and an element <code>E</code> of the list <code>Codes</code> is neither a variable nor an integer</td>
<td><code>type_error(integer, E)</code></td>
</tr>
<tr>
<td><code>Number</code> is a variable and an element <code>E</code> of the list <code>Codes</code> is an integer but not a character code</td>
<td><code>representation_error(character, code)</code></td>
</tr>
<tr>
<td><code>Number</code> is a variable, <code>Atom</code> (or <code>Chars</code> or <code>Codes</code>) cannot be parsed as a number and the value of the <code>syntax_error</code> Prolog flag is <code>error</code> (section 6.22.1, page 117)</td>
<td><code>syntax_error(atom explaining the error)</code></td>
</tr>
</tbody>
</table>

Portability

`number_atom/2` is a GNU Prolog predicate. `number_chars/2` and `number_codes/2` are ISO predicates.

GNU Prolog only raises an error about an element `E` of the `Chars` (or `Codes`) list when `Number` is a variable while the ISO reference always check this. This seems an error since the list itself is only checked if `Number` is a variable.

The `type_error(integer, E)` when an element `E` of the `Codes` is not an integer is a GNU Prolog extension. This seems to be an omission in the ISO reference since this error is detected for many other built-in predicates accepting a character code (e.g. `char_code/2`, `put_code/2`).

6.19.8 `name/2`

Templates

```
name(+atomic, ?character_code_list)
name(-atomic, +character_code_list)
```

Description

`name(Constant, Codes)` succeeds if `Codes` is a list whose elements are the character codes corresponding to the successive characters of `Constant` (a number or an atom). However, there atoms are for which
name(Constant, Codes) is true, but which will not be constructed if name/2 is called with Constant uninstantiated, e.g. the atom '1024'. For this reason the use of name/2 is discouraged and should be limited to compatibility purposes. It is preferable to use atom_codes/2 (section 6.19.6, page 104) or number_chars/2 (section 6.19.7, page 104).

Errors

| Constant is a variable and Codes is a partial list or a list with an element which is a variable | instantiation_error |
| Constant is neither a variable nor an atomic term | type_error(atomic, Constant) |
| Constant is a variable and Codes is neither a list nor a partial list | type_error(list, Codes) |
| Constant is a variable and an element E of the list Codes is neither a variable nor an integer | type_error(integer, E) |
| Constant is a variable and an element E of the list Codes is an integer but not a character code | representation_error(character_code) |

Portability

GNU Prolog predicate.

6.19.9 atom_hash/2

Templates

atom_hash(+atom, ?integer)
atom_hash(?atom, +integer)

Description

atom_hash(Atom, Hash) succeeds if Hash is the internal key associated to Atom (an existing atom). The internal key of an atom is a unique integer ≥ 0 and < to the max_atom Prolog flag (section 6.22.1, page 117).

Errors

| Atom and Hash are both variables | instantiation_error |
| Atom is neither a variable nor an atom | type_error(atom, Atom) |
| Hash is neither a variable nor an integer | type_error(integer, Hash) |
| Hash is an integer < 0 | domain_error(not_less_than_zero, Hash) |

Portability

GNU Prolog predicate.

6.19.10 new_atom/3, new_atom/2, new_atom/1

Templates

new_atom(+atom, +integer, -atom)
new_atom(+atom, -atom)
new_atom(-atom)
Description

new_atom(Prefix, Hash, Atom) unifies Atom with a new atom whose name begins with the characters of the name of Prefix and whose internal key is Hash (section 6.19.9, page 106). This predicate is then a symbol generator. It is guaranteed that Atom does not exist before the invocation of new_atom/3. The characters appended to Prefix to form Atom are in: A-Z (capital letter), a-z (small letter), 0-9 (digit), #, $, &. new_atom/2 is similar to new_atom/3, but the atom generated can have any (free) internal key.

new_atom/1 is similar to new_atom(atom_, Atom), i.e. the generated atom begins with atom_.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Hash is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Prefix is neither a variable nor an atom</td>
<td>type_error(atomic, Prefix)</td>
</tr>
<tr>
<td>Hash is neither a variable nor an integer</td>
<td>type_error(integer, Hash)</td>
</tr>
<tr>
<td>Hash is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, Hash)</td>
</tr>
<tr>
<td>Atom is not a variable</td>
<td>type_error(variable, Atom)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.19.11 current_atom/1

Templates

current_atom(?atom)

Description

current_atom(Atom) succeeds if there exists an atom that unifies with Atom. All atoms are found except those beginning with a ',' (system atoms). This predicate is re-executable on backtracking.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom is neither a variable nor an atom</td>
<td>type_error(atomic, Atom)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.19.12 atom_property/2

Templates

atom_property(?atom, ?atom_property)

Description

atom_property(Atom, Property) succeeds if current_atom(Atom) succeeds (section 6.19.11, page 107) and if Property unifies with one of the properties of the atom. This predicate is re-executable on backtracking.
Atom properties:

- `length(Length)`: Length is the length of the name of the atom.
- `hash(Hash)`: Hash is the internal key of the atom (section 6.19.9, page 106).
- `prefix_op`: if there is a prefix operator currently defined with this name.
- `infix_op`: if there is an infix operator currently defined with this name.
- `postfix_op`: if there is a postfix operator currently defined with this name.
- `needs_quotes`: if the atom must be quoted to be read later.
- `needs_scan`: if the atom must be scanned when output to be read later (e.g. contains special characters that must be output with a \ escape sequence).

Errors

| Atom is neither a variable nor an atom | type_error(atom, Atom) |
| Property is neither a variable nor a n atom property term | domain_error(atom_property, Property) |
| Property = length(E) or hash(E) and E is neither a variable nor an integer | type_error(integer, E) |

Portability

GNU Prolog predicate.

### 6.20 List processing

These predicates manipulate lists. They are bootstrapped predicates (i.e. written in Prolog) and no error cases are tested (for the moment). However, since they are written in Prolog using other built-in predicates, some errors can occur due to those built-in predicates.

#### 6.20.1 append/3

**Templates**

```prolog
append(?list, ?list, ?list)
```

**Description**

`append(List1, List2, List12)` succeeds if the concatenation of the list `List1` and the list `List2` is the list `List12`. This predicate is re-executable on backtracking (e.g. if `List12` is instantiated and both `List1` and `List2` are variable).

**Errors**

None.

**Portability**

GNU Prolog predicate.
6.20.2 member/2, memberchk/2

Templates

member(?term, ?list)
memberchk(?term, ?list)

Description

member(Element, List) succeeds if Element belongs to the List. This predicate is re-executable on backtracking and can be thus used to enumerate the elements of List.

memberchk/2 is similar to member/2 but only succeeds once.

Errors

None.

Portability

GNU Prolog predicate.

6.20.3 reverse/2

Templates

reverse(?list, ?list)

Description

reverse(List1, List2) succeeds if List2 unifies with the list List1 in reverse order.

Errors

None.

Portability

GNU Prolog predicate.

6.20.4 delete/3, select/3

Templates

delete(?list, ?term, ?list)
select(?term, ?list, ?list)

Description

delete(List1, Element, List2) removes all occurrences of Element in List1 to provide List2. A strict term equality is required, cf. (==)/2 (section 6.3.2, page 39).

select(Element, List1, List2) removes one occurrence of Element in List1 to provide List2. This predicate is re-executable on backtracking.
Errors

None.

Portability

GNU Prolog predicate.

6.20.5 permutation/2

Templates

\texttt{permutation(?list, ?list)}

Description

\texttt{permutation(List1, List2)} succeeds if \texttt{List2} is a permutation of the elements of \texttt{List1}. This predicate is re-executable on backtracking.

Errors

None.

Portability

GNU Prolog predicate.

6.20.6 prefix/2, suffix/2

Templates

\begin{align*}
\texttt{prefix(?list, ?list)} \\
\texttt{suffix(?list, ?list)}
\end{align*}

Description

\texttt{prefix(Prefix, List)} succeeds if \texttt{Prefix} is a prefix of \texttt{List}. This predicate is re-executable on backtracking.

\texttt{suffix(Suffix, List)} succeeds if \texttt{Suffix} is a suffix of \texttt{List}. This predicate is re-executable on backtracking.

Errors

None.

Portability

GNU Prolog predicate.

6.20.7 sublist/2

Templates
sublist(?list, ?list)

Description

`sublist(List1, List2)` succeeds if `List2` is a sub-list of `List1`. This predicate is re-executable on backtracking.

Errors

None.

Portability

GNU Prolog predicate.

6.20.8 last/2

Templates

`last(?list, ?term)`

Description

`last(List, Element)` succeeds if `Element` is the last element of `List`.

Errors

None.

Portability

GNU Prolog predicate.

6.20.9 length/2

Templates

`length(?list, ?integer)`

Description

`length(List, Length)` succeeds if `Length` is the length of `List`.

Errors

None.

Portability

GNU Prolog predicate.

6.20.10 nth/3

Templates
nth(+integer, ?list, ?term)

Description

nth(N, List, Element) succeeds if the Nth argument of List is Element.

Errors

None.

Portability

GNU Prolog predicate.

6.20.11  max_list/2, min_list/2, sum_list/2

Templates

min_list(+list, ?number)
max_list(+list, ?number)
sum_list(+list, ?number)

Description

min_list(List, Min) succeeds if Min is the smallest number in List.
max_list(List, Max) succeeds if Max is the largest number in List.
sum_list(List, Sum) succeeds if Sum is the sum of all the elements in List.
List must be a list of arithmetic evaluable terms (section 6.6.1, page 45).

Errors

None.

Portability

GNU Prolog predicate.

6.20.12  sort/2, sort0/2, keysort/2 sort/1, sort0/1, keysort/1

Templates

sort(+list, ?list)
sort0(+list, ?list)
keysort(+list, ?list)
sort(+list)
sort0(+list)
keysort(+list)

Description

sort(List1, List2) succeeds if List2 is the sorted list corresponding to List1 where duplicate elements are merged.
sort0/2 is similar to sort/2 except that duplicate elements are not merged.

keysort(List1, List2) succeeds if List2 is the sorted list of List1 according to the keys. The list
List1 consists of items of the form Key-Value. These items are sorted according to the value of Key
yielding the List2. Duplicate keys are not merged. This predicate is stable, i.e. if K-A occurs before K-B
in the input, then K-A will occur before K-B in the output.

sort/1, sort0/1 and keysort/1 are similar to sort/2, sort0/2 and keysort/2 but achieve a sort in-
place destructing the original List1 (this in-place assignment is not undone at backtracking). The sorted
list occupies the same memory space as the original list (saving thus memory consumption).

The time complexity of these sorts is \( O(N \log N) \), \( N \) being the length of the list to sort.

These predicates refer to the standard ordering of terms (section 6.3.1, page 39).

Errors

<table>
<thead>
<tr>
<th>List1 is a partial list</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>List1 is neither a partial list nor a list</td>
<td>type_error(list, List1)</td>
</tr>
<tr>
<td>List2 is neither a partial list nor a list</td>
<td>type_error(list, List2)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.21 Global variables

GNU Prolog provides a simple and powerful way to assign and read global variables. There are 3 kinds
of objects which can be associated to a global variable:

- a copy of a term (the assignment can be made backtrackable or not).
- a link to a term.
- an array of objects.

The initial value of a global variable is the integer 0. A global variable is referenced by a name (i.e. name
= an atom) possibly indexed if it is an array (i.e. name = a compound term). In the following, \( GVarName \)
represents such a reference to a global variable and its syntax is as follows:

\[
\begin{align*}
GVarName & ::= \ atom \\
& \quad \quad \text{atom(Index,...,Index)} \\
Index & ::= \ integer \\
& \quad \quad GVarName
\end{align*}
\]

When a \( GVarName \) is used as an index, the value of this variable must be an integer. Indexes range from
0 to Size-1 if the array has Size elements. The space necessary for copies and arrays are dynamically
allocated and recovered as soon as possible. For instance, when an atom is associated to a global variable
whose current value is an array, the space for this array is recovered (unless the assignment is to be
undone when backtracking occurs).

Arrays: the predicates \( g\_assign/2, g\_assignb/2 \) and \( g\_link/2 \) (section 6.21.1, page 114) define an
array when the assigned value is a compound term with principal functor \( g\_array \). Then an array is
assigned to \( GVarName \). There are 3 forms for the term \( g\_array \):

- \( g\_array(\text{Size}) \): if \( \text{Size} \) is an integer > 0 then defines an array of \( \text{Size} \) elements which are all
  initialized with the integer 0.
- $g_{\text{array}}(\text{Size, Initial})$: as above but the elements are initialized with the term Initial instead of 0. Initial can contain other array definitions allowing thus for multi-dimensional arrays. Some examples are presented later (section 6.21.4, page 115).

- $g_{\text{array}}(\text{List})$: as above if List is a list of length Size except that the elements of the array are initialized according to the elements of List (which can contain other array definitions).

The compound term with principal functor $g_{\text{array}.extend}$ can be used instead of $g_{\text{array}}$ to modify the structure of a (possibly) existing array. In that case, the existing elements of the array are not initialized.

When an array is read, a term of the form $g_{\text{array}}([\text{Elem0}, \ldots, \text{ElemSize-1}])$ is returned.

### 6.21.1 $g_{\text{assign/2, g_assignb/2, g_link/2}}$

**Templates**

$g_{\text{assign}}(+\text{callable_term, ?term})$
$g_{\text{assignb}}(+\text{callable_term, ?term})$
$g_{\text{link}}(+\text{callable_term, ?term})$

**Description**

$g_{\text{assign}}(\text{GVarName, Value})$ assigns a copy of the term Value to GVarName. This assignment is not undone when backtracking occurs.

$g_{\text{assignb/2}}$ is similar to $g_{\text{assign/2}}$ but the assignment is undone at backtracking.

$g_{\text{link}}(\text{GVarName, Value})$ makes a link between GVarName to the term Value. This allows the user to give a name to any Prolog term (in particular non-ground terms). Such an assignment is always undone when backtracking occurs (since the term may no longer exist). If Value is an atom or an integer, $g_{\text{link/2}}$ and $g_{\text{assignb/2}}$ have the same behavior. Since $g_{\text{link/2}}$ only handles links to existing terms it does not require extra memory space and is not expensive in terms of execution time.

**Errors**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>GVarName is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>GVarName is neither a variable nor a callable term</td>
<td>type_error(Callable, GVarName)</td>
</tr>
<tr>
<td>GVarName is a compound term and a sub-argument E is not a valid index (section 6.21, page 113)</td>
<td>domain_error(g_array_index, E)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

### 6.21.2 $g_{\text{read/2}}$

**Templates**

$g_{\text{read}}(+\text{callable_term, ?term})$

**Description**

$g_{\text{read}}(\text{GVarName, Value})$ unifies Value with the term assigned to GVarName.
6.21 Global variables

Errors

| GVarName is a variable        | instantiation_error       |
| GVarName is neither a variable nor a callable term | type_error(calling, GVarName) |
| GVarName is a compound term and a sub-argument E is not a valid index (section 6.21, page 113) | domain_error(g_array_index, E) |

Portability

GNU Prolog predicate.

6.21.3 g_array_size/2

Templates

\[ g\_array\_size(+\text{callable\_term}, ?\text{integer}) \]

Description

\( g\_array\_size(\text{GVarName}, \text{Value}) \) unifies \text{Size} with the dimension (an integer > 0) of the array assigned to \text{GVarName}. Fails if \text{GVarName} is not an array.

Errors

| GVarName is a variable        | instantiation_error       |
| GVarName is neither a variable nor a callable term | type_error(calling, GVarName) |
| GVarName is a compound term and a sub-argument E is not a valid index (section 6.21, page 113) | domain_error(g_array_index, E) |
| \text{Size} is neither a variable nor an integer | type_error(integer, Size) |

Portability

GNU Prolog predicate.

6.21.4 Examples

A simple counter: the following predicate defines a counter using a global variable:

\[ \text{inc(Var, Value)} :- \]
\[ \text{g\_read(Var, Value)}, \]
\[ \text{X is Value+1}, \]
\[ \text{g\_assign(Var, X)}. \]

The query: \text{inc(c, X)} will succeed unifying \text{X} with 0, another call to \text{inc(a, Y)} will then unify \text{Y} with 1, and so on.

Difference between \text{g\_assign/2} and \text{g\_assignb/2}: \text{g\_assign/2} does not undo its assignment when backtracking occurs whereas \text{g\_assignb/2} undoes it.
test(0ld) :-
  g_assign(x, 1),
  ( g_read(x, 0ld),
    g_assign(x, 2)
  ; g_read(x, 0ld),
    g_assign(x, 3)
 ).

testb(0ld) :-
  g_assign(x, 1),
  ( g_read(x, 0ld),
    g_assignb(x, 2)
  ; g_read(x, 0ld),
    g_assign(x, 3)
  ).

The query test(0ld) will succeed unifying 0ld with 1 and on backtracking with 2 (i.e. the assignment of the value 2 has not been undone). The query testb(0ld) will succeed unifying 0ld with 1 and on backtracking with 1 (i.e. the assignment of the value 2 has been undone).

Difference between g_assign/2 and g_link/2: g_assign/2 (and g_assignb/2) creates a copy of the term whereas g_link/2 does not. g_link/2 can be used to avoid passing big data structures (e.g. dictionaries,...) as arguments to predicates.

test(B) :-
  g_assign(b, f(X)),
  X = 12,
  g_read(b, B).

test(B) :-
  g_link(b, f(X)),
  X = 12,
  g_read(b, B).

The query test(B) will succeed unifying B with f (_) (g_assign/2 assigns a copy of the value). The query test1(B) will succeed unifying B with f(12) (g_link/2 assigns a pointer to the term).

Simple array definition: here are some queries to show how arrays can be handled:

| ?- g_assign(w, g_array(3)), g_read(w, X). |
| X = g_array([0,0,0]) |

| ?- g_assign(w(0), 16), g_assign(w(1), 32), g_assign(w(2), 64), g_read(w, X). |
| X = g_array([16,32,64]) |

this is equivalent to:

| ?- g_assign(k, g_array([16,32,64])), g_read(k, X). |
| X = g_array([16,32,64]) |

| ?- g_assign(k, g_array(3,null)), g_read(k, X), g_array_size(k, S). |
| S = 3 |
| X = g_array([null,null,null]) |

2-D array definition:

| ?- g_assign(w, g_array(2, g_array(3))), g_read(w, X). |
| X = g_array([g_array([0,0,0]),g_array([0,0,0])]) |

| ?- ( for(I,0,1), for(J,0,2), K is I*3+J, g_assign(w(I,J), K), fail ; g_read(w, X) |
| X = g_array([g_array([0,1,2]),g_array([3,4,5])]) |
| ?- g_read(w(1),X).
X = g_array([3,4,5])

Hybrid array:
| ?- g_assign(w,g_array([1,2,g_array([a,b,c]), g_array(2,z),5])), g_read(w, X).
X = g_array([1,2,g_array([a,b,c]), g_array([z,z],5)])
| ?- g_read(w(1), X), g_read(w(2,1), Y), g_read(w(3,1), Z).
X = 2
Y = b
Z = z
| ?- g_read(w(1,2),X).
{exception: error(domain_error(g_array_index,w(1,2)),g_read/2)}

Array extension:
| ?- g_assign(a, g_array([10,20,30])), g_read(a, X).
X = g_array([10,20,30])
| ?- g_assign(a, g_array_extend(5,null)), g_read(a, X).
X = g_array([10,20,30,null,null])
| ?- g_assign(a, g_array([10,20,30])), g_read(a, X).
X = g_array([10,20,30])
| ?- g_assign(a, g_array_extend([1,2,3,4,5,6])), g_read(a, X).
X = g_array([10,20,30,4,5,6])

6.22 Prolog state

6.22.1 set_prolog_flag/2

Templates

set_prolog_flag(+flag, +term)

Description

set_prolog_flag(Flag, Value) sets the value of the Prolog flag Flag to Value.

Prolog flags: a Prolog flag is an atom which is associated with a value that is either implementation defined or defined by the user. Each flag has a permitted range of values; any other value is a domain_error. The following two tables present available flags, the possible values, a description and if they are ISO or an extension. The first table presents unchangeable flags while the second one the changeable flags. For flags whose default values is machine independent, this value is underlined.
Unchangeable flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Values</th>
<th>Description</th>
<th>ISO</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounded</td>
<td>true / false</td>
<td>are integers bounded?</td>
<td>Y</td>
</tr>
<tr>
<td>max_integer</td>
<td>an integer</td>
<td>greatest integer</td>
<td>Y</td>
</tr>
<tr>
<td>min_integer</td>
<td>an integer</td>
<td>smallest integer</td>
<td>Y</td>
</tr>
<tr>
<td>integer_rounding_function</td>
<td>toward zero</td>
<td>( \text{round}(x) = \text{integer part of } x ) (section 6.6.1, page 45)</td>
<td>Y</td>
</tr>
<tr>
<td>max_arity</td>
<td>an integer</td>
<td>maximum arity for compound terms (255)</td>
<td>Y</td>
</tr>
<tr>
<td>max_atom</td>
<td>an integer</td>
<td>maximum number of atoms</td>
<td>N</td>
</tr>
<tr>
<td>max_stream</td>
<td>an integer</td>
<td>maximum number of streams</td>
<td>N</td>
</tr>
<tr>
<td>max_unget</td>
<td>an integer</td>
<td>maximum number of successive ungets</td>
<td>N</td>
</tr>
<tr>
<td>prolog_name</td>
<td>an atom</td>
<td>name of the Prolog system</td>
<td>N</td>
</tr>
<tr>
<td>prolog_version</td>
<td>an atom</td>
<td>version number of the Prolog system</td>
<td>N</td>
</tr>
<tr>
<td>prolog_date</td>
<td>an atom</td>
<td>date of the Prolog system</td>
<td>N</td>
</tr>
<tr>
<td>prolog_copyright</td>
<td>an atom</td>
<td>copyright message of the Prolog system</td>
<td>N</td>
</tr>
</tbody>
</table>

Changeable flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Values</th>
<th>Description</th>
<th>ISO</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_conversion</td>
<td>on / off</td>
<td>is character conversion activated?</td>
<td>Y</td>
</tr>
<tr>
<td>debug</td>
<td>on / off</td>
<td>is the debugger activated?</td>
<td>Y</td>
</tr>
<tr>
<td>singleton_warning</td>
<td>on / off</td>
<td>warn about named singleton variables?</td>
<td>N</td>
</tr>
<tr>
<td>strict_iso</td>
<td>on / off</td>
<td>strict ISO behavior?</td>
<td>N</td>
</tr>
<tr>
<td>double_quotes</td>
<td>atom</td>
<td>a double quoted constant is returned as:</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>chars</td>
<td>an atom</td>
<td></td>
</tr>
<tr>
<td></td>
<td>codes</td>
<td>a list of characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a list of character codes</td>
<td></td>
</tr>
<tr>
<td>unknown</td>
<td>error</td>
<td>a predicate calls an unknown procedure:</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>warning</td>
<td>an existence_error is raised</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fail</td>
<td>a message is displayed then fails quietly fails</td>
<td></td>
</tr>
<tr>
<td>syntax_error</td>
<td>error</td>
<td>a predicate causes a syntax error:</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>warning</td>
<td>a syntax_error is raised</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fail</td>
<td>a message is displayed then fails quietly fails</td>
<td></td>
</tr>
<tr>
<td>os_error</td>
<td>error</td>
<td>a predicate causes an O.S. error:</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>warning</td>
<td>a system_error is raised</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fail</td>
<td>a message is displayed then fails quietly fails</td>
<td></td>
</tr>
</tbody>
</table>

The `strict_iso` flag is introduced to allow a compatibility with other Prolog systems. When turned off the following relaxations apply:

- a callable term can be given as a predicate indicator.
- built-in predicates are found by `current_predicate/1` (section 6.8.1, page 52).

Errors

<table>
<thead>
<tr>
<th>Flag is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Flag is neither a variable nor an atom</td>
<td>type_error(atom, Flag)</td>
</tr>
<tr>
<td>Flag is an atom but not a valid flag</td>
<td>domain_error(prolog_flag, Flag)</td>
</tr>
<tr>
<td>Value is inappropriate for Flag</td>
<td>domain_error(flag_value, Flag+Value)</td>
</tr>
<tr>
<td>Value is appropriate for Flag but flag Flag is not modifiable</td>
<td>permission_error(modify, flag, Flag)</td>
</tr>
</tbody>
</table>
Portability
ISO predicate. All ISO flags are implemented.

6.22.2 current_prolog_flag/2

Templates

```prolog
current_prolog_flag(?flag, ?term)
```

Description

`current_prolog_flag(Flag, Value)` succeeds if there exists a Prolog flag that unifies with `Flag` and whose value unifies with `Value`. This predicate is re-executable on backtracking.

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag is neither a variable nor an atom</td>
<td><code>type_error(atom, Flag)</code></td>
</tr>
<tr>
<td>Flag is an atom but not a valid flag</td>
<td><code>domain_error(prolog_flag, Flag)</code></td>
</tr>
</tbody>
</table>

Portability

ISO predicate.

6.22.3 set_bip_name/2

Templates

```prolog
set_bip_name(+atom, +arity)
```

Description

`set_bip_name(Functor, Arity)` initializes the context of the error (section 4.3.1, page 25) with `Functor` and `Arity` (if `Arity < 0` only `Functor` is significant).

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functor is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>Arity is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>Functor is neither a variable nor an atom</td>
<td><code>type_error(atom, Functor)</code></td>
</tr>
<tr>
<td>Arity is neither a variable nor an integer</td>
<td><code>type_error(integer, Arity)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.22.4 current_bip_name/2

Templates

```prolog
current_bip_name(?atom, ?arity)
```
Description

current_bip_name(Functor, Arity) succeeds if Functor and Arity correspond to the context of the error (section 4.3.1, page 25) (if Arity < 0 only Functor is significant).

Errors

| Functor is neither a variable nor an atom | type_error(atomic, Functor) |
| Arity is neither a variable nor an integer | type_error(integer, Arity) |

Portability

GNU Prolog predicate.

6.22.5 write_pl_state_file/1, read_pl_state_file/1

Templates

write_pl_state_file(+source_sink)
read_pl_state_file(+source_sink)

Description

write_pl_state_file(FileName) writes onto FileName all information that influences the parsing of a term (section 6.14, page 78). This allows a sub-process written in Prolog to read this file and then process any Prolog term as done by the parent process. This file can also be passed as argument of the --pl-state option when invoking gplc (section 2.4.3, page 12). More precisely the following elements are saved:

- the value of char_conversion, double_quotes and singleton_warning Prolog flags (section 6.22.1, page 117).

read_pl_state_file(FileName) reads (restores) from FileName all information previously saved by write_pl_state_file/1.

Errors

| FileName is a variable | instantiation_error |
| FileName is neither a variable nor an atom | type_error(atomic, FileName) |
| an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117) | system_error(atomic explaining the error) |

Portability

GNU Prolog predicate.

6.23 Program state

6.23.1 consult/1, ’.’/2 - program consult

Templates
consult(+atom_or_atom_list)
'.'(atom, +atom_list)

Description

consult(Files) compiles and loads into memory each file of the list Files. Each file is compiled for byte-code using the GNU Prolog compiler (section 2.4, page 10) then loaded using load/1 (section 6.23.2, page 121). It is possible to specify user as a file name to directly enter the program from the terminal. Files can be also a single file name (i.e. an atom). Refer to the section concerning the consult of a Prolog program for more information (section 2.2.3, page 6).

The final file name of a file is computed using the predicates prolog_file_name/2 (section 6.26.3, page 126) and absolute_file_name/2 (section 6.26.1, page 125).

[ File | Files ], i.e. '.'(File, Files) is equivalent to consult([ File | Files ]).

Errors

| Files is a partial list or a list with an element E which is a variable | instantiation_error |
| Files is neither a partial list nor a list nor an atom | type_error(list, Files) |
| an element E of the Files list is neither a variable nor an atom | type_error(atom, E) |
| an element E of the Files list is an atom but not a valid pathname | domain_error(os_path, E) |
| an element E of the Files list is a valid path name but does not correspond to an existing source | existence_error(source,sink, E) |
| an error occurs executing a directive | see call/1 errors (section 5.23, page 35) |

Portability

GNU Prolog predicates.

6.23.2 load/1

Templates

load(+atom_or_atom_list)

Description

load(Files) loads into memory each file of the list Files. Each file must have been previously compiled for byte-code using the GNU Prolog compiler (section 2.4, page 10). Files can be also a single file name (i.e. an atom).

The final file name of a file is computed using the predicates absolute_file_name/2 (section 6.26.1, page 125). If no suffix is given '.wbc' is appended to the file name.

Errors
**Files** is a partial list or a list with an element E which is a variable: `instantiation_error`  
**Files** is neither a partial list nor a list nor an atom: `type_error(list, Files)`  
an element E of the Files list is neither a variable nor an atom: `type_error(atom, E)`  
an element E of the Files list is an atom but not a valid path name: `domain_error(os_path, E)`  
an element E of the Files list is a valid path name but does not correspond to an existing source: `existence_error(source_sink, E)`  
an error occurs executing a directive: see call/1 errors (section 5.23, page 35)

**Portability**

GNU Prolog predicate.

### 6.23.3 listing/1, listing/0

**Templates**

```prolog
listing(+predicate_indicator)  
listing(+atom)  
listing
```

**Description**

`listing(Pred)` lists the clauses of the consulted predicate whose predicate indicator is `Pred`. `Pred` can also be a single atom in which case all predicates whose name is `Pred` are listed (of any arity). This predicate uses `portray_clause/2` (section 6.14.8, page 86) to output the clauses.

`listing` lists all clauses of all consulted predicates.

**Errors**

<table>
<thead>
<tr>
<th>Pred is a variable</th>
<th><code>instantiation_error</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pred is neither a variable nor predicate indicator or an atom</td>
<td><code>type_error(predicate_indicator, Pred)</code></td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.24 System statistics

#### 6.24.1 statistics/0, statistics/2

**Templates**

```prolog
statistics  
statistics(?atom, ?list)
```

**Description**
statistics displays statistics about memory usage and run times.

statistics(Key, Value) unifies Value with the current value of the statistics key Key. Value a list of two elements. Times are in milliseconds, sizes of areas in bytes.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_time</td>
<td>user CPU time</td>
<td>[SinceStart, SinceLast]</td>
</tr>
<tr>
<td>system_time</td>
<td>system CPU time</td>
<td>[SinceStart, SinceLast]</td>
</tr>
<tr>
<td>cpu_time</td>
<td>total CPU time (user + system)</td>
<td>[SinceStart, SinceLast]</td>
</tr>
<tr>
<td>real_time</td>
<td>absolute time</td>
<td>[SinceStart, SinceLast]</td>
</tr>
<tr>
<td>local_stack</td>
<td>local stack sizes (control, environments, choices)</td>
<td>[UsedSize, FreeSize]</td>
</tr>
<tr>
<td>global_stack</td>
<td>global stack sizes (compound terms)</td>
<td>[UsedSize, FreeSize]</td>
</tr>
<tr>
<td>trail_stack</td>
<td>trail stack sizes (variable bindings to undo)</td>
<td>[UsedSize, FreeSize]</td>
</tr>
<tr>
<td>cstr_stack</td>
<td>constraint trail sizes (finite domain constraints)</td>
<td>[UsedSize, FreeSize]</td>
</tr>
</tbody>
</table>

Note that the key runtime is recognized as user_time for compatibility purpose.

Errors

<table>
<thead>
<tr>
<th>Key is neither a variable nor a valid key</th>
<th>domain_error(statistics_key, Key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value is neither a variable nor a list of two elements</td>
<td>domain_error(statistics_value, Value)</td>
</tr>
<tr>
<td>Value is a list of two elements and an element E is neither a variable nor an integer</td>
<td>type_error(integer, E)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.24.2 user_time/1, system_time/1, cpu_time/1, real_time/1

Templates

user_time(?integer)
system_time(?integer)
cpu_time(?integer)
real_time(?integer)

Description

user_time(Time) unifies Time with the user CPU time elapsed since the start of Prolog.
system_time(Time) unifies Time with the system CPU time elapsed since the start of Prolog.
cpu_time(Time) unifies Time with the CPU time (user + system) elapsed since the start of Prolog.
real_time(Time) unifies Time with the absolute time elapsed since the start of Prolog.

Errors

| Time is neither a variable nor an integer | type_error(integer, Time) |

Portability

GNU Prolog predicates.
6.25 Random number generator

6.25.1 set_seed/1, randomize/0

Templates

\[
\text{set} \_\text{seed}(+\text{integer})
\]
\[
\text{randomize}
\]

Description

\text{set} \_\text{seed}(\text{Seed}) \text{ reinitializes the random number generator seed with Seed.}

\text{randomize} \text{ reinitializes the random number generator. This predicates calls set} \_\text{seed}/1 \text{ with a random value depending on the absolute time.}

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Seed is neither a variable nor an integer</td>
<td>type_error(integer, Seed)</td>
</tr>
<tr>
<td>Seed is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, Seed)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.25.2 get_seed/1

Templates

\[
\text{get} \_\text{seed}(\text{?integer})
\]

Description

\text{get} \_\text{seed} \text{ (Seed) unifies Seed with the current random number generator seed.}

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed is neither a variable nor an integer</td>
<td>type_error(integer, Seed)</td>
</tr>
<tr>
<td>Seed is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, Seed)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.25.3 random/1

Templates

\[
\text{random(-float)}
\]

Description

\text{random} \text{ (Number) unifies Number with a random floating point number such that } 0.0 \leq \text{Number} < 1.0.
Errors

<table>
<thead>
<tr>
<th>Number is not a variable</th>
<th>type_error(variable, Number)</th>
</tr>
</thead>
</table>

Portability

GNU Prolog predicate.

6.25.4 random/3

Templates

\[
\text{random}(\text{number}, \text{number}, \text{-number})
\]

Description

\[
\text{random(Base, Max, Number)} \text{ unifies Number with a random number such that Base} \leq \text{Number} < \text{Max.}
\]

If both \(\text{Base}\) and \(\text{Max}\) are integers \(\text{Number}\) will be an integer, otherwise \(\text{Number}\) will be a floating point number.

Errors

<table>
<thead>
<tr>
<th>Base is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base is neither a variable nor a number</td>
<td>type_error(number, Base)</td>
</tr>
<tr>
<td>Max is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Max is neither a variable nor a number</td>
<td>type_error(number, Max)</td>
</tr>
<tr>
<td>Number is not a variable</td>
<td>type_error(variable, Number)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.26 File name processing

6.26.1 absolute_file_name/2

Templates

\[
\text{absolute_file_name}(\text{atom}, \text{atom})
\]

Description

\[
\text{absolute_file_name(File1, File2)} \text{ succeeds if File2 is the absolute pathname associated to the relative file name File1. File1 can contain \$VAR_NAME sub-strings. When such a sub-string is encountered, it is expanded with the value of the environment variable whose name is \$VAR_NAME if exists (otherwise no expansion is done). File1 can also begin with a sub-string ~USER_NAME/~, this is expanded as the home directory of the user USER_NAME. If USER_NAME does not exist File1 is an invalid pathname. If no USER_NAME is given (i.e. File1 begins with ~/) the ~ character is expanded as the value of the environment variable HOME. If the HOME variable is not defined File1 is an invalid pathname. Relative references to the current directory (/./ sub-string) and to the parent directory (/../ sub-strings) are removed and no longer appear in File2. File1 is also invalid if it contains too many /../ consecutive sub-strings (i.e. parent directory relative references). Finally if File1 is user then File2 is also unified with user to allow this predicate to be called on Prolog file names (since user in DEC-10 input/output predicates denotes the current input/output stream).}
\]
Most predicates using a file name implicitly call this predicate to obtain the desired file, e.g. open/4.

### Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>File1 is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>File1 is neither a variable nor an atom</td>
<td>type_error(atom, File1)</td>
</tr>
<tr>
<td>File2 is neither a variable nor an atom</td>
<td>type_error(atom, File2)</td>
</tr>
<tr>
<td>File1 is an atom but not a valid pathname</td>
<td>domain_error(os_path, File1)</td>
</tr>
</tbody>
</table>

### Portability

GNU Prolog predicate.

#### 6.26.2 `decompose_file_name/4`

### Templates


### Description

decompose_file_name(File, Directory, Prefix, Suffix) decomposes the pathname File and extracts the Directory part (characters before the last `/`), the Prefix part (characters after the last `/` and before the last `.`, or until the end if there is no suffix) and the Suffix part (characters from the last `. ` to the end of the string).

### Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>File is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>File is neither a variable nor an atom</td>
<td>type_error(atom, File)</td>
</tr>
<tr>
<td>Directory is neither a variable nor an atom</td>
<td>type_error(atom, Directory)</td>
</tr>
<tr>
<td>Prefix is neither a variable nor an atom</td>
<td>type_error(atom, Prefix)</td>
</tr>
<tr>
<td>Suffix is neither a variable nor an atom</td>
<td>type_error(atom, Suffix)</td>
</tr>
</tbody>
</table>

### Portability

GNU Prolog predicate.

#### 6.26.3 `prolog_file_name/2`

### Templates

`prolog_file_name(+atom, ?atom)`

### Description

prolog_file_name(File1, File2) unifies File2 with the Prolog file name associated to File1. More precisely File2 is computed as follows:

- if File1 has a suffix or if it is `user` then File2 is unified with File1.
- else if the file whose name is `File1 + '.pl'` exists then File2 is unified with this name.
- else if the file whose name is `File1 + '.pro'` exists then File2 is unified with this name.
- else File2 is unified with the name `File1 + '.pl'`. 
This predicate uses `absolute_file_name/2` to check the existence of a file (section 6.26.1, page 125).

**Errors**

<table>
<thead>
<tr>
<th>File1 is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>File1 is neither a variable nor an atom</td>
<td>type_error(atom, File1)</td>
</tr>
<tr>
<td>File2 is neither a variable nor an atom</td>
<td>type_error(atom, File2)</td>
</tr>
<tr>
<td>File1 is an atom but not a valid pathname</td>
<td>domain_error(os_path, File1)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

### 6.27 Operating system interface

#### 6.27.1 argument_counter/1

**Templates**

```
argument_counter(?integer)
```

**Description**

`argument_counter(Counter)` succeeds if `Counter` is the number of arguments of the command-line. Since the first argument is always the name of the running program, `Counter` is always ≥ 1.

**Errors**

| Counter is neither a variable nor an integer | type_error(integer, Counter) |

**Portability**

GNU Prolog predicate.

#### 6.27.2 argument_value/2

**Templates**

```
argument_value(+integer, ?atom)
```

**Description**

`argument_value(N, Arg)` succeeds if the Nth argument on the command-line unifies with `Arg`. The first argument is always the name of the running program and its number is 0. The number of arguments on the command-line can be obtained using `argument_counter/1` (section 6.27.1, page 127).

**Errors**

<table>
<thead>
<tr>
<th>N is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>N is neither a variable nor an integer</td>
<td>type_error(integer, N)</td>
</tr>
<tr>
<td>N is an integer &lt; 0</td>
<td>domain_error(not_less_than_zero, N)</td>
</tr>
<tr>
<td>Arg is neither a variable nor an atom</td>
<td>type_error(atom, Arg)</td>
</tr>
</tbody>
</table>

**Portability**
GNU Prolog predicate.

6.27.3  argument_list/1

Templates

argument_list(?list)

Description

argument_list(Args) succeeds if Args unifies with the list of atoms associated to each argument on the command-line other than the first argument (the name of the running program).

Errors

| Arg is neither a partial list nor a list | type_error(list, Args) |

Portability

GNU Prolog predicate.

6.27.4  environ/2

Templates

environ(?atom, ?atom)

Description

environ(Name, Value) succeeds if Name is the name of an environment variable whose value is Value. This predicate is re-executable on backtracking.

Errors

| Name is neither a variable nor an atom | type_error(atom, Name) |
| Value is neither a variable nor an atom | type_error(atom, Value) |

Portability

GNU Prolog predicate.

6.27.5  make_directory/1, delete_directory/1, change_directory/1

Templates

make_directory(+atom)
delete_directory(+atom)
change_directory(+atom)

Description

make_directory(PathName) creates the directory whose pathname is PathName.
delete_directory(PathName) removes the directory whose pathname is PathName.

change_directory(PathName) sets the current directory to the directory whose pathname is PathName.

See absolute_file_name/2 for information about the syntax of PathName (section 6.26.1, page 125).

Errors

<table>
<thead>
<tr>
<th>PathName is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathName is neither a variable nor an atom</td>
<td>type_error(atom, PathName)</td>
</tr>
<tr>
<td>PathName is an atom but not a valid pathname</td>
<td>domain_error(os.path, PathName)</td>
</tr>
<tr>
<td>an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>system_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.27.6 working_directory/1

Templates

    working_directory(?atom)

Description

working_directory(PathName) succeeds if PathName is the pathname of the current directory.

Errors

| PathName is neither a variable nor an atom | type_error(atom, PathName) |

Portability

GNU Prolog predicate.

6.27.7 directory_files/2

Templates

    directory_files(+atom, ?list)

Description

directory_files(PathName, Files) succeeds if Files is the list of all entries (files, sub-directories,...) in the directory whose pathname is PathName. See absolute_file_name/2 for information about the syntax of PathName (section 6.26.1, page 125).

Errors
### 6.27.8 rename_file/2

#### Templates

rename_file(+atom, +atom)

#### Description

rename_file(PathName1, PathName2) renames the file or directory whose pathname is PathName1 to PathName2. See absolute_filename/2 for information about the syntax of PathName1 and PathName2 (section 6.26.1, page 125).

#### Errors

<table>
<thead>
<tr>
<th>PathName1 is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathName1 is neither a variable nor an atom</td>
<td>type_error(atom, PathName1)</td>
</tr>
<tr>
<td>PathName1 is an atom but not a valid pathname</td>
<td>domain_error(os_path, PathName1)</td>
</tr>
<tr>
<td>PathName2 is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>PathName2 is neither a variable nor an atom</td>
<td>type_error(atom, PathName2)</td>
</tr>
<tr>
<td>PathName2 is an atom but not a valid pathname</td>
<td>domain_error(os_path, PathName2)</td>
</tr>
<tr>
<td>an operating system error occurs and value of the os_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>system_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

#### Portability

GNU Prolog predicate.

### 6.27.9 delete_file/1, unlink/1

#### Templates

delete_file(PathName)
unlink(PathName)

#### Description

delete_file(PathName) removes the existing file whose pathname is PathName.

unlink/1 is similar to delete_file/1 except that it never causes a system_error (e.g. if PathName does not refer to an existing file).
See `absolute_file_name/2` for information about the syntax of PathName (section 6.26.1, page 125).

**Errors**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathName is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>PathName is neither a variable nor an atom</td>
<td>type_error(atom, PathName)</td>
</tr>
<tr>
<td>PathName is an atom but not a valid pathname</td>
<td>domain_error(os_path, PathName)</td>
</tr>
<tr>
<td>an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>system_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicates.

### 6.27.10  file_permission/2, file_exists/1

**Templates**

- `file_permission(+atom, +atom)`
- `file_permission(+atom, +atom_list)`
- `file_exists(+atom)`

**Description**

`file_permission(PathName, Permission)` succeeds if PathName is the pathname of an existing file (or directory) whose permissions include Permission.

**File permissions:** Permission can be a single permission or a list of permissions. A permission is an atom among:

- **read:** the file or directory can be read.
- **write:** the file or directory can be written.
- **execute:** the file can be executed.
- **search:** the directory can be searched.

If PathName does not exist or if it its permissions do not include Permission this predicate fails.

`file_exists(PathName)` is equivalent to `file_permission(PathName, [])`, i.e. it succeeds if PathName is the pathname of an existing file (or directory).

See `absolute_file_name/2` for information about the syntax of PathName (section 6.26.1, page 125).

**Errors**
Portability

GNU Prolog predicates.

6.27.11 file_property/2

Templates

\[
\text{file_property}(\text{+atom, ?os_file_property})
\]

Description

file_property(PathName, Property) succeeds if PathName is the pathname of an existing file (or directory) and if Property unifies with one of the properties of the file. This predicate is re-executable on backtracking.

File properties:

- \text{absolute_file_name(File)}: File is the absolute file name of PathName (section 6.26.1, page 125).
- \text{real_file_name(File)}: File is the real file name of PathName (follows symbolic links).
- \text{type(Type)}: Type is the type of PathName. Possible values are: regular, directory, fifo, socket, character, device, block, device or unknown.
- \text{size(Size)}: Size is the size (in bytes) of PathName.
- \text{permission(Permission)}: Permission is a permission of PathName (section 6.27.10, page 131).
- \text{last_modification(DT)}: DT is the last modification date and time (section 6.27.14, page 134).

See \text{absolute_file_name/2} for information about the syntax of PathName (section 6.26.1, page 125).

Errors
PathName is a variable
PathName is neither a variable nor an atom
PathName is an atom but not a valid pathname
Property is neither a variable nor a file property term
Property = absolute_file_name(E), real_file_name(E), type(E) or permission(E) and E is neither a variable nor an atom
Property = last_modification(DateTime) and DateTime is neither a variable nor a compound term
Property = last_modification(DateTime) and DateTime is a compound term but not a structure dt/6
Property = size(E) or last_modification(DateTime) and DateTime is a structure dt/6 but an element E is neither a variable nor an integer
an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)

<table>
<thead>
<tr>
<th>PathName is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathName is neither a variable nor an atom</td>
<td>type_error(atom, PathName)</td>
</tr>
<tr>
<td>PathName is an atom but not a valid pathname</td>
<td>domain_error(os_path, PathName)</td>
</tr>
<tr>
<td>Property is neither a variable nor a file property term</td>
<td>domain_error(os_file_property, Property)</td>
</tr>
<tr>
<td>Property = absolute_file_name(E), real_file_name(E), type(E) or permission(E) and E is neither a variable nor an atom</td>
<td>type_error(atom, E)</td>
</tr>
<tr>
<td>Property = last_modification(DateTime) and DateTime is neither a variable nor a compound term</td>
<td>type_error(compound, DateTime)</td>
</tr>
<tr>
<td>Property = last_modification(DateTime) and DateTime is a compound term but not a structure dt/6</td>
<td>domain_error(date, time, DateTime)</td>
</tr>
<tr>
<td>Property = size(E) or last_modification(DateTime) and DateTime is a structure dt/6 but an element E is neither a variable nor an integer</td>
<td>type_error(integer, E)</td>
</tr>
<tr>
<td>an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>system_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

Portability
GNU Prolog predicate.

6.27.12 temporary_name/2

Templates
temporary_name(+atom, ?atom)

Description
temporary_name(Template, PathName) creates a unique file name PathName whose pathname begins by Template. Template should contain a path name with six trailing Xs. PathName is Template with the six Xs replaced with a letter and the process identifier. This predicate is an interface to the C Unix function mktemp(3).

See absolute_file_name/2 for information about the syntax of Template (section 6.26.1, page 125).

Errors

<table>
<thead>
<tr>
<th>Template is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template is neither a variable nor an atom</td>
<td>type_error(atom, Template)</td>
</tr>
<tr>
<td>Template is an atom but not a valid pathname</td>
<td>domain_error(os_path, Template)</td>
</tr>
<tr>
<td>PathName is neither a variable nor an atom</td>
<td>type_error(atom, PathName)</td>
</tr>
<tr>
<td>an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>system_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

Portability
GNU Prolog predicate.
6.27.13  temporary_file/3

Templates

\[ \text{temporary_file(+atom, +atom, ?atom)} \]

Description

temporary_file(Directory, Prefix, PathName) creates a unique file name PathName whose pathname begins by Directory/Prefix. If Directory is the empty atom \"\" a standard temporary directory will be used (e.g. /tmp). Prefix can be the empty atom \"\". This predicate is an interface to the C Unix function `tempnam(3)`.


Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>Directory is neither a variable nor an atom</td>
<td><code>type_error(atom, Directory)</code></td>
</tr>
<tr>
<td>Directory is an atom but not a valid pathname</td>
<td><code>domain_error(os_path, Directory)</code></td>
</tr>
<tr>
<td>Prefix is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>Prefix is neither a variable nor an atom</td>
<td><code>type_error(atom, Prefix)</code></td>
</tr>
<tr>
<td>PathName is neither a variable nor an atom</td>
<td><code>type_error(atom, PathName)</code></td>
</tr>
<tr>
<td>an operating system error occurs and the value of the os_error Prolog flag is error</td>
<td><code>system_error(atom explaining the error)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.27.14  date_time/1

Templates

\[ \text{date_time(?compound)} \]

Description

date_time(DateTime) unifies DateTime with a compound term containing the current date and time. DateTime is a structure \( dt(\text{Year}, \text{Month}, \text{Day}, \text{Hour}, \text{Minute}, \text{Second}) \). Each sub-argument of the term \( dt/6 \) is an integer.

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime is neither a variable nor a compound term</td>
<td><code>type_error(compound, DateTime)</code></td>
</tr>
<tr>
<td>DateTime is a compound term but not a structure ( dt/6 )</td>
<td><code>domain_error(date_time, DateTime)</code></td>
</tr>
<tr>
<td>DateTime is a structure ( dt/6 ) and an element ( E ) is not a variable nor an integer</td>
<td><code>type_error(integer, E)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.
6.27.15 host_name/1

Templates

host_name(?atom)

Description

host_name(HostName) unifies HostName with the name of the host machine executing the current GNU Prolog process. If the sockets are available (section 6.28.1, page 141), the name returned will be fully qualified. In that case, host_name/1 will also succeed if HostName is instantiated to the unqualified name (or an alias) of the machine.

Errors

<table>
<thead>
<tr>
<th>Hostname is neither a variable nor an atom an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</th>
<th>type_error(atom, HostName)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system_error(atom explaining the error)</td>
<td></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.27.16 os_version/1

Templates

os_version(?atom)

Description

os_version(OSVersion) unifies OSVersion with the operating system version of the machine executing the current GNU Prolog process.

Errors

<table>
<thead>
<tr>
<th>OSVersion is neither a variable nor an atom an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</th>
<th>type_error(atom, OSVersion)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system_error(atom explaining the error)</td>
<td></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.27.17 architecture/1

Templates

architecture(?atom)

Description
architecture(Architecture) unifies Architecture with the name of the machine executing the current GNU Prolog process.

Errors

| Architecture is neither a variable nor an atom | type_error(atom, Architecture) |
| an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117) | system_error(atom explaining the error) |

Portability

GNU Prolog predicate.

6.27.18 shell/2, shell/1, shell/0

Templates

shell(+atom, ?integer)

shell(+atom)

shell

Description

shell(Command, Status) invokes a new shell (named by the SHELL environment variable) passing Command for execution and unifies Status with the result of the execution. If Command is the empty atom ’ ’ a new interactive shell is executed. The control is returned to Prolog upon termination of the called process.

shell(Command) is equivalent to shell(Command, 0).

shell is equivalent to shell(’, 0).

Errors

| Command is a variable | instantiation_error |
| Command is neither a variable nor an atom | type_error(atom, Command) |
| Status is neither a variable nor an integer | type_error(integer, Status) |

Portability

GNU Prolog predicates.

6.27.19 system/2, system/1

Templates

system(+atom, ?integer)

system(+atom)

Description

system(Command, Status) invokes a new default shell passing Command for execution and unifies Status with the result of the execution. The control is returned to Prolog upon termination of the shell process. This predicate is an interface to the C Unix function system(3).
system(Command) is equivalent to system(Command, 0).

Errors

| Command is a variable          | instantiation_error       |
| Command is neither a variable nor an atom | type_error(atom, Command) |
| Status is neither a variable nor an integer | type_error(integer, Status) |

Portability

GNU Prolog predicates.

6.27.20 spawn/3, spawn/2

Templates

spawn(+atom, +atom_list, ?integer)
spawn(+atom, +atom_list)

Description

spawn(Command, Arguments, Status) executes Command passing as arguments of the command-line each element of the list Arguments and unifies Status with the result of the execution. The control is returned to Prolog upon termination of the command.

spawn(Command, Arguments) is equivalent to spawn(Command, Arguments, 0).

Errors

| Command is a variable          | instantiation_error       |
| Command is neither a variable nor an atom | type_error(atom, Command) |
| Arguments is a partial list or a list with an element which is a variable | instantiation_error       |
| Arguments is neither a partial list nor a list | type_error(list, Arguments) |
| an element E of the Arguments list is neither a variable nor an atom | type_error(atom, E) |
| Status is neither a variable nor an integer | type_error(integer, Status) |
| an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117) | system_error(atom explaining the error) |

Portability

GNU Prolog predicates.

6.27.21 popen/3

Templates

popen(+atom, +io_mode, -stream)

Description

popen(Command, Mode, Stream) invokes a new default shell (by creating a pipe) passing Command for execution and associates a stream either to the standard input or the standard output of the created
process. If \texttt{Mode} is \texttt{read} (resp. \texttt{write}) an input (resp. output) stream is created and \texttt{Stream} is unified with the stream-term associated. Writing to the stream writes to the standard input of the command while reading from the stream reads the command’s standard output. The stream must be closed using \texttt{close/2} (section 6.10.7, page 59). This predicate is an interface to the C Unix function \texttt{popen(3)}.

\begin{tabular}{|c|c|}
\hline
\textbf{Command} is a variable & \texttt{instantiation_error} \\
\textbf{Command} is neither a variable nor an atom & \texttt{type_error(atom, Command)} \\
\textbf{Mode} is a variable & \texttt{instantiation_error} \\
\textbf{Mode} is neither a variable nor an atom & \texttt{type_error(atom, Mode)} \\
\textbf{Stream} is not a variable & \texttt{domain_error(io_mode, Mode)} \\
\textbf{Stream} is not a variable & \texttt{type_error(variable, Stream)} \\
\textbf{an operating system error occurs and the value of the os} & \texttt{system_error(atom explaining the error)} \\
\textbf{error Prolog flag is error} & (section 6.22.1, page 117) \\
\hline
\end{tabular}

\section*{Portability}

GNU Prolog predicate.

\textbf{6.27.22} \texttt{exec/5, exec/4}

\section*{Templates}

\begin{verbatim}
exec(+atom, -stream, -stream, -stream, -integer) 
exec(+atom, -stream, -stream, -stream)
\end{verbatim}

\section*{Description}

\texttt{exec(Command, StreamIn, StreamOut, StreamErr, Pid)} invokes a new default shell passing \texttt{Command} for execution and associates streams to standard streams of the created process. \texttt{StreamIn} is unified with the stream-term associated to the standard input stream of \texttt{Command} (it is an output stream). \texttt{StreamOut} is unified with the stream-term associated to the standard output stream of \texttt{Command} (it is an input stream). \texttt{StreamErr} is unified with the stream-term associated to the standard error stream of \texttt{Command} (it is an input stream). \texttt{Pid} is unified with the process identifier of the new process. This information is only useful if it is necessary to obtain the status of the execution using \texttt{wait/2} (section 6.27.23, page 139). Until a call to \texttt{wait/2} is done the process remains in the system processes table (as a zombie process if terminated). For this reason, if the status is not needed it is preferable to use \texttt{exec/4}.

\texttt{exec/4} is similar to \texttt{exec/5} but the process is removed from system processes as soon as it is terminated.

\begin{tabular}{|c|c|}
\hline
\textbf{Command} is a variable & \texttt{instantiation_error} \\
\textbf{Command} is neither a variable nor an atom & \texttt{type_error(atom, Command)} \\
\textbf{StreamIn} is not a variable & \texttt{type_error(variable, StreamIn)} \\
\textbf{StreamOut} is not a variable & \texttt{type_error(variable, StreamOut)} \\
\textbf{StreamErr} is not a variable & \texttt{type_error(variable, StreamErr)} \\
\textbf{Pid} is not a variable & \texttt{type_error(variable, Pid)} \\
\textbf{an operating system error occurs and the value of the os} & \texttt{system_error(atom explaining the error)} \\
\textbf{error Prolog flag is error} & (section 6.22.1, page 117) \\
\hline
\end{tabular}

\section*{Portability}
GNU Prolog predicates.

**6.27.23 wait/2**

**Templates**

\[
\text{wait}(\text{+integer, ?integer})
\]

**Description**

\(\text{wait(Pid, Status)}\) waits for the child process whose identifier is Pid to terminate. Status is then unified with the exit status. This predicate is an interface to the C Unix function \text{waitpid}(2)\).

**Errors**

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pid is a variable</td>
<td>\text{instantiation_error}</td>
</tr>
<tr>
<td>Pid is neither a variable nor an integer</td>
<td>\text{type_error(integer, Pid)}</td>
</tr>
<tr>
<td>Status is neither a variable nor an integer</td>
<td>\text{type_error(integer, Status)}</td>
</tr>
<tr>
<td>An operating system error occurs and the value of the \text{os_error} Prolog flag is error</td>
<td>\text{system_error(\text{atom explaining the error})}</td>
</tr>
<tr>
<td>(\text{section 6.22.1, page 117})</td>
<td></td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

**6.27.24 prolog\_pid/1**

**Templates**

\[
\text{prolog\_pid(?integer)}
\]

**Description**

\text{prolog\_pid(Pid)} unifies Pid with the process identifier of the current GNU Prolog process.

**Errors**

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pid is neither a variable nor an integer</td>
<td>\text{type_error(integer, Pid)}</td>
</tr>
</tbody>
</table>

**Portability**

GNU Prolog predicate.

**6.27.25 send\_signal/2**

**Templates**

\[
\text{send\_signal(+integer, +integer)}
\]

\[
\text{send\_signal(+integer, +atom)}
\]

**Description**
send_signal(Pid, Signal) sends Signal to the process whose identifier is Pid. Signal can be specified directly as an integer or symbolically as an atom. Allowed atoms depend on the machine (e.g. 'SIGINT', 'SIGQUIT', 'SIGKILL', 'SIGUSR1', 'SIGUSR2', 'SIGALRM', ...). This predicate is an interface to the C Unix function kill(2).

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pid is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Pid is neither a variable nor an integer</td>
<td>type_error(integer, Pid)</td>
</tr>
<tr>
<td>Signal is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Signal is neither a variable nor an integer or an atom</td>
<td>type_error(integer, Signal)</td>
</tr>
<tr>
<td>an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>system_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.27.26  sleep/1

Templates

sleep(+number)

Description

sleep(Seconds) puts the GNU Prolog process to sleep for Seconds seconds. Seconds can be an integer or a floating point number (in which case it can be < 1). This predicate is an interface to the C Unix function usleep(3).

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Seconds is neither a variable nor a number</td>
<td>type_error(number, Seconds)</td>
</tr>
<tr>
<td>Seconds is a number &lt; 0</td>
<td>domain_error(not_less_than_zero, Seconds)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.27.27  select/5

Templates

select(+list, ?list, +list, ?list, +number)

Description

select(Reads, ReadyReads, Writes, ReadyWrites, TimeOut) waits for a number of streams (or file descriptors) to change status. ReadyReads is unified with the list of elements listed in Reads that have characters available for reading. Similarly ReadyWrites is unified with the list of elements of Writes that are ok for immediate writing. The elements of Reads and Writes are either stream-terms
or aliases or integers considered as file descriptors, e.g. for sockets (section 6.28, page 141). Streams that must be tested with select/5 should not be buffered. This can be done at the opening using open/4 (section 6.10.6, page 58) or later using set_stream_buffering/2 (section 6.10.24, page 68). TimeOut is an upper bound on the amount of time (in milliseconds) elapsed before select/5 returns. If TimeOut ≤ 0 (no timeout) select/5 waits until something is available (either reading or writing) and thus can block indefinitely. This predicate is an interface to the C Unix function select(2).

Errors

| Reads (or Writes) is a partial list or a list with an element E which is a variable | instantiation_error |
| Reads is neither a partial list nor a list | type_error(list, Reads) |
| Writes is neither a partial list nor a list | type_error(list, Writes) |
| ReadyReads is neither a partial list nor a list | type_error(list, ReadyReads) |
| ReadyWrites is neither a partial list nor a list | type_error(list, ReadyWrites) |
| an element E of the Reads (or Writes) list is not a selectable item | domain_error(selectable_item, E) |
| an element E of the Reads (or Writes) list is not an integer | domain_error(not_less_than_zero, E) |
| an element E of the Reads (or Writes) list is a stream-term or alias not associated with an open stream | domain_error(stream_or_alias, E) |
| an element E of the Reads list is associated to an output stream | domain_error(output_stream, E) |
| an element E of the Writes list is associated to an input stream | domain_error(input_stream, E) |
| TimeOut is a variable | instantiation_error |
| TimeOut is neither a variable nor a number | type_error(number, TimeOut) |
| an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117) | system_error(atom explaining the error) |

Portability

GNU Prolog predicate.

6.28 Sockets input/output

6.28.1 Introduction

This set of predicates provides a way to manipulate sockets. The predicates are straightforward interfaces to the corresponding BSD-type socket functions. This facility is available if the sockets part of GNU Prolog has been installed. A reader familiar with BSD sockets will understand them immediately otherwise a study of sockets is needed.

The domain is either the atom 'AF_INET' or 'AF_UNIX' corresponding to the same domains in BSD-type sockets.

An address is either of the form 'AF_INET'(HostName, Port) or 'AF_UNIX'(SocketName). HostName is an atom denoting a machine name, Port is a port number and SocketName is an atom denoting a socket.

By default, streams associated to sockets are block buffered. The predicate set_stream_buffering/2
(section 6.10.24, page 68) can be used to change this mode. They are also text streams by default. Use `set_stream_type/2` (section 6.10.22, page 67) to change the type if binary streams are needed.

### 6.28.2 socket/2

#### Templates

```
socket(+socket_domain, -integer)
```

#### Description

`socket(Domain, Socket)` creates a socket whose domain is `Domain` (section 6.28, page 141) and unifies `Socket` with the descriptor identifying the socket. This predicate is an interface to the C Unix function `socket(2)`.

#### Errors

<table>
<thead>
<tr>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain is a variable</td>
</tr>
<tr>
<td>Domain is neither a variable nor an atom</td>
</tr>
<tr>
<td>Domain is an atom but not a valid socket domain</td>
</tr>
<tr>
<td>Socket is not a variable</td>
</tr>
<tr>
<td>an operating system error occurs and the value of the <code>os_error</code> Prolog flag is <code>error</code></td>
</tr>
<tr>
<td>(section 6.22.1, page 117)</td>
</tr>
<tr>
<td>instantiation_error</td>
</tr>
<tr>
<td><code>type_error(atom, Domain)</code></td>
</tr>
<tr>
<td><code>domain_error(socket, Domain)</code></td>
</tr>
<tr>
<td><code>type_error(variable, Socket)</code></td>
</tr>
<tr>
<td><code>system_error(atom explaining the error)</code></td>
</tr>
</tbody>
</table>

#### Portability

GNU Prolog predicate.

### 6.28.3 socket_close/1

#### Templates

```
socket_close(+integer)
```

#### Description

`socket_close(Socket)` closes the socket whose descriptor is `Socket`. This predicate should not be used if `Socket` has given rise to a stream, e.g. by `socket_connect/2` (section 6.28.5, page 143). In that case simply use `close/2` (section 6.10.7, page 59) on the associated stream.

#### Errors

<table>
<thead>
<tr>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket is a variable</td>
</tr>
<tr>
<td>Socket is neither a variable nor an integer</td>
</tr>
<tr>
<td>an operating system error occurs and the value of the <code>os_error</code> Prolog flag is <code>error</code></td>
</tr>
<tr>
<td>(section 6.22.1, page 117)</td>
</tr>
<tr>
<td>instantiation_error</td>
</tr>
<tr>
<td><code>type_error(integer, Socket)</code></td>
</tr>
<tr>
<td><code>system_error(atom explaining the error)</code></td>
</tr>
</tbody>
</table>

#### Portability

GNU Prolog predicate.
6.28.4  socket_bind/2

Templates

socket_bind(+integer, +socket_address)

Description

socket_bind(Socket, Address) binds the socket whose descriptor is Socket to the address specified by Address (section 6.28, page 141). If Address is of the form `AF_INET'(HostName, Port) and if HostName is uninstantiated then it is unified with the current machine name. If Port is uninstantiated, it is unified to a port number picked by the operating system. This predicate is an interface to the C Unix function bind(2).

Errors

<table>
<thead>
<tr>
<th>Socket is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket is neither a variable nor an integer</td>
<td>type_error(integer, Socket)</td>
</tr>
<tr>
<td>Address is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Address is neither a variable nor a valid address</td>
<td>domain_error(socket_address, Address)</td>
</tr>
<tr>
<td>Address = 'AF_UNIX'(E) and E is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Address = 'AF_UNIX'(E) or 'AF_INET'(E, _) and E is neither a variable nor an atom</td>
<td>type_error(atom, E)</td>
</tr>
<tr>
<td>Address = 'AF_UNIX'(E) and E is an atom but not a valid pathname</td>
<td>domain_error(os_path, E)</td>
</tr>
<tr>
<td>Address = 'AF_INET'(_, E) and E is neither a variable nor an integer</td>
<td>type_error(integer, E)</td>
</tr>
</tbody>
</table>

an operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117) | system_error(atom explaining the error) |

Portability

GNU Prolog predicate.

6.28.5  socket_connect/4

Templates

socket_connect(+integer, +socket_address, -stream, -stream)

Description

socket_connect(Socket, Address, StreamIn, StreamOut) connects the socket whose descriptor is Socket to the address specified by Address (section 6.28, page 141). StreamIn is unified with a stream-term associated to the input of the connection (it is an input stream). Reading from this stream gets data from the socket. StreamOut is unified with a stream-term associated to the output of the connection (it is an output stream). Writing to this stream sends data to the socket. The use of select/5 can be useful (section 6.27.27, page 140). This predicate is an interface to the C Unix function connect(2).

Errors
Portability

GNU Prolog predicate.

6.28.6 socket.listen/2

Templates

socket.listen(+integer, +integer)

Description

socket.listen(Socket, Length) defines the socket whose descriptor is Socket to have a maximum backlog queue of Length pending connections. This predicate is an interface to the C Unix function listen(2).

Errors

| Socket is a variable | instantiation_error |
| Socket is neither a variable nor an integer | type_error(integer, Socket) |
| Address is a variable | instantiation_error |
| Address is neither a variable nor a valid address | domain_error(socket_address, Address) |
| Address = 'AF_UNIX'(E) or 'AF_INET'(E, _) or Address = 'AF_INET'(_, E) and E is a variable | instantiation_error |
| Address = 'AF_UNIX'(E) or 'AF_INET'(E, _) and E is neither a variable nor an atom | type_error(atom, E) |
| Address = 'AF_UNIX'(E) and E is an atom but not a valid pathname | domain_error(os.path, E) |
| Address = 'AF_INET'(_, E) and E is neither a variable nor an integer | type_error(integer, E) |
| StreamIn is not a variable | type_error(variable, StreamIn) |
| StreamOut is not a variable | type_error(variable, StreamOut) |
| an operating system error occurs and the value of the os.error Prolog flag is error (section 6.22.1, page 117) | system_error(atom explaining the error) |

Portability

GNU Prolog predicate.

6.28.7 socket.accept/4, socket.accept/3

Templates

socket.accept(+integer, -atom, -stream, -stream)
socket.accept(+integer, -stream, -stream)
Description

socket_accept(Socket, Client, StreamIn, StreamOut) extracts the first connection to the socket whose descriptor is Socket. If the domain is ‘AF_INET’, Client is unified with an atom whose name is the Internet host address in numbers-and-dots notation of the connecting machine. StreamIn is unified with a stream-term associated to the input of the connection (it is an input stream). Reading from this stream gets data from the socket. StreamOut is unified with a stream-term associated to the output of the connection (it is an output stream). Writing to this stream sends data to the socket. The use of select/5 can be useful (section 6.27.27, page 140). This predicate is an interface to the C Unix function accept(2).

socket_accept(Socket, StreamIn, StreamOut) is equivalent to socket_accept(Socket, _, StreamIn, StreamOut).

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>Socket is neither a variable nor an integer</td>
<td>type_error(integer, Socket)</td>
</tr>
<tr>
<td>Client is not a variable</td>
<td>type_error(variable, Client)</td>
</tr>
<tr>
<td>StreamIn is not a variable</td>
<td>type_error(variable, StreamIn)</td>
</tr>
<tr>
<td>StreamOut is not a variable</td>
<td>type_error(variable, StreamOut)</td>
</tr>
<tr>
<td>An operating system error occurs and the value of the os_error Prolog flag is error (section 6.22.1, page 117)</td>
<td>system_error(atom explaining the error)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

6.28.8 hostname_address/2

Templates

hostname_address(+atom, ?atom)
hostname_address(?atom, +atom)

Description

hostname_address(HostName, HostAddress) succeeds if the Internet host address in numbers-and-dots notation of HostName is HostAddress. Hostname can be given as a fully qualified name, or an unqualified name or an alias of the machine. The predicate will fail if the machine name or address cannot be resolved.

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HostName and HostAddress are variables</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>HostName is neither a variable nor an atom</td>
<td>type_error(atom, HostName)</td>
</tr>
<tr>
<td>HostAddress is neither a variable nor an atom</td>
<td>type_error(atom, HostAddress)</td>
</tr>
<tr>
<td>Address is neither a variable nor a valid address</td>
<td>domain_error(socket_address, Address)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.
6.29 Lineedit management

The following predicates are only available if the lineedit part of GNU Prolog has been installed.

6.29.1 get_lineedit_prompt/1

Templates

```prolog
get_lineedit_prompt(?atom)
```

Description

`get_lineedit_prompt(Prompt)` succeeds if `Prompt` is the current lineedit prompt, e.g. the top-level prompt is `'/ ';'`. By default all other reads have an empty prompt.

Errors

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prompt is neither a variable nor an atom</td>
<td><code>type_error(atom, Pred)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.29.2 set_lineedit_prompt/1

Templates

```prolog
set_lineedit_prompt(+atom)
```

Description

`set_lineedit_prompt(Prompt)` sets the current lineedit prompt to `Prompt`. This prompt will be displayed for reads from a terminal (except for top-level reads).

Errors

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prompt is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>Prompt is neither a variable nor an atom</td>
<td><code>type_error(atom, Pred)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.29.3 add_lineedit_completion/1

Templates

```prolog
add_lineedit_completion(+atom)
```

Description

`add_lineedit_completion(Word)` adds `Word` in the list of completion words maintained by `lineedit` (section 2.2.5, page 8). Only words containing letters, digits and the underscore character are added (if `Word` does not respect this restriction the predicate fails).
6.29 Linedit management

Errors

<table>
<thead>
<tr>
<th>Word is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word is neither a variable nor an atom</td>
<td>type_error(atom, Word)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

6.29.4 find_linedit_completion/2

Templates

find_linedit_completion(+atom, ?atom)

Description

find_linedit_completion(Prefix, Word) succeeds if Word is a word beginning by Prefix and belongs to the list of completion words maintained by linedit (section 2.2.5, page 8). This predicate is re-executable on backtracking.

Errors

<table>
<thead>
<tr>
<th>Prefix is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix is neither a variable nor an atom</td>
<td>type_error(atom, Prefix)</td>
</tr>
<tr>
<td>Word is neither a variable nor an atom</td>
<td>type_error(atom, Word)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.
7 Finite domain solver and built-in predicates

7.1 Introduction

The finite domain (FD) constraint solver extends Prolog with constraints over FD. This facility is available if the FD part of GNU Prolog has been installed. The solver is an instance of the Constraint Logic Programming scheme introduced by Jaffar and Lassez in 1987 [6]. Constraints on FD are solved using propagation techniques, in particular arc-consistency (AC). The interested reader can refer to “Constraint Satisfaction in Logic Programming” of P. Van Hentenryck (1989) [7]. The solver is based on the clp(FD) solver [4]. The GNU Prolog FD solver offers arithmetic constraints, boolean constraints, reified constraints and symbolic constraints on a new kind of variables: Finite Domain variables.

7.1.1 Finite Domain variables

A new type of data is introduced: FD variables which can only take values in their domains. The initial domain of an FD variable is 0..fd_max_integer where fd_max_integer represents the greatest value that any FD variable can take. The predicate fd_max_integer/1 returns this value which may be different from the max_integer Prolog flag (section 6.22.1, page 117). The domain of an FD variable X is reduced step by step by constraints in a monotonic way: when a value has been removed from the domain of X it will never reappear in the domain of X. An FD variable is fully compatible with both Prolog integers and Prolog variables. Namely, when an FD variable is expected by an FD constraint it is possible to pass a Prolog integer (considered as an FD variable whose domain is a singleton) or a Prolog variable (immediately bound to an initial range 0..fd_max_integer). This avoids the need for specific type declaration. Although it is not necessary to declare the initial domain of an FD variable (since it will be bound 0..fd_max_integer when appearing for the first time in a constraint) it is advantageous to do so and thus reduce as soon as possible the size of its domain: particularly because GNU Prolog, for efficiency reasons, does not check for overflows. For instance, without any preliminary domain definitions for X, Y and Z, the non-linear constraint X*Y#Z will fail due to an overflow when computing the upper bound of the domain of Z: fd_max_integer * fd_max_integer. This overflow causes a negative result for the upper bound and the constraint then fails.

There are two internal representations for an FD variable:

- **interval representation**: only the min and the max of the variable are maintained. In this representation it is possible to store values included in 0..fd_max_integer.

- **sparse representation**: an additional bit-vector is used to store the set of possible values for the variable (i.e. the domain). In this representation it is possible to store values included in 0..vector_max. By default vector_max is set to 127. This value can be redefined via an environment variable VECTORMAX or via the built-in predicate fd_set_vector_max/1 (section 7.2.3, page 150). The predicate fd_vector_max/1 returns the current value of vector_max (section 7.2.1, page 149).

The initial representation for an FD variable X is always an interval representation and is switched to a sparse representation when a “hole” appears in the domain (e.g. due to an inequality constraint). Once a variable uses a sparse representation it will not switch back to an interval representation even if there are no longer holes in its domain. When this switching occurs some values in the domain of X can be lost since vector_max is less than fd_max_integer. We say that “X is extra-constrained” since X is constrained by the solver to the domain 0..vector_max (via an imaginary constraint X #=< vector_max).

An extra_cstr is associated to each FD variable to indicate that values have been lost due to the switch to a sparse representation. This flag is updated on every operations. The domain of an extra-constrained FD variable is output followed by the \( \Theta \) symbol. When a constraint fails on a extra-constrained variable a message Warning: Vector too small - maybe lost solutions (FD Var:N) is displayed (N is the address of the involved variable).
7.2 FD variable parameters

Example 1 (vector\_max = 127):

<table>
<thead>
<tr>
<th>Constraint on X</th>
<th>Domain of X</th>
<th>extra_cstr</th>
<th>Lost values</th>
</tr>
</thead>
<tbody>
<tr>
<td>X #&lt; 512</td>
<td>0..512</td>
<td>off</td>
<td>none</td>
</tr>
<tr>
<td>X #\leq 10</td>
<td>0..9:11..127</td>
<td>on</td>
<td>128..512</td>
</tr>
<tr>
<td>X #&lt; 100</td>
<td>0..9:11..100</td>
<td>off</td>
<td>none</td>
</tr>
</tbody>
</table>

In this example, when the constraint X \#\leq 10 is posted some values are lost, the extra\_cstr is then switched on. However, posting the constraint X \#< 100 will turn off the flag (no values are lost).

Example 2:

<table>
<thead>
<tr>
<th>Constraint on X</th>
<th>Domain of X</th>
<th>extra_cstr</th>
<th>Lost values</th>
</tr>
</thead>
<tbody>
<tr>
<td>X #&lt; 512</td>
<td>0..512</td>
<td>off</td>
<td>none</td>
</tr>
<tr>
<td>X #&lt;= 10</td>
<td>0..9:11..127</td>
<td>on</td>
<td>128..512</td>
</tr>
<tr>
<td>X #&gt; 256</td>
<td>Warning: Vector too small...</td>
<td>on</td>
<td>128..512</td>
</tr>
</tbody>
</table>

In this example, the constraint X \#> 256 fails due to the lost of 128..512 so a message is displayed onto the terminal. The solution would consist in increasing the size of the vector either by setting the environment variable VECTORMAX (e.g. to 512) or using fd\_set\_vector\_max(512).

Finally, bit-vectors are not dynamic, i.e. all vectors have the same size (0..vector\_max). So the use of fd\_set\_vector\_max/1 is limited to the initial definition of vector sizes and must occur before any constraint. As seen before, the solver tries to display a message when a failure occurs due to a too short vector\_max. Unfortunately, in some cases it cannot detect the lost of values and no message is emitted. So the user should always take care to this parameter to be sure that it is large to encode any vector.

7.2 FD variable parameters

7.2.1 fd\_max\_integer/1

Templates

\texttt{fd\_max\_integer(?integer)}

Description

\texttt{fd\_max\_integer(N)} succeeds if N is the current value of \texttt{fd\_max\_integer} (section 7.1, page 148).

Errors

\texttt{N is neither a variable nor an integer} \texttt{type\_error(integer, N)}

Portability

GNU Prolog predicate.

7.2.2 fd\_vector\_max/1

Templates

\texttt{fd\_vector\_max(?integer)}
Description

\texttt{fd\_vector\_max(N)} succeeds if \(N\) is the current value of \texttt{vector\_max} (section 7.1, page 148).

Errors

\begin{tabular}{|l|l|}
\hline
\texttt{N} is neither a variable nor an integer & \texttt{type\_error(integer, N)} \\
\hline
\end{tabular}

Portability

GNU Prolog predicate.

7.2.3 \texttt{fd\_set\_vector\_max/1}

Templates

\begin{verbatim}
fd_set_vector_max(+integer)
\end{verbatim}

Description

\texttt{fd\_set\_vector\_max(N)} initializes \texttt{vector\_max} based on the value \(N\) (section 7.1, page 148). More precisely, on 32 bit machines, \texttt{vector\_max} is set to the smallest value of \((32\times k)-1\) which is \(\geq N\).

Errors

\begin{tabular}{|l|l|}
\hline
\texttt{N} is a variable & \texttt{instantiation\_error} \\
\texttt{N} is neither a variable nor an integer & \texttt{type\_error(integer, N)} \\
\texttt{N} is an integer < 0 & \texttt{domain\_error(not\_less\_than\_zero, N)} \\
\hline
\end{tabular}

Portability

GNU Prolog predicate.

7.3 Initial value constraints

7.3.1 \texttt{fd\_domain/3, fd\_domain\_bool/1}

Templates

\begin{verbatim}
fd\_domain(+fd\_variable\_list\_or\_fd\_variable, +integer, +integer)  
fd\_domain(?fd\_variable, +integer, +integer)  
fd\_domain\_bool(+fd\_variable\_list)  
fd\_domain\_bool(?fd\_variable)
\end{verbatim}

Description

\texttt{fd\_domain(Vars, Lower, Upper)} constraints each element \(X\) of \texttt{Vars} to take a value in \texttt{Lower .. Upper}. This predicate is generally used to set the initial domain of variables to an interval. \texttt{Vars} can be also a single FD variable (or a single Prolog variable).

\texttt{fd\_domain\_bool(Vars)} is equivalent to \texttt{fd\_domain(Vars, 0, 1)} and is used to declare boolean FD variables.

Errors
### 7.4 Type testing

#### 7.4.1 fd_var/1, non_fd_var/1, generic_var/1, non_generic_var/1

**Templates**

```prolog
fd_var(+fd_var), +integer_list)
fd_domain(?fd_var, +integer_list)
```

**Description**

`fd_domain(Vars, Values)` constraints each element X of the list `Vars` to take a value in the list `Values`. This predicate is generally used to set the initial domain of variables to a set of values. The domain of each variable of `Vars` uses a sparse representation. `Vars` can be also a single FD variable (or a single Prolog variable).

**Errors**

| Vars is not a variable but is a partial list                      | instantiation_error  |
| Vars is neither a variable nor an FD variable nor an integer nor a list | type_error(list, Vars) |
| an element E of the Vars list is neither a variable nor an FD variable nor an integer | type_error(fd_variable, E) |
| Values is a partial list or a list with an element E which is a variable | instantiation_error |
| Values is neither a partial list nor a list | type_error(list, Values) |
| an element E of the Values list is neither a variable nor an integer | type_error(integer, E) |

**Portability**

GNU Prolog predicate.

#### 7.3.2 fd_domain/2

**Templates**

```prolog
fd_domain(+fd_variable_list, +integer_list)
fd_domain(?fd_variable, +integer_list)
```

**Description**

`fd_domain(Vars, Values)` constraints each element X of the list `Vars` to take a value in the list `Values`. This predicate is generally used to set the initial domain of variables to a set of values. The domain of each variable of `Vars` uses a sparse representation. `Vars` can be also a single FD variable (or a single Prolog variable).

**Errors**

| Vars is not a variable but is a partial list                      | instantiation_error  |
| Vars is neither a variable nor an FD variable nor an integer nor a list | type_error(list, Vars) |
| an element E of the Vars list is neither a variable nor an FD variable nor an integer | type_error(fd_variable, E) |
| Values is a partial list or a list with an element E which is a variable | instantiation_error |
| Values is neither a partial list nor a list | type_error(list, Values) |
| an element E of the Values list is neither a variable nor an integer | type_error(integer, E) |

**Portability**

GNU Prolog predicate.
fd_var(?term)  generic_var(?term)
non_fd_var(?term)  non_generic_var(?term)

Description

fd_var(Term) succeeds if Term is currently an FD variable.

non_fd_var(Term) succeeds if Term is currently not an FD variable (opposite of fd_var/1).

generic_var(Term) succeeds if Term is either a Prolog variable or an FD variable.

non_generic_var(Term) succeeds if Term is neither a Prolog variable nor an FD variable (opposite of generic_var/1).

Errors

None.

Portability

GNU Prolog predicate.

7.5 FD variable information

These predicate allow the user to get some information about FD variables. They are not constraints, they only return the current state of a variable.

7.5.1 fd_min/2, fd_max/2, fd_size/2, fd_dom/2

Templates

fd_min(+fd_variable, ?integer)
fd_max(+fd_variable, ?integer)
fd_size(+fd_variable, ?integer)
fd_dom(+fd_variable, ?integer_list)

Description

fd_min(X, N) succeeds if N is the minimal value of the current domain of X.

fd_max(X, N) succeeds if N is the maximal value of the current domain of X.

fd_size(X, N) succeeds if N is the number of elements of the current domain of X.

fd_dom(X, Values) succeeds if Values is the list of values of the current domain of X.

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>X is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>X is neither an FD variable nor an integer</td>
<td>type_error(fd_variable, X)</td>
</tr>
<tr>
<td>N is neither a variable nor an integer</td>
<td>type_error(integer, N)</td>
</tr>
<tr>
<td>an element E of the Vars list is neither a variable nor</td>
<td>type_error(fd_variable, E)</td>
</tr>
<tr>
<td>an FD variable nor an integer</td>
<td></td>
</tr>
<tr>
<td>Values is neither a partial list nor a list</td>
<td>type_error(list, Values)</td>
</tr>
</tbody>
</table>
7.6 Arithmetic constraints

Portability

GNU Prolog predicate.

7.5.2 fd_has_extra_cstr/1, fd_has_vector/1, fd_use_vector/1

Templates

\[
\begin{align*}
&\text{fd\_has\_extra\_cstr(+fd\_variable)} \\
&\text{fd\_has\_vector(+fd\_variable)} \\
&\text{fd\_use\_vector(+fd\_variable)}
\end{align*}
\]

Description

fd_has_extra_cstr(X) succeeds if the extra_cstr of X is currently on (section 7.1, page 148).

fd_has_vector(X) succeeds if the current domain of X uses a sparse representation (section 7.1, page 148).

fd_use_vector(X) enforces a sparse representation for the domain of X (section 7.1, page 148).

Errors

<table>
<thead>
<tr>
<th>Condition</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>X is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>X is neither an FD variable nor an integer</td>
<td>type_error(fd_variable, X)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

7.6 Arithmetic constraints

7.6.1 FD arithmetic expressions

An FD arithmetic expression is a Prolog term built from integers, variables (Prolog or FD variables), and functionals (or operators) that represent arithmetic functions. The following table details the components of an FD arithmetic expression:
FD expressions are not restricted to be linear. However non-linear constraints usually yield less constraint propagation than linear constraints.

*, –, *, /, //, rem and ** are predefined infix operators. + and – are predefined prefix operators (section 6.14.10, page 87).

Errors

<table>
<thead>
<tr>
<th>a sub-expression is of the form _ ** E and E is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>a sub-expression E is neither a variable nor an integer nor an FD arithmetic functor</td>
<td>type_error(fd_evaluable, E)</td>
</tr>
<tr>
<td>an expression is too complex</td>
<td>resource_error(too_big_fd_constraint)</td>
</tr>
</tbody>
</table>

7.6.2 Partial AC: (#=)/2 - constraint equal, (#\=/2 - constraint not equal,
(#<)/2 - constraint less than, (#=<)/2 - constraint less than or equal,
(#>/2 - constraint greater than, (#=>)/2 - constraint greater than or equal

Templates

#=(?fd_evaluable, ?fd_evaluable)
#\!=(?fd_evaluable, ?fd_evaluable)
#<(?fd_evaluable, ?fd_evaluable)
#=<(?fd_evaluable, ?fd_evaluable)
#>(?fd_evaluable, ?fd_evaluable)
#>=(?fd_evaluable, ?fd_evaluable)

Description

FdExpr1 #= FdExpr2 constrains FdExpr1 to be equal to FdExpr2.

FdExpr1 #\= FdExpr2 constrains FdExpr1 to be different from FdExpr2.

FdExpr1 #< FdExpr2 constrains FdExpr1 to be less than FdExpr2.

FdExpr1 #=< FdExpr2 constrains FdExpr1 to be less than or equal to FdExpr2.
7.6 Arithmetic constraints

FdExpr1 => FdExpr2 constrains FdExpr1 to be greater than FdExpr2.

FdExpr1 >= FdExpr2 constrains FdExpr1 to be greater than or equal to FdExpr2.

FdExpr1 and FdExpr2 are arithmetic FD expressions (section 7.6.1, page 153).

#, #\=, #<, #=<, #> and #>= are predefined infix operators (section 6.14.10, page 87).

These predicates post arithmetic constraints that are managed by the solver using a partial arc-consistency algorithm to reduce the domain of involved variables. In this scheme only the bounds of the domain of variables are updated. This leads to less propagation than full arc-consistency techniques (section 7.6.3, page 155) but is generally more efficient for arithmetic. These arithmetic constraints can be reified (section 7.7, page 156).

Errors

Refer to the syntax of arithmetic FD expressions for possible errors (section 7.6.1, page 153).

Portability

GNU Prolog predicates.

Full AC: (#=#)/2 - constraint equal, (#\=#)/2 - constraint not equal, (#<#)/2 - constraint less than, (#=#)/2 - constraint less than or equal, (#>#)/2 - constraint greater than, (#>#)/2 - constraint greater than or equal

Templates

#=#(fd_evaluable, ?fd_evaluable)
#\=#(fd_evaluable, ?fd_evaluable)
#<#(fd_evaluable, ?fd_evaluable)
#<=#(fd_evaluable, ?fd_evaluable)
#>#(fd_evaluable, ?fd_evaluable)
#>=#(fd_evaluable, ?fd_evaluable)

Description

FdExpr1 == FdExpr2 constrains FdExpr1 to be equal to FdExpr2.

FdExpr1 \=# FdExpr2 constrains FdExpr1 to be different from FdExpr2.

FdExpr1 <= FdExpr2 constrains FdExpr1 to be less than FdExpr2.

FdExpr1 < FdExpr2 constrains FdExpr1 to be less than or equal to FdExpr2.

FdExpr1 >= FdExpr2 constrains FdExpr1 to be greater than FdExpr2.

FdExpr1 > FdExpr2 constrains FdExpr1 to be greater than or equal to FdExpr2.

FdExpr1 and FdExpr2 are arithmetic FD expressions (section 7.6.1, page 153).

#, #\=, #<, #=<, #> and #>= are predefined infix operators (section 6.14.10, page 87).

These predicates post arithmetic constraints that are managed by the solver using a full arc-consistency algorithm to reduce the domain of involved variables. In this scheme the full domain of variables is updated. This leads to more propagation than partial arc-consistency techniques (section 7.6.1, page 153).
but is generally less efficient for arithmetic. These arithmetic constraints can be reified (section 7.7.1, page 156).

Errors
Refer to the syntax of arithmetic FD expressions for possible errors (section 7.6.1, page 153).

Portability
GNU Prolog predicates.

7.6.4 fd_prime/1, fd_not_prime/1

Templates

\[
\begin{align*}
\text{fd\_prime}(\text{X}) \\
\text{fd\_not\_prime}(\text{X})
\end{align*}
\]

Description

\(\text{fd\_prime}(X)\) constraints \(X\) to be a prime number between \(0..\text{vector\_max}\). This constraint enforces a sparse representation for the domain of \(X\) (section 7.1, page 148).

\(\text{fd\_not\_prime}(X)\) constraints \(X\) to be a non prime number between \(0..\text{vector\_max}\). This constraint enforces a sparse representation for the domain of \(X\) (section 7.1, page 148).

Errors

\[
\begin{array}{|l|}
\hline
\text{X is neither an FD variable nor an integer} & \text{type\_error(fd\_variable, X)} \\
\hline
\end{array}
\]

Portability
GNU Prolog predicates.

7.7 Boolean and reified constraints

7.7.1 Boolean FD expressions

An boolean FD expression is a Prolog term built from integers (0 for false, 1 for true), variables (Prolog or FD variables), partial AC arithmetic constraints (section 7.6.2, page 154), full AC arithmetic constraints (section 7.6.3, page 155) and functors (or operators) that represent boolean functions. When a sub-expression of a boolean expression is an arithmetic constraint \(c\), it is reified. Namely, as soon as the solver detects that \(c\) is true (i.e. entailed) the sub-expression has the value 1. Similarly when the solver detects that \(c\) is false (i.e. disentailed) the sub-expression evaluates as 0. While neither the entailment nor the disentailment can be detected the sub-expression is evaluated as a domain \(0..1\). The following table details the components of an FD boolean expression:
7.7 Boolean and reified constraints

<table>
<thead>
<tr>
<th>FD Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prolog variable</td>
<td>domain 0..1</td>
</tr>
<tr>
<td>FD variable X</td>
<td>domain of X, X is constrained to be in 0..1</td>
</tr>
<tr>
<td>0 (integer)</td>
<td>0 (false)</td>
</tr>
<tr>
<td>1 (integer)</td>
<td>1 (true)</td>
</tr>
<tr>
<td>#\ E</td>
<td>not E</td>
</tr>
<tr>
<td>E1 #&lt;=&gt; E2</td>
<td>E1 equivalent to E2</td>
</tr>
<tr>
<td>E1 #&lt;=&gt; E2</td>
<td>E1 not equivalent to E2 (i.e. E1 different from E2)</td>
</tr>
<tr>
<td>E1 # E2</td>
<td>E1 exclusive OR E2 (i.e. E1 not equivalent to E2)</td>
</tr>
<tr>
<td>E1 #=&gt; E2</td>
<td>E1 implies E2</td>
</tr>
<tr>
<td>E1 #&gt;&lt;= E2</td>
<td>E1 does not imply E2</td>
</tr>
<tr>
<td>E1 #\ E2</td>
<td>E1 AND E2</td>
</tr>
<tr>
<td>E1 #/ E2</td>
<td>E1 NAND E2</td>
</tr>
<tr>
<td>E1 #/ E2</td>
<td>E1 OR E2</td>
</tr>
<tr>
<td>E1 #/ E2</td>
<td>E1 NOR E2</td>
</tr>
</tbody>
</table>

#<=>, #\<=>, #<=>, #\=>, #\, #\/, #\ and #\/ are predefined infix operators. #\ is a predefined prefix operator (section 6.14.10, page 87).

Errors

| a sub-expression E is neither a variable nor an integer (0 or 1) nor an FD boolean functor nor reified constraint | type_error(fd_bool_evaluable, E) |
| an expression is too complex | resource_error(too_big_fd_constraint) |
| a sub-expression is an invalid reified constraint | an arithmetic constraint error (section 7.6.1, page 153) |

7.7.2 (#\)/1 - constraint NOT, (#<=>)/2 - constraint equivalent, (#\<=>)/2 - constraint different, (#\)/2 - constraint XOR, (#===>)/2 - constraint imply, (#\===>)/2 - constraint not imply, (#\)/2 - constraint AND, (#\)/2 - constraint NAND, (#\)/2 - constraint OR, (#\)/2 - constraint NOR

Templates

```
#\(?fd_bool_evaluable)
#<=>(?fd_bool_evaluable, ?fd_bool_evaluable)
#\(?fd_bool_evaluable, ?fd_bool_evaluable)
#\(?fd_bool_evaluable, ?fd_bool_evaluable)
#\(?fd_bool_evaluable, ?fd_bool_evaluable)
#\(?fd_bool_evaluable, ?fd_bool_evaluable)
#\(?fd_bool_evaluable, ?fd_bool_evaluable)
#\(?fd_bool_evaluable, ?fd_bool_evaluable)
```

Description

#\= FdBoolExpr1 constraints FdBoolExpr1 to be false.

FdBoolExpr1 #<=> FdBoolExpr2 constrains FdBoolExpr1 to be equivalent to FdBoolExpr2.

FdBoolExpr1 #\<=> FdBoolExpr2 constrains FdBoolExpr1 to be equivalent to not FdBoolExpr2.
FdBoolExpr1 #= FdBoolExpr2 constrains FdBoolExpr1 XOR FdBoolExpr2 to be true.

FdBoolExpr1 #== FdBoolExpr2 constrains FdBoolExpr1 to imply FdBoolExpr2.

FdBoolExpr1 #\== FdBoolExpr1 to not imply FdBoolExpr2.

FdBoolExpr1 #/ FdBoolExpr2 constrains FdBoolExpr1 AND FdBoolExpr2 to be true.

FdBoolExpr1 #/ FdBoolExpr2 constrains FdBoolExpr1 AND FdBoolExpr2 to be false.

FdBoolExpr1 #/ FdBoolExpr2 constrains FdBoolExpr1 OR FdBoolExpr2 to be true.

FdBoolExpr1 #/ FdBoolExpr2 constrains FdBoolExpr1 OR FdBoolExpr2 to be false.

FdBoolExpr1 and FdBoolExpr2 are boolean FD expressions (section 7.7.1, page 156).

Note that #\=< (not equivalent) and ## (exclusive or) are synonymous.

These predicates post boolean constraints that are managed by the FD solver using a partial arc-consistency algorithm to reduce the domain of involved variables. The (dis)entailment of reified constraints is detected using either the bounds (for partial AC arithmetic constraints) or the full domain (for full AC arithmetic constraints).

#=<, #\=<, #=<, #\==, #\=/, #\=/, #\=/ and #\=/ are predefined infix operators. #\ is a predefined prefix operator (section 6.14.10, page 87).

Errors

Refer to the syntax of boolean FD expressions for possible errors (section 7.7.1, page 156).

Portability

GNU Prolog predicates.

7.7.3 fd_cardinality/2, fd_cardinality/3, fd_at_least_one/1, fd_at_most_one/1, fd_only_one/1

Templates

fd_cardinality(+fd_bool_evaluable_list, ?fd_variable)
fd_cardinality(+integer, ?fd_variable, +integer)
fd_at_least_one(+fd_bool_evaluable_list)
fd_at_most_one(+fd_bool_evaluable_list)
fd_only_one(+fd_bool_evaluable_list)

Description

fd_cardinality(List, Count) unifies Count with the number of constraints that are true in List. This is equivalent to post the constraint B_1 + B_2 + ... + B_n #= Count where each variable B_i is a new variable defined by the constraint B_i #=< C_i where C_i is the i\textsuperscript{th} constraint of List. Each C_i must be a boolean FD expression (section 7.7.1, page 156).

fd_cardinality(Lower, List, Upper) is equivalent to fd_cardinality(List, Count), Lower #=< Count, Count #=< Upper

fd_at_least_one(List) is equivalent to fd_cardinality(List, Count), Count #=< 1.
7.8 Symbolic constraints

7.8.1 fd_all_different/1

Templates

\[ \text{fd_all_different(+fd_variable_list)} \]

Description

\( \text{fd_all_different(List)} \) constrains all variables in \( \text{List} \) to take distinct values. This is equivalent to posting an inequality constraint for each pair of variables. This constraint is triggered when a variable becomes ground, removing its value from the domain of the other variables.

Errors

| List is a partial list                     | instantiation_error     |
| List is neither a partial list nor a list | type_error(list, List)  |
| Count is neither an FD variable nor an integer | type_error(fd_variable, Count) |
| Lower is a variable                       | instantiation_error     |
| Lower is neither a variable nor an integer | type_error(integer, Lower) |
| Upper is a variable                       | instantiation_error     |
| Upper is neither a variable nor an integer | type_error(integer, Upper) |
| an element \( E \) of the List list is an invalid boolean expression | an FD boolean constraint (section 7.7.1, page 156) |

Portability

GNU Prolog predicate.

7.8.2 fd_element/3

Templates

\[ \text{fd_element(?fd_variable, +integer_list, ?fd_variable)} \]

Description

\( \text{fd_element(I, List, X)} \) constraints \( X \) to be equal to the \( I \)th integer (from 1) of \( \text{List} \).
Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>I is neither a variable nor an FD variable nor an integer</td>
<td><code>type_error(fd_variable, I)</code></td>
</tr>
<tr>
<td>X is neither a variable nor an FD variable nor an integer</td>
<td><code>type_error(fd_variable, X)</code></td>
</tr>
<tr>
<td>List is a partial list or a list with an element E which is a variable</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>List is neither a partial list nor a list</td>
<td><code>type_error(list, List)</code></td>
</tr>
<tr>
<td>an element E of the List list is neither a variable nor an integer</td>
<td><code>type_error(fd_variable, E)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

7.8.3 `fd_element_var/3"

Templates

```
fd_element_var(?fd_variable, +fd_variable_list, ?fd_variable)
```

Description

`fd_element_var(I, List, X)` constraints `X` to be equal to the `I`th variable (from 1) of List. This constraint is similar to `fd_element/3` (section 7.8.2, page 159) but List can also contain FD variables (rather than just integers).

Errors

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>I is neither a variable nor an FD variable nor an integer</td>
<td><code>type_error(fd_variable, I)</code></td>
</tr>
<tr>
<td>X is neither a variable nor an FD variable nor an integer</td>
<td><code>type_error(fd_variable, X)</code></td>
</tr>
<tr>
<td>List is a partial list</td>
<td><code>instantiation_error</code></td>
</tr>
<tr>
<td>List is neither a partial list nor a list</td>
<td><code>type_error(list, List)</code></td>
</tr>
<tr>
<td>an element E of the List list is neither a variable nor an integer</td>
<td><code>type_error(fd_variable, E)</code></td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicate.

7.8.4 `fd_atmost/3, fd_atleast/3, fd_exactly/3"

Templates

```
fd_atmost(+integer, +fd_variable_list, +integer)
fd_atleast(+integer, +fd_variable_list, +integer)
fd_exactly(+integer, +fd_variable_list, +integer)
```

Description

`fd_atmost(N, List, V)` posts the constraint that at most `N` variables of List are equal to the value `V`. 
7.8 Symbolic constraints

fd_atleast(N, List, V) posts the constraint that at least N variables of List are equal to the value V.

fd_exactly(N, List, V) posts the constraint that at exactly N variables of List are equal to the value V.

These constraints are special cases of \texttt{fd_cardinality/2} (section 7.7.3, page 158) but their implementation is more efficient.

Errors

<table>
<thead>
<tr>
<th>Description</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>N is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>N is neither a variable nor an integer</td>
<td>type_error(integer, N)</td>
</tr>
<tr>
<td>V is a variable</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>V is neither a variable nor an integer</td>
<td>type_error(integer, V)</td>
</tr>
<tr>
<td>List is a partial list</td>
<td>instantiation_error</td>
</tr>
<tr>
<td>List is neither a partial list nor a list</td>
<td>type_error(list, List)</td>
</tr>
<tr>
<td>an element E of the List list is neither a variable nor an FD variable nor an integer</td>
<td>type_error(fd_variable, E)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.

7.8.5 \texttt{fd_relation/2, fd_relationc/2}

Templates

\begin{center}
\begin{verbatim}
fd_relation(+integer_list_list, ?fd_variable_list)
fd_relationc(+integer_list_list, ?fd_variable_list)
\end{verbatim}
\end{center}

Description

\texttt{fd_relation(Relation, Vars)} constrains the tuple of variables \texttt{Vars} to be equal to one tuple of the list \texttt{Relation}. A tuple is represented by a list.

Example: definition of the boolean AND relation so that \texttt{X AND Y \leftrightarrow Z}:

\begin{verbatim}
and(X,Y,Z):-
    fd_relation([[0,0,0],[0,1,0],[1,0,0],[1,1,1]], [X,Y,Z]).
\end{verbatim}

\texttt{fd_relationc(Colns, Vars)} is similar to \texttt{fd_relation/2} except that the relation is not given as the list of tuples but as the list of the columns of the relation. A column is represented by a list.

Example:

\begin{verbatim}
and(X,Y,Z):-
    fd_relationc([[0,0,1,1],[0,1,0,1],[0,0,0,1]], [X,Y,Z]).
\end{verbatim}

Errors
Relation is a partial list or a list with a sub-term E which is a variable | instantiation_error

Relation is neither a partial list nor a list an element E of the Relation list is neither a variable nor an integer | type_error(list, Relation)

Vars is a partial list | instantiation_error

Vars is neither a partial list nor a list an element E of the Vars list is neither a variable nor an integer nor an FD variable | type_error(fd_variable, E)

Portability

GNU Prolog predicates.

7.9 Labeling constraints

7.9.1 fd_labeling/2, fd_labeling/1, fd_labelingff/1

Templates

fd_labeling(+fd_variable_list, +fd_labeling_option_list)
fd_labeling(+fd_variable, +fd_labeling_option_list)
fd_labeling(+fd_variable_list)
fd_labeling(+fd_variable)
fd_labelingff(+fd_variable_list)
fd_labelingff(+fd_variable)

Description

fd_labeling(Vars, Options) assigns a value to each variable X of the list Vars according to the list of labeling options given by Options. Vars can be also a single FD variable. This predicate is re-executable on backtracking.

FD labeling options: Options is a list of labeling options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- variable_method(V): specifies the heuristics to select the variable to enumerate:
  - standard: no heuristics, the leftmost variable is selected.
  - first_fail (or ff): selects the variable with the smallest number of elements in its domain. If several variables have the same number of elements the leftmost variable is selected.
  - most_constrained: like first_fail but when several variables have the same number of elements selects the variable that appears in most constraints.
  - smallest: selects the variable that has the smallest value in its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - largest: selects the variable that has the greatest value in its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - max_regret: selects the variable that has the greatest difference between the smallest value and the next value of its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - random: selects randomly a variable. Each variable is only chosen once.

The default value is standard.
7.10 Optimization constraints

7.10.1 fd_minimize/2, fd_maximize/2

Templates

\[
\text{fd_minimize}(\text{callable_term}, \text{?fd_variable}) \\
\text{fd_maximize}(\text{callable_term}, \text{?fd_variable})
\]

Description

\text{fd_minimize}(\text{Goal}, X) \text{ repeatedly calls Goal to find a value that minimizes the variable X. Goal is a Prolog goal that should instantiate X, a common case being the use of \text{fd_labeling/2} (section 7.9.1, page 162). This predicate uses a branch-and-bound algorithm with restart: each time call(Goal) succeeds the computation restarts with a new constraint } X \#< V \text{ where V is the value of X at the end of the last call of Goal. When a failure occurs (either because there are no remaining choice-points for Goal or because the added constraint is inconsistent with the rest of the store) the last solution is recomputed since it is optimal.}

\text{fd_maximize}(\text{Goal}, X) \text{ is similar to } \text{fd_minimize/2 but X is maximized.}
Errors

<table>
<thead>
<tr>
<th>Goal is a variable</th>
<th>instantiation_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal is neither a variable nor a callable term</td>
<td>type_error(callable, Goal)</td>
</tr>
<tr>
<td>The predicate indicator Pred of Goal does not correspond to an existing procedure and the value of the unknown Prolog flag is error (section 6.22.1, page 117)</td>
<td>existence_error(procedure, Pred)</td>
</tr>
<tr>
<td>X is neither a variable nor an FD variable nor an integer</td>
<td>type_error(fd_variable, X)</td>
</tr>
</tbody>
</table>

Portability

GNU Prolog predicates.
8 Interfacing Prolog and C

8.1 Calling C from Prolog

8.1.1 Introduction

This interface allows a Prolog predicate to call a C function. Here are some features of this facility:

- implicit Prolog ↔ C data conversions for simple types.
- functions to handle complex types.
- error detection depending on the type of the argument.
- different kinds of arguments: input, output or input/output.
- possibility to write non-deterministic code.

This interface can then be used to write both simple and complex C routines. A simple routine uses either input or output arguments which type is simple. In that case the user does not need any knowledge of Prolog data structures since all Prolog ↔ C data conversions are implicitly achieved. To manipulate complex terms (lists, structures) a set of functions is provided. Finally it is also possible to write non-deterministic C code.

8.1.2 foreign/2 directive

The foreign/2 directive (section 5.1.13, page 33) directive is provided to declare a C function interface. The general form is foreign(Template, Options) which defines an interface predicate whose prototype is Template according to the options given by Options. Template is a callable term specifying the type/mode of each argument of the associated Prolog predicate.

Foreign options: Options is a list of foreign options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- fct_name(F): F is an atom representing the name of the C function to call. By default the name of the C function is the same as the principal functor of Template. In any case, the atom associated to the name of the function must conforms to the syntax of C identifiers.

- return(boolean/none/jump): specifies the value returned by the C function:
  - boolean: the type of the function is Bool (returns TRUE on success, FALSE otherwise).
  - none: the type of the function is void (no returned value).
  - jump: the type of the function is void(*) () (returns the address of a Prolog code to execute). The default value is boolean.

- bip_name(Name, Arity): initializes the error context with Name and Arity. If an error occurs this information is used to indicate from which predicate the error occurred (section 4.3.1, page 25). It is also possible to prevent the initialization of the error context using bip_name(none). By default Name and Arity are set to the functor and arity of Template.

- choice_size(N): this option specifies that the function implements a non-deterministic code. N is an integer specifying the size needed by the non-deterministic C function. This facility is explained later (section 8.1.7, page 168). By default a foreign function is deterministic.
foreign(Template) is equivalent to foreign(Template, []).

**Foreign modes and types:** each argument of Template specifies the foreign mode and type of the corresponding argument. This information is used to check the type of effective arguments at run-time and to perform Prolog ↔ C data conversions. Each argument of Template is formed with a mode symbol followed by a type name. Possible foreign modes are:

- +: input argument.
- -: output argument.
- ?: input/output argument.

Possible foreign types are:

<table>
<thead>
<tr>
<th>Foreign type</th>
<th>Prolog type</th>
<th>C type</th>
<th>Description of the C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer</td>
<td>int</td>
<td>value of the integer</td>
</tr>
<tr>
<td>positive</td>
<td>positive integer</td>
<td>int</td>
<td>value of the integer</td>
</tr>
<tr>
<td>float</td>
<td>floating point number</td>
<td>double</td>
<td>value of the floating point number</td>
</tr>
<tr>
<td>number</td>
<td>number</td>
<td>double</td>
<td>value of the number</td>
</tr>
<tr>
<td>atom</td>
<td>atom</td>
<td>int</td>
<td>internal key of the atom</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>int</td>
<td>value of the boolean (0=false, 1=true)</td>
</tr>
<tr>
<td>char</td>
<td>character</td>
<td>int</td>
<td>value of the character-code</td>
</tr>
<tr>
<td>Code</td>
<td>character code</td>
<td>int</td>
<td>value of the character-code</td>
</tr>
<tr>
<td>byte</td>
<td>byte</td>
<td>int</td>
<td>value of the byte</td>
</tr>
<tr>
<td>in_char</td>
<td>in-character</td>
<td>int</td>
<td>value of the character or -1 for end-of-file</td>
</tr>
<tr>
<td>in_code</td>
<td>in-character code</td>
<td>int</td>
<td>value of the character-code or -1 for end-of-file</td>
</tr>
<tr>
<td>in_byte</td>
<td>in-byte</td>
<td>int</td>
<td>value of the byte or -1 for end-of-file</td>
</tr>
<tr>
<td>string</td>
<td>atom</td>
<td>char *</td>
<td>C string containing the name of the atom</td>
</tr>
<tr>
<td>chars</td>
<td>character list</td>
<td>char *</td>
<td>C string containing the characters of the list</td>
</tr>
<tr>
<td>codes</td>
<td>character-code list</td>
<td>char *</td>
<td>C string containing the characters of the list</td>
</tr>
<tr>
<td>term</td>
<td>Prolog term</td>
<td>P1Term</td>
<td>generic Prolog term</td>
</tr>
</tbody>
</table>

**Simple foreign type:** a simple type is any foreign type listed in the above tabled except term. A simple foreign type is an atomic term (character and character-code lists are in fact lists of constants). Each simple foreign type is converted to/from a C type to simplify the writing of the C function.

**Complex foreign type:** type foreign type term refers to any Prolog term (e.g. lists, structures...). When such an type is specified the argument is passed to the C function as a P1Term (GNU Prolog C type equivalent to a long). Several functions are provided to manipulate P1Term variables (section 8.2, page 172). Since the original term is passed to the function it is possible to read its value or to unify it. So the meaning of the mode symbol is less significant. For this reason it is possible to omit the mode symbol. In that case term is equivalent to +term.

### 8.1.3 The C function

The C code is written in a C file which must first include the GNU Prolog header file called gprolog.h. This file contains all GNU Prolog C definitions (constants, types, prototypes, ...).

The type returned by a C function depends on the value of the return foreign option (section 8.1.2, page 165). If it is boolean then the C function is of type bool and shall return TRUE in case of success and FALSE otherwise. If the return option is none the C function is of type void. Finally if it is jump, the function shall return the address of a Prolog predicate and, at the exit of the function, the control is given to that predicate.
The type of the arguments of the C function depends on the mode and type declaration specified in Template for the corresponding argument as explained in the following sections.

8.1.4 Input arguments

An input argument is tested at run-time to check if its type conforms to the foreign type and then it is passed to the C function. The type of the associated C argument is given by the above table (section 8.1.2, page 165). For instance, the effective argument Arg associated to +positive foreign declaration is submitted to the following process:

- If Arg is a variable an instantiation_error is raised.
- If Arg is neither a variable nor an integer a type_error(integer, Arg) is raised.
- If Arg is an integer < 0 a domain_error(not_less_than_zero, Arg) is raised.
- Otherwise the value of Arg is passed to the C is passed to the C function as an integer (int).

When +string is specified the string passed to the function is the internal string of the corresponding atom and should not be modified.

When +term is specified the term passed to the function is the original Prolog term. It can be read and/or unified. It is also the case when term is specified without any mode symbol.

8.1.5 Output arguments

An output argument is tested at run-time to check if its type conforms to the foreign type and it is unified with the value set by the C function. The type of the associated C argument is a pointer to the type given by the above table (section 8.1.2, page 165). For instance, the effective argument Arg associated to -positive foreign declaration is handled as follows:

- If Arg is neither a variable nor an integer a type_error(integer, Arg) is raised.
- If Arg is an integer < 0 a domain_error(not_less_than_zero, Arg) is raised.
- Otherwise a pointer to an integer (int *) is passed to the C function. If the function returns TRUE the integer stored at this location is unified with Arg.

When -term is specified, the function must construct a term into its corresponding argument (which is of type P1Term *). At the exit of the function this term will be unified with the actual predicate argument.

8.1.6 Input/output arguments

Basically an input/output argument is treated as in input argument if it is not a variable, as an output argument otherwise. The type of the associated C argument is a pointer to a FIOArg (GNU Prolog C type) defined as follows:

typedef struct
{
    Bool is_var;
    Bool unify;
    union
    {

```c
long l;
char *s;
double d;
}

The field is_var is set to TRUE if the argument is a variable and FALSE otherwise. This value can be tested by the C function to determine which treatment to perform. The field unify controls whether the effective argument must be unified at the exit of the C function. Initially unify is set to the same value as is_var (i.e. a variable argument will be unified while a non-variable argument will not) but it can be modified by the C function. The field value stores the value of the argument. It is declared as a C union since there are several kinds of value types. The field s is used for C strings, d for C doubles and l otherwise (int, PlTerm). If is_var is FALSE then value contains the input value of the argument with the same conventions as for input arguments (section 8.1.4, page 167). At the exit of the function, if unify is TRUE value must contain the value to unify with the same conventions as for output arguments (section 8.1.5, page 167).

For instance, the effective argument Arg associated to ?positive foreign declaration is handled as follows:

- if Arg is a variable is_var and unify are set to TRUE else to FALSE and its value is copied in value.l.
- if Arg is neither a variable nor an integer a type_error(integer, Arg) is raised.
- if Arg is an integer < 0 a domain_error(not_less_than_zero, Arg) is raised.
- otherwise a pointer to the FI0Arg (FI0Arg *) is passed to the C function. If the function returns TRUE and if unify is TRUE the value stored in value.l is unified with Arg.

### 8.1.7 Writing non-deterministic C code

The interface allows the user to write non-deterministic C code. When a C function is non-deterministic, a choice-point is created for this function. When a failure occurs, if all more recent non-deterministic code are finished, the function is re-invoked. It is then important to inform Prolog when there is no more solution (i.e. no more choice) for a non-deterministic code. So, when no more choices remains the function must remove the choice-point. The interface increments a counter each time the function is re-invoked. At the first call this counter is equal to 0. This information allows the function to detect its first call. When writing non-deterministic code, it is often useful to record data between consecutive re-invocations of the function. The interface maintains a buffer to record such an information. The size of this buffer is given by choice_size(N) when using foreign/2 (section 8.1.2, page 165). This size is the number of (consecutive) longs needed by the C function. Inside the function it is possible to call the following functions/macros:

```c
void Get_Choice_Counter(void)
TYPE Get_Choice_Buffer (TYPE)
void No_More_Choice (void)
```

The function Get_Choice_Counter() returns the value of the invocation counter (0 at the first call).

The macro Get_Choice_Buffer(TYPE) returns a pointer to the buffer (casted to TYPE).

The function No_More_Choice() deletes the choice point associated to the function.
8.1 Calling C from Prolog

8.1.8 Example: input and output arguments

Let us define a predicate `first_occurrence(A, C, P)` which unifies P with the position (from 0) of the first occurrence of the character C in the atom A. The predicate must fail if C does not appear in A.

In the prolog file `prog.pl`:

```prolog
:- foreign(first_occurrence(+string, +char, -positive)).
```

In the C file `utils.c`:

```c
#include <string.h>
#include "gprolog.h"

Bool first_occurrence(char *str, int c, int *pos)
{
    char *p;

    if ((p=strchr(str,c))==NULL) /* C does not appear in A */
        return FALSE; /* fail */
    *pos=p-str; /* set the output argument */
    return TRUE; /* succeed */
}
```

The compilation produces an executable called `prog`:

```bash
% gplc prog.pl utils.c
```

Examples of use:

```prolog
| ?- first_occurrence(prolog, p, X).
X = 0
| ?- first_occurrence(prolog, k, X).
no
| ?- first_occurrence(prolog, A, X).
{exception: error(instantiation_error,first_occurrence/3)}
| ?- first_occurrence(prolog, 1,X).
{exception: error(type_error(character,1),first_occurrence/3)}
```

8.1.9 Example: non-deterministic code

We here define a predicate `occurrence(A, C, P)` which unifies P with the position (from 0) of one occurrence of the character C in the atom A. The predicate will fail if C does not appear in A. The predicate is re-executable on backtracking. The information that must be recorded between two invocations of the function is the next starting position in A to search for C.

In the prolog file `prog.pl`:

```prolog
:- foreign(occurrence(+string, +char, -positive), [choice_size(1)]).
```
In the C file `utils.c`:

```c
#include <string.h>
#include "gprolog.h"

Bool occurrence(char *str, int c, int *pos)
{
    char **info_pos;
    char *p;

    info_pos=Get_Choice_Buffer(char **); /* recover the buffer */
    if (Get_Choice.Counter()==0) /* first invocation ? */
        *info_pos=str;
    if ((p=strchr(*info_pos,c))==NULL) /* C does not appear */
    {
        No_More_Choice(); /* remove choice-point */
        return FALSE; /* fail */
    }
    *pos=p-str; /* set the output argument */
    *info_pos=p+1; /* update next starting pos */
    return TRUE; /* succeed */
}
```

The compilation produces an executable called `prog`:

```
% gplc prog.pll utils.c
```

Examples of use:

```
| ?- occurrence(prolog, o, X).
X = 2 ? (here the user presses ; to compute another solution)
X = 4 ? (here the user presses ; to compute another solution)
no (no more solution)
| ?- occurrence(prolog, k, X).
no
```

In the first example when the second (the last) occurrence is found (X=4) the choice-point remains and the failure is detected only when another solution is requested (by pressing ;). It is possible to improve this behavior by deleting the choice-point when there is no more occurrence. To do this it is necessary to do one search ahead. The information stored is the position of the next occurrence.

Here is the corresponding definition of the C function:

```c
#include <string.h>
#include "gprolog.h"

Bool occurrence(char *str, int c, int *pos)
{
```
char **info_pos;
char *p;

info_pos=Get_Choice_Buffer(char **); /* recover the buffer */
if (Get_Choice_Counter()==0) /* first invocation ? */
{
    if ((p=strchr(str,c))==NULL) /* C does not appear at all */
    {
        No_More_Choice(); /* remove choice-point */
        return FALSE; /* fail */
    }
    *info_pos=p;
}
else /* info_pos = an occurrence */

*pos=*info_pos-str; /* set the output argument */
if ((p=strchr(*info_pos+1,c))==NULL) /* no more occurrence */
    No_More_Choice(); /* remove choice-point */
else
    *info_pos=p; /* else update next solution */

return TRUE; /* succeed */

Examples of use:

| ?- occurrence(prolog, 1, X).
| X = 3 (here the user is not prompted since there is no more alternative)
| ?- occurrence(prolog, o, X).
| X = 2 ? (here the user presses ; to compute another solution)
| X = 4 (here the user is not prompted since there is no more alternative)

8.1.10 Example: input/output arguments

We here define a predicate char ascii(Char, Code) which converts in both directions the character Char and its character-code Code. This predicate is then similar to char code/2 (section 6.19.4, page 103).

In the prolog file prog.pl:

:- foreign(char_ascii(?char, ?code), [fct_name('Char_Ascii')]).

In the C file utils.c:

#include "gprolog.h"

Bool Char_Ascii(FIOArg *c, FIOArg *ascii)
{
    if (!c->is_var) /* Char is not a variable */
    {

ascii->unify=TRUE;        /* enforce unif. of Code */
ascii->value.l=c->value.l;  /* set Code */
return TRUE;               /* succeed */
}

if (ascii->is_var)          /* Code is also a variable */
    Pl_Err_Instantiation(); /* emit instantiation_error */

    c->value.l=ascii->value.l;  /* set Char */
    return TRUE;               /* succeed */
}

If Char is instantiated it is necessary to enforce the unification of Code since it could be instantiated. Recall that by default if an input/output argument is instantiated it will not be unified at the exit of the function (section 8.1.6, page 167). If both Char and Code are variables the function raises an instantiation_error. The way to raise Prolog errors is described later (section 8.3, page 178).

The compilation produces an executable called prog:

% gplc prog.pl utils.c

Examples of use:

<table>
<thead>
<tr>
<th>?- char_ascii(a, X).</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?- char_ascii(X, 65).</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 'A'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?- char_ascii(a, 12).</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
</tr>
</tbody>
</table>

| ?- char_ascii(X, X). |
|{exception: error(instantiation_error, char_ascii/2)} |

| ?- char_ascii(1, 12). |
|{exception: error(type_error(character,1),char_ascii/2)} |

8.2 Manipulating Prolog terms

8.2.1 Introduction

In the following we presents a set of functions to manipulate Prolog terms. For simple foreign terms the functions manipulate simple C types (section 8.1.2, page 165).

Functions managing lists handle an array of 2 elements (of type P1Term) containing the terms corresponding to the head and the tail of the list. For the empty list NULL is passed as the array. These functions require to flatten a list in each sub-list. To simplify the management of proper lists (i.e. lists terminated by []) a set of functions is provided that handle the number of elements of the list (an integer) and an array whose elements (of type P1Term) are the elements of the list. The caller of these functions must provide the array.
Functions managing compound terms handle a functor (the principal functor of the term), an arity \( N \geq 0 \) and an array of \( N \) elements (of type \( {	t PTerm} \)) containing the sub-terms of the compound term. Since a list is a special case of compound term (functor = ' . ' and arity=2) it is possible to use any function managing compound terms to deal with a list but the error detection is not the same. Indeed many functions check if the Prolog argument is correct. The name of a read or unify function checking the Prolog arguments is of the form \( \text{Name} \_ \text{Check}() \). For each of these functions there is a also check-free version called \( \text{Name}() \). We then only present the name of checking functions.

### 8.2.2 Managing Prolog atoms

Each atom has a unique internal key which corresponds to its index in the GNU Prolog atom table. It is possible to obtain the information about an atom and to create new atoms using:

```c
char *Atom_Name (int atom)
int Atom_Length (int atom)
Bool Atom_Needs_Quote (int atom)
Bool Atom_Needs_Scan (int atom)
Bool Is_Valid_Atom (int atom)
int Create_Atom (char *str)
int Create_Allocate_Atom(char *str)
int Find_Atom (char *str)
int atom_char[256]
int atom_nil
int atom_false
int atom_true
int atom_end_of_file
```

The macro \( \text{Atom} \_ \text{Name} \_ \text{(atom)} \) returns the internal string of \( \text{atom} \) (this string should not be modified). The function \( \text{Atom} \_ \text{Length} \_ \text{(atom)} \) returns the length (of the name) of \( \text{atom} \).

The function \( \text{Atom} \_ \text{Needs} \_ \text{Scan} \_ \text{(atom)} \) indicates if the canonical form of \( \text{atom} \) needs to be quoted as done by \( \text{write}(2) \) (section 6.14.6, page 83). In that case \( \text{Atom} \_ \text{Needs} \_ \text{Scan} \_ \text{(atom)} \) indicates if this simply comes down to write quotes around the name of \( \text{atom} \) or if it necessary to scan each character of the name because there are some non-printable characters (or included quote characters). The function \( \text{Is} \_ \text{Valid} \_ \text{Atom} \_ \text{(atom)} \) is true only if \( \text{atom} \) is the internal key of an existing atom.

The function \( \text{Create} \_ \text{Atom} \_ \text{(str)} \) adds a new atom whose name is the content of \( \text{str} \) to the system and returns its internal key. If the atom already exists its key is simply returned. The string \( \text{str} \) passed to the function should not be modified later. The function \( \text{Create} \_ \text{Allocate} \_ \text{Atom} \_ \text{(str)} \) is provided when this condition cannot be ensured. It simply makes a dynamic copy of \( \text{str} \).

The function \( \text{Find} \_ \text{Atom} \_ \text{(str)} \) returns the internal key of the atom whose name is \( \text{str} \) or -1 if does not exist.

All atoms corresponding to a single character already exist and their key can be obtained via the global array \( \text{atom} \_ \text{char}[\]\. For instance \( \text{atom} \_ \text{char} [' . ' ] \) is the atom associated to ' . ' (this atom is the functor of lists). The other variables correspond to the internal key of frequently used atoms: [], false, true and end_of_file.

### 8.2.3 Reading Prolog terms

The name of all functions presented here are of the form \( \text{Rd} \_ \text{Name} \_ \text{Check}() \). They all check the validity of the Prolog term to read emitting appropriate errors if necessary. Each function has a check-free version called \( \text{Rd} \_ \text{Name}() \).
Simple foreign types: for each simple foreign type (section 8.1.2, page 165) there is a read function (used by the interface when an input argument is provided):

```c
int Rd_Integer_Check (PlTerm term)
int Rd_Positive_Check (PlTerm term)
double Rd_Float_Check (PlTerm term)
double Rd_Number_Check (PlTerm term)
int Rd_Atomic_Check (PlTerm term)
int Rd_Boolean_Check (PlTerm term)
int Rd_Char_Check (PlTerm term)
int Rd_In_Char_Check (PlTerm term)
int Rd_Code_Check (PlTerm term)
int Rd_In_Code_Check (PlTerm term)
int Rd_In_Byte_Check (PlTerm term)
char *Rd_String_Check (PlTerm term)
char *Rd_Chars_Check (PlTerm term)
char *Rd_Codes_Check (PlTerm term)
int Rd_Chars_Str_Check (PlTerm term, char *str)
int Rd_Codes_Str_Check (PlTerm term, char *str)
```

All functions returning a C string (char *) use a same buffer. The function `Rd_Chars_Str_Check()` is similar to `Rd_Chars_Check()` but accepts as argument a string to store the result and returns the length of that string (which is also the length of the Prolog list). Similarly for `Rd_Codes_Str_Check()`.

Complex terms: the following functions return the sub-arguments (terms) of complex terms as an array of `PlTerm` except `Rd_Proper_List_Check()` which returns the size of the list read (and initializes the array element). Refer to the introduction of this section for more information about the arguments of complex functions (section 8.2.1, page 172).

```c
int Rd_Proper_List_Check (PlTerm term, PlTerm *arg)
PlTerm *Rd_List_Check (PlTerm term)
PlTerm *Rd_Compound_Check (PlTerm term, int *functor, int *arity)
PlTerm *Rd_Callable_Check (PlTerm term, int *functor, int *arity)
```

8.2.4 Unifying Prolog terms

The name of all functions presented here are of the form `Un_Name_Check()`. They all check the validity of the Prolog term to unify emitting appropriate errors if necessary. Each function has a check-free version called `Un_Name()`.

Simple foreign types: for each simple foreign type (section 8.1.2, page 165) there is an unify function (used by the interface when an output argument is provided):

```c
Bool Un_Integer_Check (int n, PlTerm term)
Bool Un_Positive_Check (int n, PlTerm term)
Bool Un_Float_Check (double n, PlTerm term)
Bool Un_Number_Check (double n, PlTerm term)
Bool Un_Atom_Check (int atom, PlTerm term)
Bool Un_Boolean_Check (int b, PlTerm term)
Bool Un_Char_Check (int c, PlTerm term)
Bool Un_In_Char_Check (int c, PlTerm term)
Bool Un_Code_Check (int c, PlTerm term)
Bool Un_In_Code_Check (int c, PlTerm term)
Bool Un_Byte_Check (int b, PlTerm term)
Bool Un_In_Byte_Check (int b, PlTerm term)
```
8.2 Manipulating Prolog terms

Bool Un_String_Check (char *str, P1Term term)
Bool Un_Chars_Check (char *str, P1Term term)
Bool Un_Codes_Check (char *str, P1Term term)

The function Un_Number_Check(n, term) unifies term with an integer if n is an integer, with a floating point number otherwise. The function Un_String_Check(str, term) creates the atom corresponding to str and then unifies term with it (same as Un_Atom_Check(CreateAllocate_Atom(str), term)).

Complex terms: the following functions accept the sub-arguments (terms) of complex terms as an array of P1Term. Refer to the introduction of this section for more information about the arguments of complex functions (section 8.2.1, page 172).

Bool Un_Proper_List_Check(int size, P1Term *arg, P1Term term)
Bool Un_List_Check (P1Term *arg, P1Term term)
Bool Un_Compound_Check (int functor, int arity, P1Term *arg, P1Term term)
Bool Un_Callable_Check (int functor, int arity, P1Term *arg, P1Term term)

All these functions check the type of the term to unify and return the result of the unification. Generally if an unification fails the C function returns FALSE to enforce a failure. However if there are several arguments to unify and if an unification fails then the C function returns FALSE and the type of other arguments has not been checked. Normally all error cases are tested before doing any work to be sure that the predicate fails/succeeds only if no error condition is satisfied. So a good method is to check if the validity of all arguments to unify and later to do the unification (using check-free functions). Obviously if there is only one to unify it is more efficient to use a unify function checking the argument. For the other cases the interface provides a set of functions to check the type of a term.

Simple foreign types: for each simple foreign type (section 8.1.2, page 165) there is check-for-unification function (used by the interface when an output argument is provided):

void Check_For_Un_Integer (P1Term term)
void Check_For_Un_Positive(P1Term term)
void Check_For_Un_Float (P1Term term)
void Check_For_Un_Number (P1Term term)
void Check_For_Un_Atom (P1Term term)
void Check_For_Un_Boolean (P1Term term)
void Check_For_Un_Char (P1Term term)
void Check_For_Un_In_Char (P1Term term)
void Check_For_Un_Code (P1Term term)
void Check_For_Un_In_Code (P1Term term)
void Check_For_Un_Byte (P1Term term)
void Check_For_Un_In_Byte (P1Term term)
void Check_For-Un_String (P1Term term)
void Check_For_Un_Chars (P1Term term)
void Check_For_Un_Codes (P1Term term)

Complex terms: the following functions check the validity of complex terms:

void Check_For_Un_List (P1Term term)
void Check_For_Un_Compound(P1Term term)
void Check_For_Un_Callable(P1Term term)
void Check_For_Un_Variable(P1Term term)

The function Check_For_Un_List(term) checks if term can be unified with a list. This test is done for the entire list (not only for the functor/arity of term but also recursively on the tail of the list). The function Check_For_Un_Variable(term) ensures that term is not currently instantiated. These functions can be defined using functions to test the type of a Prolog term (section 8.2.6, page 176) and functions to raise Prolog errors (section 8.3, page 178). For instance Check_For_Un_List(term) is defined as follows:
8.2.5 Creating Prolog terms

These functions are provided to creates Prolog terms. Each function returns a PlTerm containing the created term.

**Simple foreign types:** for each simple foreign type (section 8.1.2, page 165) there is a creation function:

```c
PlTerm Mk_Integer (int n) 
PlTerm Mk_Positive(int n) 
PlTerm Mk_Float  (double n) 
PlTerm Mk_Number (double n) 
PlTerm Mk_Atom   (int atom) 
PlTerm Mk_Boolean (int b) 
PlTerm Mk_Char    (int c) 
PlTerm Mk_In_Char (int c) 
PlTerm Mk_Code   (int c) 
PlTerm Mk_In_Code (int c) 
PlTerm Mk_Byte   (int b) 
PlTerm Mk_In_Byte (int b) 
PlTerm Mk_String (char *str) 
PlTerm Mk_Chars  (char *str) 
PlTerm Mk_Codes  (char *str)
```

The function Mk_Number(n, term) initializes term with an integer if n is an integer, with a floating point number otherwise. The function Mk_String(str) first creates an atom corresponding to str and then returns that Prolog atom (i.e. equivalent to Mk_Atom(Create_Allocate_Atom(str))).

**Complex terms:** the following functions accept the sub-arguments (terms) of complex terms as an array of PlTerm. Refer to the introduction of this section for more information about the arguments of complex functions (section 8.2.1, page 172).

```c
PlTerm Mk_Proper_List(int size, PlTerm *arg) 
PlTerm Mk_List       (PlTerm *arg) 
PlTerm Mk_Compound   (int functor, int arity, PlTerm *arg) 
PlTerm Mk_Callable   (int functor, int arity, PlTerm *arg)
```

8.2.6 Testing the type of Prolog terms

The following functions test the type of a Prolog term. Each function corresponds to a type testing built-in predicate (section 6.1.1, page 37).

```c
Bool B1t_Var        (PlTerm term) 
Bool B1t_Nom_Var    (PlTerm term) 
Bool B1t_Atom       (PlTerm term) 
Bool B1t_Integer    (PlTerm term) 
Bool B1t_Float      (PlTerm term) 
Bool B1t_Number     (PlTerm term) 
Bool B1t_Atomic     (PlTerm term) 
```
8.2 Manipulating Prolog terms

Bool Blt_Compound (PlTerm term)
Bool Blt_Callable (PlTerm term)
Bool Blt_List (PlTerm term)
Bool Blt_Partial_List (PlTerm term)
Bool Blt_List_Or_Partial_List (PlTerm term)
Bool Blt_Fd_Var (PlTerm term)
Bool Blt_Non_Fd_Var (PlTerm term)
Bool Blt_Generic_Var (PlTerm term)
int Type_Of_Term (PlTerm term)
int List_Length (PlTerm list)

The function Type_Of_Term(term) returns the type of term, the following constants can be used to test this type (e.g. in a switch instruction):

- PLV: Prolog variable.
- FDV: finite domain variable.
- INT: integer.
- FLT: floating point number.
- ATN: atom.
- LST: list.
- STC: structure

The tag LST means a term whose principal functor is '.' and whose arity is 2 (recall that the empty list is the atom []). The tag STC means any other compound term.

The function List_Length(list) returns the number of elements of the list (0 for the empty list). If list is not a list this function returns -1.

8.2.7 Comparing Prolog terms

The following functions compares Prolog terms. Each function corresponds to a comparison built-in predicate (section 6.3.2, page 39).

Bool Blt_Term_Eq (PlTerm term1, PlTerm term2)
Bool Blt_Term_Neq (PlTerm term1, PlTerm term2)
Bool Blt_Term_Lt (PlTerm term1, PlTerm term2)
Bool Blt_Term_Lte (PlTerm term1, PlTerm term2)
Bool Blt_Term_Gt (PlTerm term1, PlTerm term2)
Bool Blt_Term_Gte (PlTerm term1, PlTerm term2)

All these functions are based on a general comparison function returning a negative integer if term1 is less than term2, 0 if they are equal and a positive integer otherwise:

int Term_Compare (PlTerm term1, PlTerm term2)

8.2.8 Copying Prolog terms

The following functions make a copy of a Prolog term:
void Copy_Term (PlTerm *dst_adr, PlTerm *src_adr)
void Copy_Contiguous_Term(PlTerm *dst_adr, PlTerm *src_adr)
int Term_Size (PlTerm term)

The function Copy_Term(dst_adr, src_adr) makes a copy of the term located at src_adr and stores it from the address given by dst_adr. The result is a contiguous term. If it can be ensured that the source term is a contiguous term (i.e. result of a previous copy) the function Copy_Contiguous_Term() can be used instead (it is faster). In any case, sufficient space should be available for the copy (i.e. from dst_adr). The function Term_Size(term) returns the number of PlTerm needed by term.

8.2.9 Comparing and evaluating arithmetic expressions

The following functions compare arithmetic expressions. Each function corresponds to a comparison built-in predicate (section 6.6.3, page 48).

    Bool Blt_Eq (PlTerm expr1, PlTerm expr2)
    Bool Blt_Neq(PlTerm expr1, PlTerm expr2)
    Bool Blt_Lt (PlTerm expr1, PlTerm expr2)
    Bool Blt_Lte(PlTerm expr1, PlTerm expr2)
    Bool Blt_Gt (PlTerm expr1, PlTerm expr2)
    Bool Blt_Gte(PlTerm expr1, PlTerm expr2)

The following function evaluates the expression expr and stores its result as a Prolog number (integer or floating point number) in result:

    void Math_Load_Value(PlTerm expr, PlTerm *result)

This function can be followed by a read function (section 8.2.3, page 173) to obtain the result.

8.3 Raising Prolog errors

The following functions allows a C function to raise a Prolog error. Refer to the section concerning Prolog errors for more information about the effect of raising an error (section 4.3, page 25).

8.3.1 Managing the error context

When one of the following error function is invoked it refers to the implicit error context (section 4.3.1, page 25). This context indicates the name and the arity of the concerned predicate. When using a foreign/2 declaration this context is set by default to the name and arity of the associated Prolog predicate. This can be controlled using the bip_name option (section 8.1.2, page 165). In any case, the following functions can also be used to modify this context:

    void Set_C_Bip_Name (char *functor, int arity)
    void Unset_C_Bip_Name(void)

The function Set_C_Bip_Name(functor, arity) initializes the context of the error with functor and arity (if arity<0 only functor is significant). The function Unset_C_Bip_Name() removes such an initialization (the context is then reset to the last Functor/Arity set by a call to set_bip_name/2 (section 6.22.3, page 119). This is useful when writing a C routine to define a context for errors occurring in this routine and, before exiting to restore the previous context.
8.3 Raising Prolog errors

8.3.2 Instantiation error

The following function raises an instantiation error (section 4.3.2, page 25):

```c
void P1_Err_Instantiation(void)
```

8.3.3 Type error

The following function raises a type error (section 4.3.3, page 26):

```c
void P1_Err_Type(int atom_type, P1Term culprit)
```

Atom `atom_type` is (the internal key of) the atom associated to the expected type. For each type name `T` there is a corresponding predefined atom stored in a global variable whose name is of the form `type_T`. `culprit` is the argument which caused the error.

Example: `x` is an atom while an integer was expected: `P1_Err_Type(type_integer, x)`.

8.3.4 Domain error

The following function raises a domain error (section 4.3.4, page 26):

```c
void P1_Err_Domain(int atom_domain, P1Term culprit)
```

Atom `atom_domain` is (the internal key of) the atom associated to the expected domain. For each domain name `D` there is a corresponding predefined atom stored in a global variable whose name is of the form `domain_D`. `culprit` is the argument which caused the error.

Example: `x` is `< 0` but should be `≥ 0`: `P1_Err_Domain(domain_not_less_than_zero, x)`.

8.3.5 Existence error

The following function raises an existence error (section 4.3.5, page 27):

```c
void P1_Err_Existence(int atom_object, P1Term culprit)
```

Atom `atom_object` is (the internal key of) the atom associated to the type of the object. For each object name `O` there is a corresponding predefined atom stored in a global variable whose name is of the form `existence_O`. `culprit` is the argument which caused the error.

Example: `x` does not refer to an existing source: `P1_Err_Existence(existence_source_sink, x)`.

8.3.6 Permission error

The following function raises a permission error (section 4.3.6, page 27):

```c
void P1_Err_Permission(int atom_operation, int atom_permission, P1Term culprit)
```

Atom `atom_operation` is (the internal key of) the atom associated to the operation which caused the error. For each operation name `O` there is a corresponding predefined atom stored in a global variable whose name is of the form `permission_operation_O`. `atom_permission` is (the internal key of) the atom associated
to the tried permission. For each permission name \( P \) there is a corresponding predefined atom stored in a global variable whose name is of the form \texttt{permission.type}_{P}. \texttt{culprit} is the argument which caused the error.

Example: reading from an output stream \( x \): \texttt{Pl.Err.Permission(permission.operation.input, permission.type.stream, x)}.

8.3.7 Representation error

The following function raises a representation error (section 4.3.7, page 27):

\[
\text{void Pl.Err.Representation(int atom.limit)}
\]

\( \text{atom.limit} \) is (the internal key of) the atom associated to the reached limit. For each limit name \( L \) there is a corresponding predefined atom stored in a global variable whose name is of the form \texttt{representation}_{L}.

Example: an arity too big occurs: \texttt{Pl.Err.Representation(representation.max arity)}.

8.3.8 Evaluation error

The following function raises an evaluation error (section 4.3.8, page 28):

\[
\text{void Pl.Err.Evaluation(int atom.error)}
\]

\( \text{atom.error} \) is (the internal key of) the atom associated to the error. For each evaluation error name \( E \) there is a corresponding predefined atom stored in a global variable whose name is of the form \texttt{evaluation}_{E}.

Example: a division by zero occurs: \texttt{Pl.Err.Evaluation(evaluation.zero.divisor)}.

8.3.9 Resource error

The following function raises a resource error (section 4.3.9, page 28):

\[
\text{void Pl.Err.Resource(int atom.resource)}
\]

\( \text{atom.resource} \) is (the internal key of) the atom associated to the resource. For each resource error name \( R \) there is a corresponding predefined atom stored in a global variable whose name is of the form \texttt{resource}_{R}.

Example: too many open streams: \texttt{Pl.Err.Resource(resource.too.many.open.streams)}.

8.3.10 Syntax error

The following function raises a syntax error (section 4.3.10, page 28):

\[
\text{void Pl.Err.Syntax(int atom.error)}
\]

\( \text{atom.error} \) is (the internal key of) the atom associated to the error. There is no predefined syntax error atoms.
Example: a / is expected: `Pl_Err_Syntax(Create_Atom("/ expected"))`.

The following function emits a syntax error according to the value of the syntax_error Prolog flag (section 6.22.1, page 117). This function can then return (if the value of the flag is either warning or fail). In that case the calling function should fail (e.g. returning FALSE). This function accepts a file name (the empty string C "" can be passed), a line and column number and an error message string. Using this function makes it possible to further call the built-in predicate syntax_error_info/4 (section 6.14.4, page 82):

```c
void Emit_Syntax_Error(char *file_name, int line, int column, char *message)
```

Example: a / is expected: `Emit_Syntax_Error("data", 10, 30, "/ expected")`.

8.3.11 System error

The following function raises a system error (4.3.11, page *):

```c
void Pl_Err_System(int atom_error)
```

`atom_error` is the internal key of the atom associated to the error. There is no predefined system error atoms.

Example: an invalid pathname is given: `Pl_Err_System(Create_Atom("invalid path name"))`.

The following function emits a system error associated to an operating system error according to the value of the os_error Prolog flag (section 6.22.1, page 117). This function can then return (if the value of the flag is either warning or fail). In that case the calling function should fail (e.g. returning FALSE). This function uses the value of the errno C library variable:

```c
void 0s_Error(void)
```

Example: a call to the C Unix function chdir(3) returns -1: `0s_Error()`.

8.4 Calling Prolog from C

8.4.1 Introduction

The following functions allows a C function to call a Prolog predicate:

```c
int Pl_Query_Start  (int functor, int arity, PlTerm *arg,
                    Bool recoverable)
int Pl_Query_Next_Solution(void)
void Pl_Query_End    (int op)
PlTerm Pl_Get_Exception  (void)
void Pl_Exec_Continuation  (int functor, int arity, PlTerm *arg)
```

The invocation of a Prolog predicate should be done as follows:

- open a query using `Pl_Query_Start()` (computes the first solution)
- eventually compute next solutions using `Pl_Query_Next_Solution()`
- close the query using `Pl_Query_End()`
The function `Pl_Query_Start(functor, arity, arg, recoverable)` is used to initialize a query (computes the first solution). The arguments `functor`, `arity` and `arg` are similar to those of the functions handling complex terms (section 8.2.1, page 172). The argument `recoverable` shall be set to `TRUE` if the user wants to recover, at the end of the query, the memory space consumed by the query (in that case an additional choice-point is created). This function returns:

- **PL_FAILURE** (a constant equal to `FALSE`, i.e. 0) if the query fails.
- **PL_SUCCESS** (a constant equal to `TRUE`, i.e. 1) in case of success. In that case the argument array `arg` can be used to obtain the unification performed by the query.
- **PL_EXCEPTION** (a constant equal to 2). In that case function `Pl_Set_Exception()` can be used to obtained the exceptional term raised by `throw/1` (section 5.2.4, page 35).

The function `Pl_Query_Next_Solution()` is used to compute a new solution. It must be only used if the result of the previous solution was **PL_SUCCESS**. This functions returns the same kind of values as `Pl_Query_Start()` (see above).

The function `Pl_Query_End(op)` is used to finish a query. This function mainly manages the remaining alternatives of the query. However, even if the query has no alternatives this function must be used to correctly finish the query. The value of `op` is:

- **PL_RECOVER**: to recover the memory space consumed by the query. After that the state of Prolog stacks is exactly the same as before opening the query. To use this option the query must have been opened specifying `TRUE` for `recoverable` (see above).
- **PL_CUT**: to cut remaining alternatives. The effect of this option is similar to a cut after the query.
- **PL_KEEP_FOR_PROLOG**: to keep the alternatives for Prolog. This is useful when the query was invoked in a foreign C function. In that case, when the predicate corresponding to the C foreign function is invoked a query is executed and the remaining alternatives are then available as alternatives of that predicate.

Note that several queries can be nested since a stack of queries is maintained. For instance, it is possible to call a query and before terminating it to call another query. In that case the first execution of `Pl_Query_End()` will finish the second query (i.e. the inner) and the next execution of `Pl_Query_End()` will finish the first query.

Finally, the function `Pl_Exec_Continuation(functor, arity, arg)` replaces the current calculus by the execution of the specified predicate. The arguments `functor`, `arity` and `arg` are similar to those of the functions handling complex terms (section 8.2.1, page 172).

### 8.4.2 Example: my_call/1 - a call/1 clone

We here define a predicate `my_call(Goal)` which acts like `call(Goal)` except that we do not handle exceptions (if an exception occurs the goal simply fails):

In the prolog file `prog.pl`:

```prolog
:- foreign(my_call(term)).
```

In the C file `utils.c`:

```c
#include <string.h>
#include "@prolog.h"

Bool my_call(PlTerm goal)
```
{  
  P1Term *arg;  
  int  functor, arity;  
  int  result;  

  arg=Rd_Callable_Check(goal, &functor, &arity);  
  result=P1_Query_Start(functor, arity, arg, FALSE);  
  P1_Query_End(PL_KEEP_FOR_PROLOG);  
  return (result==PL_SUCCESS);  
}

The compilation produces an executable called prog:

% gplc prog.pl utils.c

Examples of use:

| ?- my_call(write(hello)).
  hello
| ?- my_call(for(X,1,3)).
  X = 1 ? (here the user presses ; to compute another solution)
  X = 2 ? (here the user presses ; to compute another solution)
  X = 3 (here the user is not prompted since there is no more alternative)
| ?- my_call(1).
  {exception: error(type_error(callable,1),my_call/1)}
| ?- my_call(call(1)).
  no

When my_call(1) is called an error is raised due to the use of Rd_Callable_Check(). However the error raised by my_call(call(1)) is ignored and FALSE (i.e. a failure) is returned by the foreign function.

To really simulate the behavior of call/1 when an exception is recovered it should be re-raised to be captured by an earlier handler. The idea is then to execute a throw/1 as the continuation. This is what it is done by the following code:

```c
#include <string.h>
#include "gprolog.h"

Bool my_call(P1Term goal})
{
  P1Term *arg;
  int  functor, arity;
  int  result;

  arg=Rd_Callable_Check(goal, &functor, &arity);
  result=P1_Query_Start(functor, arity, arg, FALSE);
  P1_Query_End(PL_KEEP_FOR_PROLOG);
  if (result==PL_EXCEPTION)
  {
```
PlTerm except=Pl_Get_Exception();
Pl_Exec_Continuation(Find_Atom("throw"), 1, &except);
}
return (result==PL_SUCCESS);
}

The following code propagates the error raised by call/1.

| ?- my_call(call(1)).
{exception: error(type_error(callable,1),my_call/1)}

Finally note that a simpler way to define my_call/1 is to use Pl_Exec_Continuation() as follows:

#include <string.h>
#include "gprolog.h"

Bool my_call(PlTerm goal)
{
    PlTerm *args;
    int functor, arity;

    args=Rd_Callable_Check(goal, &functor, &arity);
    Pl_Exec_Continuation(functor, arity, args);
    return TRUE;
}

8.4.3 Example: recovering the list of all operators

We here define a predicate all_op(List) which unifies List with the list of all currently defined operators as would be done by: findall(X,current_op(\_,\_,X),List).

In the prolog file prog.pl:

:- foreign(all_op(term)).

In the C file utils.c:

#include <string.h>
#include "gprolog.h"

Bool all_op(PlTerm list)
{
    PlTerm op[1024];
    PlTerm args[3];
    int n=0;
    int result;

    args[0]=Mk_Variable();
    args[1]=Mk_Variable();
    args[2]=Mk_Variable();
    result=Pl_Query_Start(Find_Atom("current_op"), 3, args, TRUE);
    while(result)
    {
        op[n++]=Mk_Atom(Rd_Atom(args[2])); /* arg #2 is the name of the op */
8.5 Defining a new C main() function

```c
result=PL_Query_Next_Solution();
}
PL_Query_End(PL_RECOVER);
return Un_Proper_List_Check(n, op, list);
}
```

Note that we know here that there is no source for exception. In that case the result of `PL_Query_Start` and `PL_Query_End` can be considered as a boolean.

The compilation produces an executable called `prog`:
```
% gplc prog.pl util.c
```

Example of use:
```
| ?- all_op(L).
L = [:-, :-, \
L = [:-, :-, 
```

8.5 Defining a new C main() function

GNU Prolog allows the user to define his own main() function. This can be useful to perform several tasks before starting the Prolog engine. To do this simply define a classical `main(argc, argv)` function.

The following functions can then be used:
```
int Start_Prolog(int argc, char *argv[])
void Reset_Prolog(void)
Bool Try_Execute_Top_Level(void)
```

The function `Start_Prolog(argc, argv)` initializes the Prolog engine (argc and argv are the command-line variables). This function collects all linked objects (issued from the compilation of Prolog files) and initializes them. The initialization of a Prolog object file consists in adding to appropriate tables new atoms, new predicates and executing its system directives. A system directive is generated by the Prolog to WAM compiler to reflect a (user) directive executed at compile-time such as `op/3` (section 5.1.9, page 32). Indeed, when the compiler encounters such a directive it immediately executes it and also generates a system directive to execute it at the start of the executable. When all system directives have been executed the Prolog engine executes all initialization directives defined with `initialization/1` (section 5.1.12, page 33). The function returns the number of user directives (i.e. `initialization/1`) executed.

The function `Reset_Prolog()` reinitializes the Prolog engine (i.e. reset all Prolog stacks).

The function `Try_Execute_Top_Level()` executes the top-level if linked (section 2.4.3, page 12) and returns `TRUE`. If the top-level is not present the functions returns `FALSE`.

Here is the definition of the default GNU Prolog `main()` function:
```
int main(int argc, char *argv[])
{
  int nb_user_directive;
```
nb_user_directive=Start_Prolog(argc, argv);

if (Try_Execute_Top_Level())
    return 0;

if (nb_user_directive)
    return 0;

printf("Warning: no initial goal executed\n"
    " use a directive :- initialization(Goal)\n"
    " or remove the link option --no-top-level"
    " (or --min-bips or --min-size)\n");

    return 1;
}

8.5.1 Example: asking for ancestors

In this example we use the following Prolog code (in a file called prog.pl):

parent(bob, mary).
parent(jane, mary).
parent(mary, peter).
parent(paul, peter).
parent(peter, john).

anc(X, Y):-
    parent(X, Y).

anc(X, Z) :-
    parent(X, Y),
    anc(Y, Z).

The following main() function asks for the name of a person and displays all successors of that person (this is equivalent to the Prolog query: anc(Result, Name)).

int main(int argc, char *argv[])
{
    int func;
    WamWord arg[10];
    char str[100];
    char *sol[100];
    int i, nb_sol=0;
    Bool res;

    Start_Prolog(argc,argv);

    func=Find_Atom("anc");
    for(;;)
    {
        printf("\nEnter a name (or 'end' to finish): ");
        scanf("%s",str);

        if (strcmp(str, "end")==0)
break;

arg[0]=Mk_Variable();
arg[1]=Mk_String(str);
b_sol=0;
res=Pl_Query_Start(func, 2, arg, TRUE);
while(res)
{
    sol[nb_sol++]=Rd_String(arg[0]);
    res=Pl_Query_Next_Solution();
}
Pl_Query_End(PL_RECOVER);

for(i=0;i<nb_sol;i++)
    printf(" solution: \%s\n", sol[i]);
    printf("\%d solution(s)\n", nb_sol);
return 0;
}

The compilation produces an executable called prog:

% gplc prog.pl utils.c

Examples of use:

Enter a name (or 'end' to finish): john
    solution: peter
    solution: bob
    solution: jane
    solution: mary
    solution: paul
5 solution(s)

Enter a name (or 'end' to finish): mary
    solution: bob
    solution: jane
2 solution(s)

Enter a name (or 'end' to finish): end
References

http://www.isg.sfu.ca/~hak/documents/wam.html


ftp://ftp.inria.fr/INRIA/Projects/loco/publications/wamcc/wamcc.ps

ftp://ftp.inria.fr/INRIA/Projects/loco/publications/clp_fd/long_clp_fd.ps


In *Principles Of Programming Languages,* Munich, Germany, January 1987.


## Contents

Acknowledgments  

1 Introduction  

2 Using GNU Prolog  
  2.1 Introduction  
  2.2 The GNU Prolog interactive interpreter  
    2.2.1 Starting/exiting the interactive interpreter  
    2.2.2 The interactive interpreter read-execute-write loop  
    2.2.3 Consulting a Prolog program  
    2.2.4 Interrupting a query  
    2.2.5 The line editor  
  2.3 Adjusting the size of Prolog stacks  
  2.4 The GNU Prolog compiler  
    2.4.1 Different kinds of codes  
    2.4.2 Compilation scheme  
    2.4.3 Using the compiler  
    2.4.4 Running an executable  
    2.4.5 Generating a new interactive interpreter  
    2.4.6 The hexadecimal predicate name encoding  

3 Debugging  
  3.1 Introduction  
  3.2 The procedure box model  
  3.3 Debugging predicates  
    3.3.1 Running and stopping the debugger  
    3.3.2 Leasing ports  
    3.3.3 Spy-points  
  3.4 Debugging messages  
  3.5 Debugger commands  
  3.6 The WAM debugger  

4 Format of definitions  
  4.1 General format  
  4.2 Types and modes  
  4.3 Errors  
    4.3.1 General format and error context  
    4.3.2 Instantiation error  
    4.3.3 Type error  
    4.3.4 Domain error  
    4.3.5 Existence error  
    4.3.6 Permission error  
    4.3.7 Representation error  
    4.3.8 Evaluation error  
    4.3.9 Resource error  
    4.3.10 Syntax error  
    4.3.11 System error  

5 Prolog directives and control constructs  
  5.1 Prolog directives  
    5.1.1 Introduction  
    5.1.2 dynamic/1  
    5.1.3 public/1  
    5.1.4 multifile/1  
    5.1.5 discontiguous/1  
    5.1.6 built_in/0, built_in/1, built_in_fd/0, built_in_fd/1
5.1.7 include/1 ........................................ 31
5.1.8 ensure_loaded/1 .................................. 32
5.1.9 op/3 .............................................. 32
5.1.10 char_conversion/2 ............................... 32
5.1.11 set_prolog_flag/2 ............................... 32
5.1.12 initialization/1 .................................. 33
5.1.13 foreign/2, foreign/1 ............................ 33

5.2 Prolog control constructs ........................... 33
  5.2.1 true/0, fail/0, !/0 .............................. 33
  5.2.2 (',')/2 - conjunction, (;)/2 - disjunction, (->)/2 - if-then .... 34
  5.2.3 call/1 ........................................... 35
  5.2.4 catch/3, throw/1 ............................... 35

6 Prolog built-in predicates ........................... 37
  6.1 Type testing ...................................... 37
      6.1.1 var/1, nonvar/1, atom/1, integer/1, float/1, number/1, atomic/1, compound/1, callable/1, list/1, partial_list/1, list_or_partial_list/1 ... 37
  6.2 Term unification ................................ 38
      6.2.1 (=)/2 - Prolog unification ................... 38
      6.2.2 unify_with_occurs_check/2 .................. 38
      6.2.3 (\=/)/2 - not Prolog unifiable ............... 38
  6.3 Term comparison ................................ 39
      6.3.1 Standard total ordering of terms ............. 39
      6.3.2 (==)/2 - term identical, (\==)/2 - term not identical,
          (\<)/2 - term less than, (\=<)/2 - term less than or equal to,
          (\>)/2 - term greater than, (\>=)/2 - term greater than or equal to ..... 39
      6.3.3 compare/3 .................................... 40
  6.4 Term processing .................................. 40
      6.4.1 functor/3 ..................................... 40
      6.4.2 arg/3 ......................................... 41
      6.4.3 (=..)/2 - univ ................................ 41
      6.4.4 copy_term/2 .................................. 42
      6.4.5 setarg/4, setarg/3 ............................. 42
  6.5 Variable naming/numbering ....................... 43
      6.5.1 name_singleton_vars/1 ........................ 43
      6.5.2 name_query_vars/2 ............................ 43
      6.5.3 bind_variables/2, numbervars/3, numbervars/1 ... 44
      6.5.4 term_ref/2 .................................... 45
  6.6 Arithmetic ....................................... 45
      6.6.1 Evaluation of an arithmetic expression ...... 45
      6.6.2 (is)/2 - evaluate expression ................ 47
      6.6.3 (=:=)/2 - arithmetic equal, (\=\=)/2 - arithmetic not equal,
          (\<)/2 - arithmetic less than, (\=<)/2 - arithmetic less than or equal to,
          (\>)/2 - arithmetic greater than, (\>=)/2 - arithmetic greater than or equal to ... 48
  6.7 Dynamic clause management ....................... 48
      6.7.1 Introduction .................................. 48
      6.7.2 asserta/1, assertz/1 ........................ 49
      6.7.3 retract/1 ..................................... 50
      6.7.4 retractall/1 .................................. 50
      6.7.5 clause/2 ...................................... 51
      6.7.6 abolish/1 ..................................... 51
  6.8 Predicate information ............................. 52
      6.8.1 current_predicate/1 ........................... 52
      6.8.2 predicate_property/2 .......................... 52
  6.9 All solutions ..................................... 53
      6.9.1 Introduction .................................. 53
      6.9.2 findall/3 ..................................... 53
6.14.7 format/3, format/2 ........................................ 85
6.14.8 portray_clause/2, portray_clause/1 ....................... 86
6.14.9 get_print_stream/1 ....................................... 87
6.14.10 op/3 ................................................................ 87
6.14.11 current_op/3 ............................................. 89
6.14.12 char_conversion/2 ........................................ 89
6.14.13 current_char_conversion/2 ............................... 90
6.15 Input/output from/to constant terms ........................... 91
6.15.1 read_term_from_atom/3, read_from_atom/2, read_token_from_atom/2 ........................ 91
6.15.2 read_term_from_chars/3, read_from_chars/2, read_token_from_chars/2 ......... 91
6.15.3 read_term_from_codes/3, read_from_codes/2, read_token_from_codes/2 .... 92
6.15.4 write_term_to_atom/3, write_to_atom/2, writeq_to_atom/2, format_to_atom/3 ................ 92
6.15.5 write_term_to_chars/3, write_to_chars/2, writeq_to_chars/2, write_canonical_to_chars/2, display_to_chars/2, print_to_chars/2, format_to_chars/3 ........................................ 93
6.15.6 write_term_to_codes/3, write_to_codes/2, writeq_to_codes/2, write_canonical_to_codes/2, display_to_codes/2, print_to_codes/2, format_to_codes/3 ........................................ 93
6.16 DEC-10 compatibility input/output ............................ 94
6.16.1 Introduction ................................................ 94
6.16.2 see/1, tell/1, append/1 ................................... 94
6.16.3 seeing/1, telling/1 ....................................... 95
6.16.4 seem/0, told/0 .......................................... 95
6.16.5 get0/1, get/1, skip/1 .................................. 96
6.16.6 put/1, tab/1 ............................................. 96
6.17 Term expansion ............................................. 96
6.17.1 Definite clause grammars ................................ 96
6.17.2 expand_term/2, term_expansion/2 .................... 98
6.17.3 phrase/3, phrase/2 ..................................... 98
6.18 Logic, control and exceptions ................................. 99
6.18.1 abort/0, stop/0, top_level/0, break/0, halt/1, halt/0 ........................................ 99
6.18.2 call/2, once/1, (\+/1 - not provable ................ 100
6.18.3 repeat/0 ............................................... 100
6.18.4 for/3 .................................................... 101
6.19 Atomic term processing ....................................... 101
6.19.1 atom_length/2 ........................................ 101
6.19.2 atom_concat/3 ......................................... 102
6.19.3 sub_atom/5 ........................................... 102
6.19.4 char_code/2 ........................................... 103
6.19.5 lower_upper/2 ......................................... 103
6.19.6 atom_chars/2, atom_codes/2 ......................... 104
6.19.7 number_atom/2, number_chars/2, number_codes/2 .......... 104
6.19.8 name/2 ............................................... 105
6.19.9 atom_hash/2 .......................................... 106
6.19.10 new_atom/3, new_atom/2, new_atom/1 ............... 106
6.19.11 current_atom/1 ...................................... 107
6.19.12 atom_property/2 .................................... 107
6.20 List processing ............................................ 108
6.20.1 append/3 ............................................. 108
6.20.2 member/2, memberchk/2 ................................ 109
6.20.3 reverse/2 ............................................ 109
6.20.4 delete/3, select/3 ................................... 109
6.20.5 permutation/2 ....................................... 110
6.20.6 prefix/2, suffix/2 ................................... 110
6.20.7 sublist/2 ............................................. 110
6.20.8  last/2  .............................................................. 111
6.20.9  length/2  .......................................................... 111
6.20.10 nth/3  ............................................................... 111
6.20.11 max_list/2, min_list/2, sum_list/2  ......................... 112
6.20.12 sort/2, sort0/2, keysort/2 sort/1, sort0/1, keysort/1 ...... 112

6.21  Global variables .................................................... 113
  6.21.1  g_assign/2, g_assignb/2, g_link/2 ................. 114
  6.21.2  g_read/2  ....................................................... 114
  6.21.3  g_array_size/2  ................................................. 115
  6.21.4  Examples .......................................................... 115

6.22  Prolog state ........................................................ 117
  6.22.1  set_prolog_flag/2  ............................................. 117
  6.22.2  current_prolog_flag/2  ..................................... 119
  6.22.3  set_bip_name/2  ................................................. 119
  6.22.4  current_bip_name/2  ........................................... 119
  6.22.5  write_pl_state_file/1, read_pl_state_file/1 ............. 120

6.23  Program state ....................................................... 120
  6.23.1  consult/1, ' /2 - program consult  ..................... 120
  6.23.2  load/1  .......................................................... 121
  6.23.3  listing/1, listing/0  ......................................... 122

6.24  System statistics .................................................. 122
  6.24.1  statistics/0, statistics/2  ................................. 122
  6.24.2  user_time/1, system_time/1, cpu_time/1, real_time/1 .... 123

6.25  Random number generator ......................................... 124
  6.25.1  set_seed/1, randomize/0  .................................. 124
  6.25.2  get_seed/1  ..................................................... 124
  6.25.3  random/1  ....................................................... 124
  6.25.4  random/3  ....................................................... 125

6.26  File name processing ............................................... 125
  6.26.1  absolute_file_name/2  ...................................... 125
  6.26.2  decompose_file_name/4  ..................................... 126
  6.26.3  prolog_file_name/2  ......................................... 126

6.27  Operating system interface ........................................ 127
  6.27.1  argument_counter/1  ........................................ 127
  6.27.2  argument_value/2  ............................................ 127
  6.27.3  argument_list/1  ............................................. 128
  6.27.4  environ/2  ..................................................... 128
  6.27.5  make_directory/1, delete_directory/1, change_directory/1 . 128
  6.27.6  working_directory/1  ....................................... 129
  6.27.7  directory_files/2  ............................................ 129
  6.27.8  rename_file/2  ............................................... 130
  6.27.9  delete_file/1, unlink/1  ................................... 130
  6.27.10  file_permission/2, file_exists/1  ...................... 131
  6.27.11  file_property/2  ............................................ 131
  6.27.12  temporary_name/2  ......................................... 132
  6.27.13  temporary_file/3  ......................................... 133
  6.27.14  date_time/1  ................................................. 134
  6.27.15  host_name/1  ................................................ 134
  6.27.16  os_version/1  ................................................. 135
  6.27.17  architecture/1  ............................................. 135
  6.27.18  shell/2, shell/1, shell/0  .............................. 135
  6.27.19  system/2, system/1  ....................................... 136
  6.27.20  spawn/3, spawn/2  ......................................... 137
  6.27.21  open/3  ....................................................... 137
  6.27.22  exec/5, exec/4  ............................................. 138
  6.27.23  wait/2  ......................................................... 139
  6.27.24  prolog.pid/1  ................................................. 139
7 Finite domain solver and built-in predicates

7.1 Introduction ........................................... 148

7.1.1 Finite Domain variables .......................... 148

7.2 FD variable parameters ............................... 149

7.2.1 fd_max_integer/1 .................................. 149

7.2.2 fd_vector_max/1 .................................... 149

7.2.3 fd_set_vector_max/1 ................................. 150

7.3 Initial value constraints ............................... 150

7.3.1 fd_domain/3, fd_domain_bool/1 .................... 150

7.3.2 fd_domain/2 ......................................... 151

7.4 Type testing ........................................... 151

7.4.1 fd_var/1, non_fd_var/1, generic_var/1, non_generic_var/1 . 151

7.5 FD variable information ............................... 152

7.5.1 fd_min/2, fd_max/2, fd_size/2, fd_dom/2 .......... 152

7.5.2 fd_has_extra_cstr/1, fd_has_vector/1, fd_use_vector/1 . 153

7.6 Arithmetic constraints ................................. 153

7.6.1 FD arithmetic expressions ........................ 153

7.6.2 Partial AC: (\#)=/2 - constraint equal, (\#\#)=/2 - constraint not equal, (\#\#)/2 - constraint less than, (\#\#\#)=/2 - constraint less than or equal, (\#\#\#)/2 - constraint greater than, (\#\#\#\#)=/2 - constraint greater than or equal ........................ 154

7.6.3 Full AC: (\#\#\#)=/2 - constraint equal, (\#\#\#\#)=/2 - constraint not equal, (\#\#\#\#\#)=/2 - constraint less than, (\#\#\#\#\#\#)=/2 - constraint less than or equal, (\#\#\#\#\#\#\#)=/2 - constraint greater than, (\#\#\#\#\#\#\#\#)=/2 - constraint greater than or equal ........................ 155

7.6.4 fd_prime/1, fd_not_prime/1 ....................... 156

7.7 Boolean and reified constraints ....................... 156

7.7.1 Boolean FD expressions .......................... 156

7.7.2 (\#\#)/1 - constraint NOT, (\#\#\#)/2 - constraint equivalent, (\#\#\#)/2 - constraint different, (\#\#\#\#)/2 - constraint XOR, (\#\#\#\#\#)/2 - constraint imply, (\#\#\#\#\#\#)/2 - constraint not imply, (\#\#\#\#\#\#\#)/2 - constraint AND, (\#\#\#\#\#\#\#\#)/2 - constraint NAND, (\#\#\#\#\#\#\#\#\#)/2 - constraint OR, (\#\#\#\#\#\#\#\#\#\#)/2 - constraint NOR .......................... 157

7.7.3 fd_cardinality/2, fd_cardinality/3, fd_at_least_one/1, fd_at_most_one/1, fd_only_one/1 .............................. 158

7.8 Symbolic constraints ................................. 159

7.8.1 fd_all_different/1 ................................. 159

7.8.2 fd_element/3 ....................................... 159

7.8.3 fd_element_var/3 .................................... 160

7.8.4 fd_atmost/3, fd_atleast/3, fd_exactly/3 ............ 160

7.8.5 fd_relation/2, fd_relationc/2 ...................... 161
8 Interfacing Prolog and C 165
8.1 Calling C from Prolog 165
  8.1.1 Introduction 165
  8.1.2 foreign/2 directive 165
  8.1.3 The C function 166
  8.1.4 Input arguments 167
  8.1.5 Output arguments 167
  8.1.6 Input/output arguments 167
  8.1.7 Writing non-deterministic C code 168
  8.1.8 Example: input and output arguments 169
  8.1.9 Example: non-deterministic code 169
  8.1.10 Example: input/output arguments 171
8.2 Manipulating Prolog terms 172
  8.2.1 Introduction 172
  8.2.2 Managing Prolog atoms 173
  8.2.3 Reading Prolog terms 173
  8.2.4 Unifying Prolog terms 174
  8.2.5 Creating Prolog terms 176
  8.2.6 Testing the type of Prolog terms 176
  8.2.7 Comparing Prolog terms 177
  8.2.8 Copying Prolog terms 177
  8.2.9 Comparing and evaluating arithmetic expressions 178
8.3 Raising Prolog errors 178
  8.3.1 Managing the error context 178
  8.3.2 Instantiation error 179
  8.3.3 Type error 179
  8.3.4 Domain error 179
  8.3.5 Existence error 179
  8.3.6 Permission error 179
  8.3.7 Representation error 180
  8.3.8 Evaluation error 180
  8.3.9 Resource error 180
  8.3.10 Syntax error 180
  8.3.11 System error 181
8.4 Calling Prolog from C 181
  8.4.1 Introduction 181
  8.4.2 Example: my_call/1 - a call/1 clone 182
  8.4.3 Example: recovering the list of all operators 184
8.5 Defining a new C main() function 185
  8.5.1 Example: asking for ancestors 186

References 188

Index 196
Index

!/0, 34, 35
./2, 121
( , )/2, 34
(-->)/2, 97
(->)/2, 34
( ; )/2, 34
( = )/2, 38
( = = )/2, 42
( = := )/2, 48
( = = = )/2, 39, 109
( = < )/2, 48
( = \ )/2, 48
( 0 = < )/2, 40
( 0 = < )/2, 40
( 0 = = )/2, 40
( 0 / \ )/2 (FD), 158
( 0 = )/2 (FD), 154
( 0 = = = )/2 (FD), 158
( 0 = = = )/2 (FD), 155
( 0 = < )/2 (FD), 154
( 0 = < = )/2 (FD), 155
( 0 = < # )/2 (FD), 155
( 0 = # )/2 (FD), 158
( 0 = # = )/2 (FD), 154
( 0 = # = = )/2 (FD), 158
( 0 = # = # )/2 (FD), 155
( 0 = # = # )/2 (FD), 155
( 0 = # = # )/2 (FD), 155
( 0 = # = )/2 (FD), 158
( 0 = # = )/2 (FD), 158
( 0 = # = )/2 (FD), 158
( 0 / \ )/2 (FD), 158
( 0 / \ )/2 (FD), 158
( 0 / = )/2 (FD), 154
( 0 / = = )/2 (FD), 158
( 0 / = # )/2 (FD), 155
( 0 / = # )/2 (FD), 155
( 0 / = # )/2 (FD), 155

(is)/2, 47
(< )/2, 48
( > )/2, 48
( >= )/2, 48
( \ + )/1, 100
( \ = )/2, 39
( \ = = )/2, 40
--assembly, 12
--aux-father, 16
--aux-father2, 16
--c-compiler, 13
--cmd-line, 16
--comment, 13
--compile-msg, 12
--cstr-size, 13
--encode, 16
--fast-math, 12, 47
--fd-to-c, 12
--fixed-sizes, 10, 13
--global-size, 13
--help, 12, 16
--keep-void-inst, 12
--local-size, 10, 13
--min-bips, 13
--min-fd-bips, 13
--min-pl-bips, 13
--min-reg-opt, 12
--min-size, 13
--mini-assembly, 12
--no-call-c, 12
--no-debugger, 13
--no-decode-hexa, 12
--no-del-temp, 12
--no-fd-lib, 13
--no-inline, 12
--no-opt-last-subterm, 12
--no-redef-error, 12
--no-reg-opt, 12
--no-reorder, 12
--no-singl-warn, 12
--no-susp-warn, 12
--no-top-level, 13
--object, 12
--output, 12
--pl-state, 12, 120
--printf, 16
--relax, 16
--statistics, 12
--strip, 13
--temp-dir, 12
--trail-size, 13
--verbose, 12
--version, 12, 16
--wam-for-byte-code, 12
--wam-for-native, 12
-A, 13
-C, 13
-F, 12
-H, 16
-L, 13
-M, 12
-P, 16
-S, 12
-W, 12
-c, 12
-h, 12
-o, 12
-s, 13
-v, 12
-v, 12
:- initialization/1, 14, 15, 33, 185
input (property), 61
integer/1, 37
integer_rounding_function (flag), 47, 118
interpreter, see top-level
jump (option), 165, 166
keysort/1, 113
keysort/2, 113
largest (FD option), 162
last/2, 111
last_modification (property), 132
last_read_start_line_column/2, 82
leash/1 (debug), 19, 20
length (property), 108
length/2, 111
line (option), 59, 68
line_count/2, 64, 65
line_position/2, 65
linedit, 8, 73, 146, 147
list/1, 37
list_or_partial_list/1, 37
listing/0, 122
listing/1, 21, 86, 122
load/1, 7, 11, 13, 121
loose (debug), 19
lower_upper/2, 103
MA, 10
make_directory/1, 128
max (FD option), 163
max arity (flag), 118
max_atom (flag), 106, 118
max_depth (option), 84
max_integer (flag), 118, 148
max_list/2, 112
max_regret (FD option), 162
max_stream (flag), 118
max_unget (flag), 74, 77, 118
member/2, 109
memberchk/2, 109
middle (FD option), 163
min (FD option), 163
min_integer (flag), 118
min_list/2, 112
mini-assembly, 2, 10, 16
mode (property), 61
most_constrained (FD option), 162
:- multifile/1, 30
name/2, 105
name_query_vars/2, 44
name_singleton_vars/1, 43, 86
namevars (option), 6, 44, 83
native_code (property), 53
needs_quotes (property), 108
needs_scan (property), 108
new_atom/1, 107
new_atom/2, 107
new_atom/3, 107
next (option), 44
nl/0, 75
nl/1, 75
nodebug/0 (debug), 19, 21
nom_fd_var/1 (FD), 152
nom_generic_var/1 (FD), 152
none (debug), 19
none (option), 59, 68, 165, 166
nomvar/1, 37
nospy/1 (debug), 19, 21
nospyall/0 (debug), 20
notrace/0 (debug), 19
nth/3, 112
number/1, 37
number_atom/2, 105
number_chars/2, 105
number_codes/2, 105
numbervars (option), 6, 44, 83
numbervars/1, 45, 86
numbervars/3, 45
once/1, 100
:- op/3, 32
op/3, 32, 87
open/3, 59
open/4, 55, 58, 67, 68, 141
open_input_atom_stream/2, 69
open_input_chars_stream/2, 69
open_input_codes_stream/2, 69
open_output_atom_stream/1, 71
open_output_chars_stream/1, 71
open_output_codes_stream/1, 71
os_error (flag), 118, 181
os_version/1, 135
output (property), 61
partial_list/1, 37
peek_byte/1, 77
peek_byte/2, 77
peek_char/1, 74
peek_char/2, 74
peek_code/1, 74
peek_code/2, 74
permission (property), 132
permutation/2, 110
phrase/2, 99
phrase/3, 99
popen/3, 55, 137
portray/1, 84, 87
portray_clause/1, 86
portray_clause/2, 86, 122
portrayed (option), 83
position (property), 61
socket_connect/4, 55, 143
socket_listen/2, 144
sort/1, 113
sort/2, 112
sort0/1, 113
sort0/2, 113
space_args (option), 83
spawn/2, 137
spawn/3, 137
spy/1 (debug), 19, 21
spypoint_condition/3 (debug), 20, 21
standard (FD option), 162
static (property), 52
statistics/0, 123
statistics/2, 123
stop/0, 99
stream_line_column/3, 65
stream_position/2, 62, 63
stream_property/2, 61, 62, 63, 67
strict_iso (flag), 24, 52, 118
string (token), 81
sub_atom/5, 102
sublist/2, 111
suffix/2, 110
sum_list/2, 123
syntax_error (flag), 79, 118, 181
syntax_error (option), 79
syntax_error_info/4, 82, 181
system/1, 136
system/2, 136
system_time/1, 123
tab/1, 96
tell/1, 94
telling/1, 95
temporary_file/3, 134
temporary_name/2, 133
term_ref/2, 45
text (option), 67
throw/1, 18, 25, 35, 182
tight (debug), 19
told/0, 95
top-level, 4, 8, 13, 15, 99, 146, 185
top_level/0, 99
trace/0 (debug), 7, 18
ttrue/0, 34
type (option), 58
type (property), 61, 132
unknown (permission), 132
unlink/1, 130
user (property), 53
user, 94, 95, 121, 125, 126
user_input, 55, 59, 94, 95
user_output, 55, 59, 94, 95
user_time/1, 123
value_method (FD option), 163
var (token), 81
var/1, 37
variable_method (FD option), 162
variable_names (option), 44, 79
variables (option), 79
vector_max (FD), 148, 150, 156
wait/2, 139
WAM, 2, 10, 11, 21
wam_debug/0 (debug), 19, 21
warning (option), 79
Warren Abstract Machine, see WAM
working_directory/1, 129
write (mode), 58
write (permission), 131
write/1, 84, 85
write/2, 84, 92–94
write_canonical/1, 84, 85
write_canonical/2, 84, 92–94
write_canonical_to_atom/2, 92
write_canonical_to_chars/2, 93
write_canonical_to_codes/2, 94
write_pl_state_file/1, 13, 120
write_term/2, 84
write_term/3, 6, 20, 83, 92–94
write_term_to_atom/3, 92
write_term_to_chars/3, 93
write_term_to_codes/3, 94
write_to_atom/2, 92
write_to_chars/2, 93
write_to_codes/2, 94
writeq/1, 84, 85
writeq/2, 84, 92–94, 173
writeq_to_atom/2, 92
writeq_to_chars/2, 93
writeq_to_codes/2, 94