Barry's Prolog
Reference Manual & User Guide

Version P1A02

Barry Watson

Contents

Chapter 1

Introduction

1.1 About

The Barry's Prolog Manual describes a mostly ISO compliant Prolog system which consists of an interpreter, a compiler, a debugger, and a collection of useful predicates. This manual tells you how to configure and start the system and provides a reference to all the library predicates. This manual is intended for any user of the system. It assumes you are familiar with Prolog programming.

1.2 Notation

1.2.1 Typeface

Throughout the manual examples of Prolog code or interactions with the Prolog system are indicated by text shown in a mono-width typeface. The following is an example of an interaction where the user instructs the system to print the string "Hello World" followed by a newline.

```
| ?- print('Hello World'), nl.
Hello World
% yes
```

Every now and then a line of text representing an interaction with the system is too long to be shown as one line in the manual. In this case, the line is broken up to make it look better on the page.

1.2.2 Predicate reference

Whenever a predicate needs to be referenced in the text, the predicate indicator is given. A predicate indicator is the name of the predicate and the arity of the predicate separated by a slash operator. For instance: `nl/0`,

`print/1`. Several predicates may share the same name, but the name and arity combination is unique.

1.2.3 Mode declaration

For each predicate described in this manual, a mode declaration is given. A mode declaration prefixes each predicate argument with one of the following operators: `+`, `-`, or `?`. As an example, here is the mode declaration for `atom_index/3`:

```
atom_index(+A, +I, ?C)
```

The following table shows the meaning of these operators:

`+` This argument is an input argument. It must not be an uninstantiated variable.

`-` This argument is an output argument. It must be an uninstantiated variable which will be instantiated by the predicate.

`?` This argument is either an input or an output argument. The may be an uninstantiated variable.

Chapter 2

Using Barry's Prolog

2.1 Getting Started

The system is started with the command `BarrysProlog`. When this command is given, you should see something similar to the following:

```
prolog@herbrand:~$ BarrysProlog
Prolog
Copyright (C) 2012-2014 Barry Watson. All rights reserved.
Abstract Machine version: P1A01.
Prolog version: P1A01.
Welcome!
% ~/.BarrysProlog file consulted.
| ?-
```

One of the tasks performed by the system when it is initialising is the loading of the file named `.BarrysProlog` which is found in your home directory. The contents of this file can be used to configure the system to your liking (load additional files, set flags, etc.). If this file doesn't exist, then the following will be seen instead:

```
prolog@herbrand:~$ BarrysProlog
Prolog
Copyright (C) 2012-2014 Barry Watson. All rights reserved.
Abstract Machine version: P1A01.
Prolog version: P1A01.
Welcome!
% No ~/.BarrysProlog file found.
| ?-
```

You should create this file and add at least the following lines to ensure trouble free usage:

```
:- add_file_search_path(runtime, '/opt/BarrysProlog/runtime').
:- add_file_search_path(examples, '/opt/BarrysProlog/examples').
:- ensure_loaded(runtime(elementary_functions)).
:- ensure_loaded(runtime(special_functions)).
% End of /home/prolog/.BarrysProlog
```

The first two lines assume that BarrysProlog has been installed in the directory `/opt` so you may need to change this to suit your installation (See the documentation for `add_file_search_path/2`). The last term read in any Prolog source file must be followed by whitespace (a newline, a tab, a space, or a comment). This is the reason behind the comment on the last line.

2.2 Queries

The prompt `| ?-` tells you that the system is ready for a query. The system will now input lines until it has read a full Prolog term followed by a full-stop (`.`). If more than one line needs to be read, then on subsequent lines the prompt is changed to `|`.

```
| ?- print(1),
|    print(2),
|    nl.
12
% yes
```

If the query contains any variables, these are reported upon success:

```
| ?- member(X, [one, two, three]).
X = one ?
```

The question mark shown is another prompt. You can now either type `;` followed by enter/return for more solutions:

```
| ?- member(X, [one, two, three]).
X = one ? ;

X = two ? ;

X = three ? ;

% no
```

Or you can just input enter/return for a new query prompt.

```
| ?- member(X, [one, two, three]).
X = one ?

% yes
```

2.3 Messages

Errors, warnings, and informational messages from the system are prefixed by one of the characters `!`, `*`, or `%`. You can also generate such messages using `print_message/2`.

```
| ?- print_message(error, 'This is an error').
! This is an error
% yes
| ?- print_message(warning, 'This is a warning').
* This is a warning
% yes
| ?- print_message(informational, 'This is information').
% This is information
% yes
```

2.4 Loading Prolog Files

You can load files of Prolog code with the predicates `consult/1`, `reconsult/1`, `ensure_loaded/1`, and `'.'/2`. See the manual pages for these predicates to find out which one suits your needs. Each of the loading methods will process directives which allow you to control the loading process. The rest of this section describes these directives.

2.4.1 `discontiguous/1`

The argument to this directive is a predicate indicator, a sequence of predicate indicators (separated by `','/2`), or a list of predicate indicators. This directive informs the system that the procedures indicated by the predicate indicators may be declared by a sequence of discontiguous clauses. This will prevent any warning messages being displayed. See the documentation for the Prolog flag `discontiguous_clauses_warnings` for more information. This indicator has no effect outside the file it is defined (unless the file is included in another file). Example:

```
:- discontiguous(foo/1).
```

2.4.2 `dynamic/1`

This is defined by the ISO standard but has no meaning when loading in Barry's Prolog. The compiler on the other hand will process this directive. See Chapter 15 for more information. Example:

```
:- dynamic(foo/1).
```

2.4.3 ensure_loaded/1

This has the effect of calling ensure_loaded/1 as soon as the directive has been read. This means that the loading of the current file may be paused whilst the file named in the argument of the directive is loaded (should it need to be). Example:

```
:- ensure_loaded(runtime(statistics)).
```

2.4.4 include/1

This has the effect of including the contents of the argument file into the current file. The use of an include directive should have the same effect as if the user had manually included the contents of the argument file into the file being processed. However, any errors or warnings generated in the included file will be reported with the correct file name and position. Example:

```
:- include('/home/prolog/code/useful_operator_definitions').
```

2.4.5 initialization/1

When the current load has been completed, the initialization directive's argument is called. A file can have several such directives. However, you should note that there is no guarantee given for the order in which the arguments of the directives are called. Example:

```
:- initialization(print('Loaded!')).
```

Should the goal fail then a warning message is printed. As an example, consider the following directive:

```
:- initialization(fail).
```

It generates the following message:

```
* The following goal failed - fail.
```

2.4.6 multifile/1

The argument to this directive is a predicate indicator, a sequence of predicate indicators (separated by ','/2), or a list of predicate indicators. This directive informs the system that the procedures indicated by the predicate indicators may be defined in more than one file. The effect is that reconsult/1 does not remove previous definitions of these procedures. This indicator has no effect outside the file it is defined (unless the file is included in another file).

```
:- multifile(foo/1).
```

2.4.7 `Prolog Term`

A Prolog term that does not have the form of the previously defined directives is called as soon as it is read. This is subtly different from the `initialization/1` directive which schedules its argument to be called when loading has completed. Example:

```
:- op(500, xfx, =>).
```

Should the goal fail then a warning message is printed. As an example, consider the following directive:

```
:- fail.
```

It generates the following message:

```
* The following goal failed - fail.
```

2.5 Interrupting Execution

At any time during execution you can input control-C to enter the interrupt handler. If the system is in the middle of a garbage collection, then the system will wait until garbage collection has completed before entering the interrupt handler. You will then be presented with a menu of options:

```
Interrupt
a - abort
c - continue
d - set debug flag. The debugger will leap.
e - exit Prolog. This is equivalent to a halt.
n - clear trace & debug flags.
t - set trace & debug flags. The debugger will creep.
x - exit Prolog. This is equivalent to a halt.
```

Pressing any one of the keys shown on the left of the menu will activate the choice. You do not need to input enter/return.

2.5.1 Abort

This choice is equivalent to a call of `abort/0`. This is useful if your code is stuck in an endless loop and you wish to abort the execution and jump back to the query prompt.

2.5.2 Continue

This choice simply exits the interrupt handler and continues execution from where it was interrupted.

2.5.3 Set debug flag

This choice is equivalent to a call of `debug/0`. This is useful if the debugger is loaded and you have spypoints set but debugging is turned off. Setting the flag `debug` will then mean that the next spypoint hit will enter the debugger.

2.5.4 Exit Prolog

This choice is equivalent to a call of `halt/0`.

2.5.5 Clear trace & debug flags

This choice is equivalent to a call of `nodebug/0`.

2.5.6 Set trace & debug flags

This choice is equivalent to a call of `trace/0`. This is useful if the debugger is loaded and you wish to see whereabouts in the code you are executing.

Chapter 3

The Prolog Language

3.1 Syntax

Input is written using the Unicode character set. The syntax rules which determine if a sequence of input characters is acceptable Prolog are a superset of those defined by the ISO standard. By far the most important syntax rule defines the term; it is either a variable, an atom, a number, or a compound.

Variables

There are two mutually exclusive types of variable which are distinguished according to the format of their names:

Anonymous A variable with the name `_`.

Named One of the following:

- A variable whose name starts with an upper case letter which is optionally followed by a sequence of characters where each character is either alphanumeric or `_`. The upper case letters that can start the name are those accepted by the built in predicate `prolog_lexical_upper_case_letter/1`.
- A variable whose name starts with `_` and is followed by a non-empty sequence of characters where each character is either alphanumeric or `_`.

The difference between these two types of variables is that two copies of the same named variable may refer to the same variable. However, two anonymous variables always refer to two different variables. Examples:

```
Foo
_foo
_
```

Atoms

An atom is a symbol whose name is comprised of a sequence of characters which conform to one of the following cases:

- A lower case letter optionally followed by a sequence of alphanumeric characters. The lower case letters that can start the name are those accepted by the built in predicate `prolog_lexical_lower_case_letter/1`.

- A sequence comprised solely of characters drawn from either the following set: `# $ & * / + - : . < = > ? @ \^ ~ ' ¡ ¢ £ ¤ ¥ ¦ § ¨ © « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ » ¼ ½ ¾ ¿` or a subset of all Unicode mathematical arrows and symbols. Each possible character is accepted by the built in predicate `prolog_lexical_symbol/1`. **Note:** The sequence `/*` is not a valid prefix for an atom name — it starts a comment.

- One of the following solo characters: `!` or `;`

- One of the following two-character sequences: `[]` or `{}`.

- Any sequence of characters within single quotes, e.g., `'example'`. If you want to have a single quote within the single quotes you must prefix it with another single quote, e.g., `'example''s'`.

- If the Prolog flag `double_quotes` is set to `atom`, then any sequence of characters within double quotes, e.g., `"example"`. If you want to have a double quote within the double quotes you must prefix it with another double quote, e.g., `"he said ""no"""`. See the documentation for the built-in predicates `current_prolog_flag/2`, and `set_prolog_flag/2`.

The character sequences contained within single or double quotes may contain escape codes depending on the value of the Prolog flag `char_escapes`. See the documentation for the built-in predicates `current_prolog_flag/2`, and `set_prolog_flag/2`. When permitted (set to `on`), the following list of escape codes can be used. Note that any characters given in lower case can also be given in upper case. Some of the definitions use the term whitespace which is defined later.

`\a` Replace with Unicode code 7 (bell).

`\b` Replace with Unicode code 8 (backspace).

`\d` Replace with Unicode code 127 (delete).

`\e` Replace with Unicode code 27 (escape).

`\f` Replace with Unicode code 12 (form feed).

`\n` Replace with Unicode code 10 (newline).

`\r` Replace with Unicode code 13 (return).

`\t` Replace with Unicode code 9 (horizontal tab).

`\v` Replace with Unicode code 11 (vertical tab).

`\\` Replace with Unicode code 97 (backslash - \).

`\c` Ignore all subsequent whitespaces up to the next non-whitespace character.

`\C` Here `C` is any whitespace character. It is ignored.

`\^?` Replace with Unicode code 127 (delete).

`\^C` Here `C` is a character whose Unicode code modulo 32 is the Unicode code of the escape sequence. These are the usual carat control codes for ASCII values 0 to 31.

`\OOO` Here `OOO` is a sequence of from 1 to 3 octal digits. Replace with the character represented by the Unicode code equal to the octal number.

`\uXXXXXX` Here `XXXXXX` is a sequence of from 1 to 6 hexadecimal digits. Replace with the character represented by the Unicode code equal to the hexadecimal number.

`\C` Here `C` is any character not defined as an escape character. The backslash is ignored and the entire escape sequence is replaced with `C`.

Numbers

A number can be of the type integer or floating-point. Both types are of arbitrary precision. See the documentation for the built-in predicate `eval/2`. A number can be given in any radix from 2 (binary) to 36. The radix digits used to build numbers are the usual digits `0` to `9`, and the alphabetical characters `a` to `z`, or `A` to `Z` (character case has no meaning). The alphabetical characters are used for radices 11 to 36.

An integer is an optional negative sign (`-`) followed by a sequence of radix digits. Examples:

```
123
-123
```

The format of a floating-point number is more involved than that of the integer. It is best described formally with a context free grammar. In the following, the `<radix-digit>` grammar class is undefined because it was defined above.

```
<floating-point-number> ::= <integer> <fraction> <exponent>
<floating-point-number> ::= <integer> <fraction>
<floating-point-number> ::= <fraction> <exponent>
<floating-point-number> ::= <fraction>
<integer> ::= "-" <radix-digits>
<integer> ::= <radix-digits>
<fraction> ::= "." <radix-digits>
<exponent> ::= "e" <signed-integer>
<exponent> ::= "E" <signed-integer>
<signed-integer> ::= "+" <radix-digits>
<signed-integer> ::= "-" <radix-digits>
<signed-integer> ::= <radix-digits>
<radix-digits> ::= <radix-digit>
<radix-digits> ::= <radix-digit> <radix-digits>
```

From this we see that only the fraction part must be given, all other parts are optional. Examples:

```
1.23
1.2e3
1.2e+3
-1.2E-3
.23
```

As mentioned above, numbers can be given in a radix other than decimal. The radix to be used is indicated by a prefix. Examples of numbers with radix prefixes:

```
0xdeadbeef
16'deadbeef
0b1011
2'1011
0o17
8'18
```

Of the examples just given, the first two were in radix 16 (hexadecimal), the next two were radix 2 (binary), and the last two were in radix 8 (octal). The prefixes `0b`, `0o`, and `0x` are defined by the ISO standard. The numerical prefixes predate the ISO standard and more flexible as you can provide a numerical prefix between 2 and 36 (inclusive). Note that floating-point numbers in radix 15 or higher that contain the radix digits `e` or `E` will not be interpreted correctly as `e` and `E` will be taken as the first character of the exponent part.

For ease of reading, the underscore is ignored in numbers, for example, one million in decimal can be input as `1_000_000`.

Compounds

A compound term is a symbol (an atom) immediately followed by a set of terms within parenthesis. Multiple terms within the parenthesis are separated by commas. Examples:

```
foo(bar).
numbers(1,2,3,4).
```

The leading symbol is called a functor. The outermost functor in a term is called the principle functor. The set of terms within the parenthesis are called arguments. The number of arguments is called the compound term's arity. In the above example, `foo` has arity `1`, and `numbers` has arity `4`. It is often necessary to describe a whole set of terms which share the same functor and arity. Prolog programmers usually denote such a set by writing functor and arity separated with a `/`, e.g., `foo/1`, `numbers/4`. This is known as a predicate indicator.

Lists

Like many programming languages, Prolog offers a list construction mechanism. A list is represented by a compound structure which by convention is the functor `'.'/2`. The first argument is the head of the list and the second argument is the tail. The empty list is represented by the atom `[]`. Example: the list of even integers greater than zero and less than ten would be represented as

```
'.'(2, '.'(4, '.'(6, '.'(8, []))))
```

As lists are so frequently used, a convenient syntax is provided. The term `'.'(A, B)` can be input and printed as `[A|B]`, and the term `'.'(A, '.'(B, C))` can be input and printed as `[A|[B|C]]`. More generally, the term

```
'.'(A, '.'(B, (..., '.'(Y, Z) ...)))
```

can be input and printed as

```
[A|[B|[...|[Y|Z]...]]]
```

This can be further shortened to the equivalent

```
[A, B, ..., Y|Z]
```

Finally, if the last list element is the empty list, then there is another shortcut; the term

```
[A, B, ..., Y|[]]
```

can be input and printed as

```
[A, B, ..., Y]
```

Some types of lists can be built using quotation marks. However, the contents of the list depends upon the value of the Prolog flag `double_quotes`. If the flag is set to `codes`, then the Prolog term

```
"ABCDEFG"
```

is equivalent to

```
[65,66,67,68,69,70,71]
```

The numbers are the Unicode codes of the corresponding letters. If the value of the Prolog flag `double_quotes` is set to `chars`, then the Prolog term

```
"ABCDEFG"
```

is equivalent to

```
['A','B','C','D','E','F','G']
```

See the documentation for the built-in predicates `current_prolog_flag/2`, and `set_prolog_flag/2`.

Whitespace

Terms can be separated by spaces, tabs, and other unprintable characters. These separators, a.k.a whitespace, have no meaning. Specifically, a whitespace character is a Unicode code of 32 or less, or a Unicode code greater than or equal to 127 and less than or equal to 160. Those Unicode codes which can be used as whitespace are those accepted by the built in predicate `prolog_lexical_ws/1`.

Comments

There are two types of comment. C style comments which start with `/*` and end with `*/`. Everything contained between these is treated as whitespace. Another type of comment starts with a `%` and anything between that character and the end of the current line is considered whitespace. Examples:

```
/* I am a
   comment. */
% As am I.
```

Operators

Those terms which have functors with arity of one or two, may be written in prefix, infix, or postfix notation. In this case these functors are called operators. Just like operators found in expressions in other languages, Prolog operators can be described in terms of priority (a.k.a. precedence) and associativity. See the documentation for the built in predicates `op/3` and `current_op/3` for details.

In the demonstrations that follow, we use calls of

```
op(Priority, Associativity, AtomName)
```

to specify the operator. The meaning of the three arguments are

`Priority` This is an integer between `0` and `1200` (inclusive). The lower of two priority values has a higher precedence, that is to say it binds more tightly. A call of `op/3` where `Priority` is `0` will remove the entry associated with `AtomName` and `Associativity`.

`Associativity` This argument is one of the following:

- `fx` A one argument non-associative prefix operator.
- `fy` A one argument right-associative prefix operator.
- `xfx` A two argument non-associative infix operator.
- `xfy` A two argument right-associative infix operator.
- `yfx` A two argument left-associative infix.
- `xf` A one argument non-associative postfix operator.
- `yf` A one argument left-associative postfix operator.

`AtomName` Depending on the number of arguments specified in `Associativity`, the operator is either `AtomName/1` or `AtomName/2`.

To demonstrate how operators work, let us take three functors, `f/2`, `g/2`, and `h/1`. Normally we would build a term from these like this:

```
f(g(1, 2), h(3))
```

We can define the `h/1` functor as a prefix operator

```
op(0, fx, h).  % Priority of zero removes existing operator h/1.
op(500, fx, h).
```

and write the very same term like this:

```
f(g(1,2), h 3)
```

In a similar way we could have defined the functor as a postfix operator

```
op(0, fx, h). % Remove the prefix operator h/1.
op(500, xf, h).
```

which would allow us to write this:

```
f(g(1,2), 3 h)
```

The functor `g/2` could be made infix

```
op(0, xfy, g).
op(600, xfy, g).
```

enabling us to write

```
f(1 g 2, 3 h)
```

The functor `f/2` could also be made infix

```
op(0, xfy, f).
op(700, xfy, f).
```

enabling us to write

```
1 g 2 f 3 h
```

By changing the fixity of the operators, we have completely transformed the input, but it still represents the same term.

To demonstrate how priority alters the parsing of input, consider the input

```
1 f 2 h
```

which with the definitions given above, represents the term

```
f(1,h(2))
```

Now, should we alter the priority of `h/1` from a level lower than `f/2` to a level higher than it, e.g.,

```
op(0, xf, h).
op(800, xf, h).
```

the term which the input represents changes to

```
h(f(1,2))
```

As we can see, the operator with the highest priority value becomes the principle functor. It is useful to know that parenthesis overrides priorities by giving the term enclosed a priority of zero. So, we can write

```
1 f (2 h)
```

to force `f/2` to be the principle functor, i.e, give us

```
f(1,h(2))
```

Note that the comma (`','`) that is used to separate functor arguments, or list elements, is actually an operator: `','/2` with priority `1000` and associativity `xfy`. This means that any term with a principle functor of `1000` or more in an argument or list will have to be written with parenthesis.

All that remains is to show how associativity alters the parsing of input. With the configuration of the operator `f/2` as infix right associative (`xfy`), the input

```
1 f 2 f 3
```

would represent the term

```
f(1,f(2,3))
```

Should we change `f/2` to be left associative instead,

```
op(0, xfy, f).
op(700, yfx, f).
```

then the same input would represent

```
f(f(1,2),3)
```

Note that if we change `f/2` to be non-associative (`xfx`), then the input in this example would lead to a syntax error.

The operators that are defined at system start-up are shown in Table 3.1.

3.2 Exceptions

Exceptions are used to report errors. Prolog code can signal an error with `throw/1` and handle an error with `catch/3`. The format of an error exception is `error(Error, Details)`. The term `Error` indicates the class of error and the term `Details` provides additional information intended to help pin-point failure. The predicate `print_message/2` in combination with `add_generate_message/1` can be used to print user-friendly error messages. What follows is a description of all the possible error exceptions:

`error(domain_error(A, B), domain_error(G, N, A, B))` Argument number `N` of the goal `G` was the term `B` which was not of domain `A`.

`error(evaluation_error(A), existence_error(G, N, A))` Argument number `N` of the goal `G` could not be evaluated due to reason `A`.

Priority	Associativity	Name
1200	xfx	:-
1200	xfx	-->
1200	fx	:-
1200	fx	?-
1159	fx	discontiguous
1159	fx	dynamic
1159	fx	ensure_loaded
1159	fx	include
1159	fx	initialization
1159	fx	multifile
1100	xfy	;
1050	xfy	->
1000	xfy	,
900	fy	\+
700	xfx	=
700	xfx	\=
700	xfx	==
700	xfx	\==
700	xfx	@<
700	xfx	@=<
700	xfx	@>
700	xfx	@>=
700	xfx	=..
700	xfx	is
700	xfx	=:=
700	xfx	=\=
700	xfx	<
700	xfx	=<
700	xfx	>
700	xfx	>=
500	yfx	+
500	yfx	-
500	yfx	/\
500	yfx	\/
400	yfx	*
400	yfx	/
400	yfx	//
400	yfx	div
400	yfx	rem
400	yfx	mod
400	yfx	<<
400	yfx	>>
200	xfx	**
200	xfy	^
200	fy	-
200	fy	\

Table 3.1: The Prolog operator table

`error(existence_error(A, B), existence_error(G, N, A, B))` Argument number `N` of the goal `G` was the object `B` of type `A` which does not exist.

`error(instantiation_error, instantiation_error(G, N))` Argument number `N` of the goal `G` was uninstantiated.

`error(lexical_error(M), lexical_error(G, A, B, M))` The goal `G` caused a lexical error in the file named `A` at position `B`. The term `M` is a descriptive message.

`error(permission_error(A, B, C), permission_error(G, A, B, C))` The action `A` on the object `C` of type `B` in goal `G` could not be performed due to lack of permission.

`error(representation_error(A), representation_error(G, N, A))` Argument number `N` of the goal `G` could not be represented as a value of the type `A`.

`error(resource_error(code), _)` There was no room left in the code section. The second argument is a dummy variable.

`error(resource_error(clause_store), _)` There was no room left in the clause store. The second argument is a dummy variable.

`error(resource_error(constants), _)` There was no room left in the constants table. The second argument is a dummy variable.

`error(resource_error(functors), _)` There was no room left in the functors table. The second argument is a dummy variable.

`error(resource_error(heap), _)` There was no room left on the heap (global stack). The second argument is a dummy variable.

`error(resource_error(local_stack), _)` There was no room left on the local stack. The second argument is a dummy variable.

`error(resource_error(pdl), _)` There was no room left on the PDL (Push Down List). The second argument is a dummy variable.

`error(resource_error(strings), _)` There was no room left in the strings table. The second argument is a dummy variable.

`error(resource_error(trail), _)` There was no room left on the trail. The second argument is a dummy variable.

`error(syntax_error(M), syntax_error(G, A, B, M, C))` The goal `G` caused a syntax error in the file named `A` at position `B`. The term `M` is a descriptive message, and `C` is the list of remaining input tokens.

`error(type_error(A, B), type_error(G, N, A, B))` Argument number `N` of the goal `G` was the term `B` which was not of type `A`.

Chapter 4

Built-in Predicates

4.1 Predicates

4.1.1 `abolish/1`

Synopsis

```
abolish(+PI)
```

Description

Removes the predicate identified by the predicate indicator `PI` from the clause store if it is dynamic. When you compile a predicate it becomes static and cannot be abolished.

Examples

```
| ?- assert(foo).
% yes
| ?- foo.
% yes
| ?- abolish(foo/0).
% yes
| ?- foo.
! ERROR
! Error class : existence error
! Goal in error : foo
! The predicate foo/0 does not exist.
```

Errors

`instantiation_error` `PI` must be ground.

`representation_error(max_arity) PI` must be a predicate indicator with an arity that is less than the value of the Prolog flag `max_arity`.

`domain_error(not_less_than_zero, Arity) PI` must be a predicate indicator with an arity that is not less than zero.

`type_error(integer, Arity) PI` must be a predicate indicator with an arity that is of type integer.

`type_error(atom, Name) PI` must be a predicate indicator with a valid name, that is an atom.

`permission_error(modify, static_procedure(PI))` The predicate to be abolished must not be static.

See also

`retract/1`, `retractall/1`.

4.1.2 `abort/0`

Synopsis

```
abort
```

Description

Exit the current break level. The initial break level cannot be exited. This predicate neither succeeds nor fails.

Examples

```
| ?- break.
% Entering break level 1.
[1]
| ?- abort.
% yes
| ?- abort.
| ?-
```

Errors

None.

See also

`break/0`.

4.1.3 absolute_file_name/2

Synopsis

```
absolute_file_name(+RF, -AF)
```

Description

Behaves as if it were defined as follows:

```
absolute_file_name(RF, AF) :-
        absolute_file_name(RF,
                           [file_type(prolog),
                            access(exist),
                            ignore_underscores(true),
                            file_errors(fail)],
                           AF),
        !.
absolute_file_name(RF, AF) :-
        absolute_file_name(RF, [], AF).
```

Examples

See absolute_file_name/3.

Errors

See absolute_file_name/3.

See also

absolute_file_name/3.

4.1.4 absolute_file_name/3

Synopsis

```
absolute_file_name(+RF, +Options, -AF)
```

Description

The relative file name `RF` is translated into the absolute file name `AF`. The argument `RF` is a term such that `source_sink(RF)` is true, and `AF` is an atom whose name is the corresponding absolute file name. The argument `Options` is a list of terms which control the translation process. Should conflicting options be given, the last such option in the list is given precedence. The list of valid option terms are as follows:

`access(A)` The access permission or existence of `AF` can be checked, or even ignored, with this option. If `Options` contains `file_errors(error)` then an exception will be thrown upon permission or existence errors, otherwise the call of `absolute_file_name/3` will fail. Here `A` can either be an atom or a list of atoms chosen from the following:

- `read` The file specified by `AF` must be readable.
- `write` The file specified by `AF` must be writable or at least it can be created if it does not exist.
- `append` The same as the option `write` but with the addition that the file position can be set to the end of the file.
- `exist` The file specified by `AF` must already exist.
- `none` The file system is not used, nothing is checked and no errors can occur.

`extensions(E)` The argument `E` is either an atom or a list of atoms. Each such atom is a file extension that should be tried when building `AF`. The atom with no name — `''` — signifies no extension. Should `RF` have an extension already, then this option has no effect. The options `extensions(E)` and `file_type(T)` may conflict with each other (and with multiples of themselves).

`file_errors(E)` Here `E` must be one of the following:

- `error` Any errors encountered will lead to an exception being thrown.
- `fail` Any errors encountered will lead to the failure of the call to `absolute_file_name/3`.

`file_type(T)` This option either specifies that we are dealing with a directory and not a normal file, or, it is a shorthand for commonly used file extensions which could be defined with the option `extensions(E)`. The options `extensions(E)` and `file_type(T)` may conflict with each other (and with multiples of themselves). The argument `T` must be one of the following:

- `text` Equivalent to no file extension.
- `prolog` Equivalent to `extensions(['fasl', 'pl', ''])`
- `fasl` Equivalent to `extensions('fasl')`
- `directory` This file type is the only way to tell `absolute_file_name/3` that we are dealing with a directory and not a normal file.

`ignore_underscores(I)` The translation process can be instructed to ignore underscores if it will lead to a successful translation. Here `I` must be one of the following:

`true` Any underscores in `RF` may be removed when translating to `AF`.

`false` Underscores cannot be removed from `RF`.

`solutions(S)` The determinicity of `absolute_file_name/3` is specified by this option. Here `S` must be one of the following:

`first` Make `absolute_file_name/3` deterministic, i.e., discard any other possible answers for `AF`.

`all` Make `absolute_file_name/3` non-deterministic, i.e., a later failure could give another answer for `AF`.

Examples

```
| ?- bagof(P,
           absolute_file_name('/f_b',
                              [file_type(prolog),
                               ignore_underscores(true),
                               solutions(all),
                               access(none)],
                              P),
           Ps).
P = _521856
Ps = [/f_b.fasl,/f_b.pl,/f_b,/fb.fasl,/fb.pl,/fb] ?

% yes
| ?- absolute_file_name('~/.login', [access(none)], P).
P = /home/bwat/.login ?

% yes
```

Errors

`instantiation_error` The arguments `RF` and `Options` were not both ground.

`domain_error(source_sink, RF)` The argument `RF` is not a source_sink term.

`domain_error(absolute_file_name_option, O)` An invalid option, `O`, was given.

`type_error(list, O)` The argument `Options` wasn't a proper list. Note it is not necessarily so that `O` will be the full argument `Options`.

`permission_error(access, read, AF)` The access permissions of the file specified by `AF` didn't include reading.

`permission_error(access, write, AF)` The access permissions of the file specified by `AF` didn't include writing.

`permission_error(access, append, AF)` The access permissions of the file specified by `AF` didn't include appending.

`existence_error(source_sink, AF)` The file specified by `AF` doesn't exist.

`permission_error(access, home_directory, UserName)` The home directory of the user `UserName` couldn't be accessed due to a lack of permission.

See also

`file_search_path/2`, `open/4`.

4.1.5 add_file_search_path/2

Synopsis

```
add_file_search_path(+A, +D)
```

Description

Creates the mapping between the path alias `A` and the directory `D` which is used by `file_search_path/2` which is in turn used by `absolute_file_name/2` to build file names.

Examples

```
| ?- add_file_search_path(logging_dir, '/usr/local/logs').
% yes
| ?- absolute_file_name(logging_dir('system.log'), A).
A = /usr/local/logs/system.log ?

% yes
```

Errors

`instantiation_error` Either the argument `A` or the argument `D` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

`current_file_search_path/2`, `del_file_search_path/2`.

4.1.6 add_generate_message/1

Synopsis

```
add_generate_message(+A)
```

Description

Registers the predicate identified by the predicate indicator `A/3` as a generate message hook. A generate message hook is a predicate with arity 3, the first argument being the input term, the second argument being the expansion of the input term, the third being unified with the atom `[]` when it is called. When a message is written by `print_message/2` , then the second argument is given to each generate message hook, one at a time in the order they were registered with `add_generate_message/1`. If one of the hooks succeeds then the message which was given as an argument is discarded and its expansion is used instead. If none of the hooks succeed, then the process just continues with the original message argument. The registration of term expansion hooks which are already registered has no effect.

The message that is generated by `A` should be either an empty list, or a list where the last element must be the atom `nl` and all of the other elements are of one of three forms (see `generate_message_lines/3`):

`Format-Args` These are written as if there was a call to `format(Stream, Format, Args)`.

`write_term(Term, Options)` . These are written as if there was a call to `write_term(Stream, Term, Options)`.

`nl` . This generates a newline as in a call to `nl/0`.

When displayed, all list elements between the `nl` elements are written on a separate line with the appropriate message severity prefix. See `print_message/2`.

Examples

Here we add a generate message hook to customise the message displayed when we catch an existence error in the top-level loop.

```
| ?- expand_term((gm(error(existence_error(predicate, F/N),
                           existence_error(Goal,
                                           not_applicable,
                                           predicate,
                                           F/N))) -->
                 ['OOOPS!!! ' - [], nl,
                  'Existence error'- [], nl,
                  'Goal in error : ~q' - [Goal], nl,
```

```
                'The predicate ~q does not exist.'-[F/N], nl]),
            Expansion),
assert(Expansion),
add_generate_message(gm).
F = _531728
N = _532496
Goal = _537184
Expansion = /* long line deleted ... */

% yes
| ?- foo(a,b,c,d).
! OOOPS!!!
! Existence error
! Goal in error : foo(a,b,c,d)
! The predicate foo/4 does not exist.
```

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

`current_generate_message/1`, `del_generate_message/1`, `expand_term/2`.

4.1.7 add_message_hook/1

Synopsis

`add_message_hook(+A)`

Description

Registers the predicate identified by the predicate indicator `A/3` as a message hook. A message hook is called by the procedure `print_message/2` with the first argument being the message severity, the second being the message, and the third being the list of message lines. The first hook that succeeds terminates the message printing process. If no such hook succeeds or no hook has been registered then `print_message/2` prints the lines onto the stream `user_error`.

Examples

```
| ?- assert((test(Severity, Msg, Lines) :-
        format('Severity: ~w~nMsg: ~w~nLines: ~w~n',
```

```
                [Severity, Msg, Lines]),
        print_message_lines(user_error, Severity, Lines))).
Severity = _620352
Msg = _622160
Lines = _623168 ?

% yes
| ?- add_message_hook(test).
Severity: informational
Msg: yes
Lines: [[~w-[yes]]]
% yes
Severity: query
Msg: []
Lines: []
| ?- print_message(error, 'I am an error.').
Severity: error
Msg: I am an error.
Lines: [[~w-[I am an error.]]]
! I am an error.
Severity: informational
Msg: yes
Lines: [[~w-[yes]]]
% yes
Severity: query
Msg: []
Lines: []
```

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

`current_message_hook/1`, `del_message_hook/1`, `print_message_lines/3`.

4.1.8 add_portray/1

Synopsis

`add_portray(+A)`

Description

Registers the predicate identified by the predicate indicator `A/2` as a portray hook. A portray hook is a predicate with arity 2, the first argument being a stream term, and the second argument the term to be portrayed. A call of `write_term(S, T, O)` where the list `O` contains the element `portray(true)` — such as in the case of `print/2` — will, for each subterm `U` of `T` in turn, search for a portray hook which succeeds when passed the stream term `S` and `U`. Should such a portray hook exist then the writing of `U` as a sub-process of the writing of `T` is complete. Should no such portray hook exist then `U` is written as it normally would be. Portray hooks are found in the order that they are registered with `add_portray/1`. The subsequent registration of an already registered hook has no effect.

Examples

```
| ?- assert((my_portray(S, foo) :- write(S, goo_not_foo))).
S = _524784 ?

% yes
| ?- print(foo), nl.
foo
% yes
| ?-  add_portray(my_portray).
% yes
| ?- print(foo), nl.
goo_not_foo
% yes
| ?- del_portray(my_portray).
% yes
| ?- print(foo), nl.
foo
% yes
```

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

`current_portray/1`, `del_portray/1`.

4.1.9 add_portray_message/1

Synopsis

`add_portray_message(+A)`

Description

Registers the predicate identified by the predicate indicator `A/2` as a portray message hook. A portray message hook, which is not to be confused with a portray hook, is a predicate with arity 2, the first argument being a severity term, and the second argument the message to be portrayed. Such a hook can be used to override the operation of print_message/2, which will call all registered hooks until one succeeds. Should no hook succeed then print_message/2, is written as it normally would be. Portray message hooks are found in the order that they are registered with `add_portray_message/1`. The subsequent registration of an already registered hook has no effect.

Examples

```
| ?- assert((ph(Severity, Message) :-
                print('Hook provided with '),
                print(Severity/Message),
                nl)).
Severity = _523024
Message = _525456 ?

% yes
| ?- add_portray_message(ph).
Hook provided with informational/yes
| ?- foo(a,b,c,d).
Hook provided with error/error(existence_error(predicate,foo/4),
                               existence_error(foo(a,b,c,d),
                                               not_applicable,
                                               predicate,
                                               foo/4))
```

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

current_portray_message/1, del_portray_message/1.

4.1.10 add_query_class_hook/1

Synopsis

```
add_query_class_hook(+A)
```

Description

Registers the predicate identified by the predicate indicator `A/5` as a query class hook. These hooks give the user an opportunity to define their own classes of queries. The mode specification for the hook is:
`hook(+QueryClass, +Prompt, +InputMethod, +MapMethod, +FailureMode)`.
The five arguments of the hook are:

`QueryClass` A term used to identify the class being described by the hook. If the first argument to `ask_query/4` is unifies with `QueryClass` then the query class described by this hook is selected.

`Prompt` A term shown to the user as an indication that input is being requested.

`InputMethod` See `query_input/3`

`MapMethod` See `query_map/4`

`FailureMode` See `ask_query/4`.

Examples

```
| ?- assert(swedish_query_class_hook(swedish_yes_no_maybe,
                                     ' (j, n, eller k) ',
                                     line,
                                     char([ja-"Jj",
                                           nej-"Nn",
                                           kanske-"Kk"]),
                                     help_query)).
% yes
| ?- add_query_class_hook(swedish_query_class_hook).
% yes
| ?- ask_query(swedish_yes_no_maybe, [] ,[], Input).
 (j, n, eller k) K
Input = kanske ?

% yes
```

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

`current_query_class_hook/1`, `del_query_class_hook/1`.

4.1.11 add_query_input_hook/1

Synopsis

`add_query_input_hook(+A)`

Description

Registers the predicate identified by the predicate indicator `A/3` as a query input hook. These hooks give the user an opportunity to define their own query input methods. An example of query input hook could be `hook(InputMethod, Prompt, RawInput)`:

`InputMethod` This is the term used to identify this input hook. A query class would use this term to select this query input hook.

`Prompt` This is the term that should shown to the user as an indication that input is being requested.

`RawInput` The input which this hook has read.

Examples

In this example we define three different query hooks: class, map, and input. Together they allow us to prompt the user for a single character code which is then mapped to a predetermined set of answer terms.

```
| ?- assert((code_input_hook(code, Prompt, RawInput) :-
                      prompt(OldPrompt),
                      prompt(Prompt),
                      catch(get_code(RawInput),
                            E,
                            (prompt(OldPrompt),
                             throw(E))),
                      prompt(OldPrompt))).
Prompt = _527312
RawInput = _529296
OldPrompt = _534016
```

```
E = _546736 ?

% yes
| ?- add_query_input_hook(code_input_hook).
% yes
| ?- assert((code_map(code(Pairs), RawInput, Result, Answer) :-
                     member(Name-Abbreviation, Pairs),
                     member(RawInput, Abbreviation),
                     !,
                     Answer = Name,
                     Result = success)).
Pairs = _525536
RawInput = _527488
Result = _529888
Answer = _531872
Name = _536176
Abbreviation = _537616 ?

% yes
| ?- add_query_map_hook(code_map).
% yes
| ?- assert(ynm_class_hook(yes_no_maybe,
                     ' (y, n, or m) ',
                     code,
                     code([yes-[89,121],
                           no-[78,110],
                           maybe-[77,109]]),
                     help_query)).
% yes
| ?- add_query_class_hook(ynm_class_hook).
% yes
| ?- ask_query(yes_no_maybe, [] ,[], Input).
 (y, n, or m) m
Input = maybe
% yes
```

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

ask_query/4, current_query_input_hook/1, del_query_input_hook/1, query_class/5, query_input/3, query_map/4.

4.1.12 add_query_map_hook/1

Synopsis

```
add_query_map_hook(+A)
```

Description

Registers the predicate identified by the predicate indicator `A/4` as a query map hook. The purpose of query mapping is to translate (map) user input. A hook lets the user define their own such mappings. An example of query map hook could be
`hook(MapMethod, RawInput, Result, Answer)`:

`MapMethod` is a term used to identify the mapping being described by the hook. The query class selects the mapping method.

`RawInput` is the term to be mapped.

`Result` is the result of the mapping.

`Answer` is unified with the atom `success` upon a successful mapping. Any other atom would indicate failure.

Examples

See add_query_input_hook/1 for an example.

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

ask_query/4, current_query_map_hook/1, del_query_map_hook/1, query_class/5.

4.1.13 add_term_expansion/1

Synopsis

```
add_term_expansion(+A)
```

Description

Registers the predicate identified by the predicate indicator `A/2` as a term expansion hook. A term expansion hook is a predicate with arity 2, the first argument being the input term, and the second argument being the expansion of the input term. When a term is read by `consult/1`, `reconsult/1`, or the top-level loop, then that term is given to each term expansion hook, one at a time in the order they were registered with `add_term_expansion/1`. If one of the hooks succeeds then the term which was read is discarded and its expansion is used instead. If none of the hooks succeed, then the process just continues with the input term. The term expansion phase occurs before the DCG expansion phase. The registration of term expansion hooks which are already registered has no effect.

Examples

```
| ?- assert((foo :- print(foo), nl)).
% yes
| ?- assert((goo_not_foo :- print(goo), nl)).
% yes
| ?- assert(my_expander(foo, goo_not_foo)).
% yes
| ?- foo.
foo
% yes
| ?- add_term_expansion(my_expander).
% yes
| ?- foo.
goo
% yes
| ?- del_term_expansion(my_expander).
% yes
| ?- foo.
foo
% yes
```

Errors

`instantiation_error` The argument `A` was not instantiated.

`type_error(atom, A)` The argument `A` was not an atom.

See also

`current_term_expansion/1`, `del_term_expansion/1`, `expand_term/2`, `term_expansion/2`.

4.1.14 append/3

Synopsis

```
append(?T1, ?T2, ?T3)
```

Description

Succeeds if all of the arguments are lists and `T3` is equal to the concatenation of `T1` and `T2`. This predicate is the usual `append/3` — a.k.a. `conc/3` — found in Prolog textbooks.

Examples

```
| ?- append([a,b,c], [d,e,f], L).
L = [a,b,c,d,e,f] ?

% yes
| ?- append(L, [d,e,f], [a,b,c,d,e,f]).
L = [a,b,c] ?

% yes
```

Errors

None.

See also

None.

4.1.15 apply/2

Synopsis

```
apply(?T1, ?T2)
```

Description

Behaves as if it were defined as follows:

```
apply(Functor, Args) :-
        Call =.. [Functor|Args],
        call(Call).
```

Examples

```
| ?- X = print, apply(X, [99]), nl.
99
X = print ?

% yes
```

Errors

`instantiation_error` One of the arguments, `T1` or `T2`, was not instantiated.

`type_error(list, T2)` The argument `T2` was not a list.

`domain_error(non_empty_list, [])` The argument `T2` was the empty list `[]`.

See also

`call/1`.

4.1.16 arg/3

Synopsis

`arg(+I, +T, ?X)`

Description

`X` is the $\texttt{I}^{\text{th}}$ argument of the compound term `T`. This predicate will fail if `I` is zero or if `I` is greater than the arity of the functor of `T`.

Examples

```
| ?- arg(1, foo(a,b), Arg).
Arg = a ?

% yes
```

Errors

`instantiation_error` Either `I` or `T` were not instantiated.

`domain_error(not_less_than_zero, I)` The argument `I` was less than zero.

`type_error(compound, T)` The argument `T` was not a compound term.

See also

None.

4.1.17 Arithmetic comparison with evaluation

Synopsis

```
+L =:= +R
+L =\= +R
+L < +R
+L =< +R
+L > +R
+L >= +R
```

Description

Arithmetic comparison of arithmetic expressions `L` and `R`. The expressions `L` and `R` are evaluated with `eval/2` and then compared.

`L =:= R` `L` equal to `R`.

`L =\= R` `L` not equal to `R`.

`L < R` `L` less than `R`.

`L =< R` `L` less than or equal to `R`.

`L > R` `L` greater than `R`.

`L >= R` `L` greater than or equal to `R`.

Examples

```
| ?- 3.3 > 3.
% yes
| ?- 3.0 =:= 3.
% yes
| ?- 4 =< 5.
% yes
| ?- 5*4 =:= 10*2.
% yes
```

Errors

Errors are not thrown by these predicates. The evaluation of the arguments may however throw errors. See `eval/2`.

See also

eval/2.

4.1.18 Arithmetic comparison without evaluation

Synopsis

```
equal(+N1, +N2)
less_than(+N1, +N2)
greater_than(+N1, +N2)
less_than_equal(+N1, +N2)
greater_than_equal(+N1, +N2)
```

Description

Arithmetic comparison of numbers `N1` and `N2`. These predicates do *not* evaluate their arguments.

Examples

```
| ?- equal(3.0, 3).
% yes
| ?- less_than(4, 99).
% yes
| ?- greater_than(5, 4).
% yes
| ?- less_than_equal(3, 3).
% yes
| ?- greater_than_equal(4, -4).
% yes
```

Errors

None.

See also

None.

4.1.19 `arity/2`

Synopsis

```
arity(+Term, ?Arity)
```

Description

Succeeds if the arity of the principle functor of `Term` is `Arity`. This predicate behaves as if it were defined as:

```
arity(Term, N) :-
        functor(Term, _, N).
```

Examples

```
| ?- arity(9,0).
% yes
| ?- arity(foo(1,2,3), X).
X = 3 ?

% yes
```

Errors

See `functor/3`.

See also

`functor/3`.

4.1.20 ask_query/4

Synopsis

```
ask_query(+Class, +Query, +Help, -Answer)
```

Description

Prompts the user for input, reads input, and possibly provides help upon input failure. The body of `ask_query/4` is essentially the following:

```
query_class(Class, Prompt, InputMethod, MapMethod, FailureMode),
generate_message_lines(HelpLines, Help, []),
generate_message_lines(QueryLines, Query, []),
message_hook(query, Query, QueryLines),
query_input(InputMethod, Prompt, Input),
query_map(MapMethod, Input, Result, Answer),
```

Should `Result` be `success`, then `ask_query/4` has finished. Otherwise, behaviour depends upon `FailureMode` where the possible values are:

`none` Control loops back to `query_input/3`.

`query` Control loops back to `message_hook/3` redisplaying the query text.

`help` The help text is displayed with `message_hook(help, Help, HelpLines)` and control loops back to `query_input/3`.

`help_query` The help text is displayed with `message_hook(help, Help, HelpLines)` then control loops back to `message_hook/3` redisplaying the query text.

Examples

```
| ?- ask_query(query, [], [], Input).
| ?- term(1,2,3).
Input = term(1,2,3)-[] ?

% yes
| ?- ask_query(yes_or_no,
                [’Reboot?’-[], nl],
                [’Please answer yes or no.’-[], nl],
                Answer).
Reboot? (y or n) m

Please answer yes or no.
Reboot? (y or n) n

Answer = no ?

% yes
```

Errors

None.

See also

None.

4.1.21 `assert/1`

Synopsis

`assert(+C)`

Description

Equivalent to `assertz(C)`.

Examples

See assertz/1.

Errors

None.

See also

asserta/1.

4.1.22 asserta/1

Synopsis

```
asserta(+C)
```

Description

Adds the clause `C` to the clause store. The first clause of the relevant procedure will be `C`. To ensure that all clauses have a body term, if `C` is an atom or a compound term with a principle functor different from `':-'/2`, then the argument is rewritten to be `C :- true`.

Examples

```
| ?- asserta(foo(2)).
% yes
| ?- asserta(foo(1)).
% yes
| ?- foo(X), print(X), nl, fail ; print(done), nl.
1
2
done
X = _521408 ?

% yes
```

Errors

`instantiation_error` The head of the argument `C` was not instantiated.

`type_error(callable, Head)` The head of the argument `C` was not callable, that is to say it was not a predication.

`type_error(callable, Body)` The body of the argument `C` was not callable, that is to say not a well formed body term.

`permission_error(modify, static_procedure, Proc)` The procedure to be modified was static.

See also

`assert/1`, `assertz/1`, `predication/1`, `well_formed_body_term/1`.

4.1.23 `assertz/1`

Synopsis

```
assertz(+C)
```

Description

Adds the clause `C` to the clause store. The last clause of the relevant procedure will be `C`. To ensure that all clauses have a body term, if `C` is an atom or a compound term with a principle functor different from `':-'/2`, then the argument is rewritten to be `C :- true`.

Examples

```
| ?- assertz(foo(2)).
% yes
| ?- assertz(foo(1)).
% yes
| ?- foo(X), print(X), nl, fail ; print(done), nl.
2
1
done
X = _521408 ?

% yes
```

Errors

`instantiation_error` The head of the argument `C` was not instantiated.

`type_error(callable, Head)` The head of the argument `C` was not callable, that is to say it was not a predication.

`type_error(callable, Body)` The body of the argument `C` was not callable, that is to say not a well formed body term.

`permission_error(modify, static_procedure, Proc)` The procedure to be modified was static.

See also

assert/1, asserta/1, predication/1, well_formed_body_term/1.

4.1.24 at_end_of_stream/0

Synopsis

```
at_end_of_stream
```

Description

Equivalent to at_end_of_stream(user_input).

Examples

See at_end_of_stream/1.

Errors

See at_end_of_stream/1.

See also

at_end_of_stream/1.

4.1.25 at_end_of_stream/1

Synopsis

```
at_end_of_stream(+S)
```

Description

Succeeds if the stream identified by the stream term S has the property end_of_stream(at), i.e., the stream position for S is at the end of the stream and there are no more data to be read. S must be a stream term or a stream alias.

Examples

```
skip_to_end_of_text_file(Stream) :-
        repeat,
        get_char(Stream, _),
        at_end_of_stream(Stream),
        !.
```

Errors

`instantiation_error` The argument `S` was not instantiated.

`domain_error(stream_or_alias, S)` The argument `S` was not a valid stream term or atom.

`existence_error(stream, S)` The argument `S` did not identify an open stream.

See also

`at_end_of_stream/0`, `stream_alias/2`.

4.1.26 `atom/1`

Synopsis

```
atom(+T)
```

Description

Succeeds if `T` is an atom.

Examples

```
| ?- atom(atom).
% yes
| ?- atom(atom(atom)).
% no
| ?- atom(99).
% no
```

Errors

None.

See also

None.

4.1.27 `atom_chars/2`

Synopsis

```
atom_chars(+A, ?L)
atom_chars(?A, +L)
```

Description

Succeeds if the name of the atom `A` corresponds to the list of characters `L`.

Examples

```
| ?- atom_chars(foo, C).
C = [f,o,o] ?

% yes
| ?- atom_chars('', C).
C = [] ?

% yes
| ?- atom_chars(A, [a,t,o,m]).
A = atom ?

% yes
```

Errors

`instantiation_error` Either both arguments were uninstantiated, or, `A` was uninstantiated and `L` was not ground.

`type_error(atom, A)` The argument `A` was instantiated but it was not an atom.

`type_error(list, L)` The argument `L` was instantiated but it was not a list.

`type_error(character, E)` The argument `L` contained an element `E` which was not a character.

See also

atom_codes/2, character/1.

4.1.28 atom_codes/2

Synopsis

```
atom_codes(+A, ?L)
atom_codes(?A, +L)
```

Description

Succeeds if the name of the atom `A` corresponds to the list of character codes `L`.

Examples

```
| ?- atom_codes(foo, L).
L = [102,111,111] ?

% yes
| ?- atom_codes(A, [102, 103, 104]).
A = fgh ?

% yes
| ?- atom_codes('', []).
% yes
```

Errors

`instantiation_error` Either both arguments were uninstantiated, or, `A` was uninstantiated and `L` was not ground.

`type_error(atom, A)` The argument `A` was instantiated but it was not an atom.

`type_error(list, L)` The argument `L` was instantiated but it was not a list.

`representation_error(character_code)` The argument `L` contained an element which was not a character code.

`domain_error(code_list_not_too_long, L)` The argument `L` contained too many elements.

See also

`atom_chars/2`, `character_code/1`, `name/2`.

4.1.29 `atom_concat/3`

Synopsis

```
atom_concat(+T1, +T2, ?T3)
atom_concat(?T1, ?T2, +T3)
```

Description

The name of atom `T3` is a concatenation of the names of the atoms `T1` and `T2` — in that order.

Examples

```
| ?- atom_concat(hello, world, A).
A = helloworld ?

% yes
| ?- atom_concat(A, world, helloworld).
A = hello ?

% yes
```

Errors

`instantiation_error` Either, both `T1` and `T3` are uninstantiated, or, both `T2` and `T3` are uninstantiated.

`type_error(atom, T)` An instantiated argument `T` was not a concatable atom.

See also

`concatable_atom/1`.

4.1.30 atom_index/3

Synopsis

```
atom_index(+A, +I, ?C)
```

Description

Succeeds if the $\texttt{I}^{\text{th}}$ character code of the name of the atom `A` is `C`. The first character code is indexed by the integer 0.

Examples

```
% yes
| ?- atom_index(foo ,1, C).
C = 111 ?

% yes
| ?- atom_index(foo, 0, C).
C = 102 ?

% yes
```

Errors

`instantiation_error` At least one of the arguments `A` or `I` was not instantiated.

`type_error(integer, I)` The argument `I` was not an integer.

`domain_error(not_less_than_zero, I)` The argument `I` was less than zero.

See also

None.

4.1.31 `atom_length/2`

Synopsis

```
atom_length(+T, ?N)
```

Description

The number of characters in the name of atom `T` is the integer `N`.

Examples

```
| ?- atom_length(atom_length, N).
N = 11 ?

% yes
| ?- atom_length('', N).
N = 0 ?

% yes
```

Errors

`instantiation_error` The argument `T` was not instantiated.

`type_error(atom, T)` The argument `T` was not an atom.

`type_error(integer, N)` The argument `N` was instantiated and not an integer.

`domain_error(not_less_than_zero, I)` The argument `N` was instantiated and was an integer less than zero.

See also

None.

4.1.32 atomic/1

Synopsis

```
atomic(+T)
```

Description

Succeeds if the term `T` is atomic. This definition of atomic term is:

(i) all atoms are atomic terms, and

(ii) all integers are atomic terms.

Note that floating-point numbers are *not* atomic, the reason being that they are represented by compound terms.

Examples

```
| ?- atomic(atomic).
% yes
| ?- atomic(99).
% yes
| ?- atomic(99.0).
% no
```

Errors

None.

See also

`atom/1`, `integer/1`.

4.1.33 between/3

Synopsis

```
between(+Low, +High, ?Value)
```

Description

Succeeds if `Low =< Value` and `Value =< High`. This predicate behaves as if it were defined as follows:

```
between(Low, High, Low) :-
        Low =< High.
between(Low, High, Value) :-
        Low < High,
```

```
        NextLow is Low + 1,
        between(NextLow, High, Value).
```

Examples

```
| ?- between(1, 2, X), print(X), nl, fail ; print(done), nl.
1
2
done
X = _25248 ?

% yes
```

Errors

Since the arguments are evaluated, errors may be raised by `eval/2`.

See also

None.

4.1.34 bagof/3

Synopsis

`bagof(+Template, +Goal, ?Bag)`

Description

`Bag` is a list representing a multiset of instances of `Template` which satisfy `Goal`. Should `Goal` be unsatisfiable, `bagof/3` fails. For this to be useful you will want at least one uninstantiated variable in `Template` to be free in `Goal`. Those uninstantiated variables which are part of `Goal` but not part of `Template` are considered to be universally quantified and thus may give rise to several solutions for the `bagof/3` call. As an example of this, the following shows a `bagof/3` query giving 3 solutions:

```
% yes
| ?- bagof(X, member(X-Y, [1-2,3-4,5-6]), L).
X = _521856
Y = 2
L = [1] ? ;

X = _521856
Y = 4
L = [3] ? ;
```

```
X = _521856
Y = 6
L = [5] ? ;

% no
```

It is possible to avoid these "extra" solutions by existentially quantifying these uninstantiated free variables that are in `Goal` but not in `Template`. Variables are existentially quantified with the `'^'/2` operator. Here we alter the previous example and existentially quantify the variable `Y`.

```
| ?- bagof(X, Y^member(X-Y, [1-2,3-4,5-6]), L).
X = _521856
Y = _522720
L = [1,3,5] ?

% yes
```

Examples

See description above.

Errors

`instantiation_error` The argument `Goal` was uninstantiated.

`type_error(callable, Goal)` The argument `Goal` was not callable.

`type_error(list, Bag)` The argument `Bag` was neither a list nor a partial list.

See also

`findall/3`, `setof/3`.

4.1.35 `break/0`

Synopsis

`break`

Description

Initiates a new top-level loop. The debugging flags are cleared before starting the new top-level loop and they are restored upon an abort.

Examples

```
| ?- break.
% Entering break level 1.
[1]
| ?- break.
% Entering break level 2.
[2]
| ?- abort.
% yes
[1]
| ?- abort.
% yes
```

Errors

None.

See also

abort/0.

4.1.36 byte/1

Synopsis

```
byte(+B)
```

Description

Succeeds if `B` is a valid byte. Behaves as if it were defined as follows:

```
byte(Byte) :-
        integer(Byte),
        Byte >= 0,
        Byte =< 255.
```

Examples

```
| ?- byte(199).
% yes
```

Errors

None.

See also

None.

4.1.37 'C'/3

Synopsis

```
'C'(?A, ?B, ?C)
```

Description

This predicate is provided for historical reasons as some pieces of Prolog code expect it to be present. 'C'/3 behaves as if it were defined as follows.

```
'C'([H|T], H, T).
```

Examples

```
| ?- 'C'([one, two, three], one, Rest).
Rest = [two,three] ?

% yes
```

Errors

None.

See also

None.

4.1.38 call/1

Synopsis

```
call(+T)
```

Description

Invokes the interpreter on the term T.

Examples

```
| ?- call(true).
% yes
| ?- call(fail).
% no
```

```
| ?- call(call((print(call), nl))).
call
% yes
```

Errors

`instantiation_error` The argument `T` was not instantiated.

`type_error(callable, T)` The argument `T` was not a callable term.

See also

`callable_term/1`.

4.1.39 call/3

Synopsis

```
call(+T1, +T2, +T3)
```

Description

Calls the term formed by taking the callable term `T1` and adding the terms `T2` and `T3` as the new last two arguments. So if `T1` is a term with functor `F` and arity `N`, then the new term would have the predicate indicator `F/(N+2)`. This predicate is used in code generated by Definite Clause Grammar expansion; it behaves as if defined as follows:

```
call(Pred, Arg1, Arg2) :-
        Pred =.. [Functor|Args],
        append(Args, [Arg1, Arg2], NewArgs),
        New_pred =.. [Functor|NewArgs],
        call(New_pred).
```

Examples

```
| ?- expand_term((a --> call(b)), T).
T = a(_544096,_544064) :- call(b,_544096,_544064) ?

% yes
```

Errors

`instantiation_error` The argument `T1` was not instantiated.

See also

`call/1`.

4.1.40 callable_term/1

Synopsis

```
callable_term(+T)
```

Description

Succeeds if `T` is a callable term. This definition of callable term is:

(i) all atoms are callable terms, and

(ii) all compound terms are callable terms.

Examples

```
| ?- callable_term(foo).
% yes
| ?- callable_term(99).
% no
| ?- callable_term(T).
% no
```

Errors

None.

See also

`call/1`.

4.1.41 catch/3

Synopsis

```
catch(+T1, +T2, +T3)
```

Description

Sets up an environment for the duration of a call of `T1` whereby any unhandled call to `throw(Obj)` where `T2=Obj` is handled by a call to `T3` — subject any variable bindings introduced by `T2=Obj`. This means `catch/3` can succeed in two ways:

(i) `call(T1)` succeeds, or

(ii) `call(T1)` throws an unhandled exception with `throw(Obj)` where `T2=Obj` and `(T2=Obj, call(T3))` succeeds.

Should it be the case that `T2=Obj` fails, then the throw is unhandled and any outer `catch/3` environments are checked for successful handling. Should `call(T1)` or `call(T3)` fail, then `call/3` fails.

Examples

```
| ?- catch(true, object, (print(cought), nl)).
% yes
| ?- catch(fail, object, (print(cought), nl)).
% no
| ?- catch(throw(object), object, fail).
% no
```

Errors

`instantiation_error` The argument `T1` was not instantiated.

`type_error(callable, T1)` The argument `T1` was not a callable term.

See also

`throw/1`.

4.1.42 char_code/2

Synopsis

```
char_code(+Char, ?Code)
char_code(?Char, +Code)
```

Description

Succeeds if the one atom character `Char` has the character code `Code`.

Examples

```
| ?- char_code(Char, 65).
Char = A ?

% yes
| ?- char_code('B', Code).
Code = 66 ?

% yes
```

Errors

`instantiation_error` Both arguments were uninstantiated.

`type_error(character, Char)` The argument `Char` was not a character.

`type_error(integer, Code)` The argument `Code` was not an integer.

`representation_error(character_code)` The argument `Code` was not a valid character code.

See also

`character/1`, `character_code/1`.

4.1.43 char_conversion/2

Synopsis

```
char_conversion(+From, +To)
```

Description

Updates the character conversion table so that `current_char_conversion/2` can map the character `From` to `To`. When called with `From` being identical to `To` — the identity mapping — any character conversion table entry for the argument character is removed. The conversion table is used when unquoted terms are read by `read_term/3`. It is always a good idea to quote the arguments to `char_conversion/2` so as to avoid any conversion problems.

Examples

Note how quotes are used with arguments and with the functor name to avoid problems.

```
| ?- char_conversion('a', 'a').
% yes
| ?- char_conversion('a', 'b').
% yes
| ?- 'current_char_conversion'('a', To).
To = b ?

% yes
| ?- 'char_conversion'('a', 'a').
% yes
| ?- current_char_conversion('a', To).
% no
```

Errors

`instantiation_error` One of the arguments was not instantiated.

`representation_error(character)` One of the arguments was not a character.

See also

`character/1`, `current_char_conversion/2`.

4.1.44 `character/1`

Synopsis

```
character(+T)
```

Description

Succeeds if `T` is a character.

Examples

```
| ?- character(99).
% no
| ?- character([]).
% no
| ?- character(c).
% yes
```

Errors

None.

See also

None.

4.1.45 `character_code/1`

Synopsis

```
character_code(+T)
```

Description

Succeeds if `T` is a character code.

Examples

```
| ?- character_code(99).
% yes
| ?- character_code([]).
% no
| ?- character_code(c).
% no
```

Errors

None.

See also

None.

4.1.46 clause/2

Synopsis

`clause(+H, ?B)`

Description

Succeeds if the term `H :- B` has been asserted into the clause store. Only dynamic — a.k.a. public — predicates are found in the clause store. Predicates which are static — a.k.a. private — cannot be retrieved with `clause/2`.

Examples

```
| ?- assert(foo).
% yes
| ?- clause(foo, true).
% yes
```

Errors

`instantiation_error` The argument `H` was not instantiated.

`type_error(callable, H)` The argument `H` was not a callable term.

`type_error(callable, B)` The argument `B` was neither a variable nor a callable term.

`permission_error(access, private_procedure, F/N)` The argument `H` was the head term of a private procedure which has the predicate indicator `F/N`.

See also

`callable_term/1`.

4.1.47 `close/1`

Synopsis

```
close(+S)
```

Description

Equivalent to `close(S,[])`. See `close/2`.

Examples

See `close/2`.

Errors

See `close/2`.

See also

`close/2`.

4.1.48 `close/2`

Synopsis

```
close(+S, +O)
```

Description

Closes the stream `S` according to the list of options `O`. The possible members of the options list are:

- `force(false)` If there is an error upon closing, then the stream is not closed.
- `force(true)` The stream is always closed, regardless of whether closing is in error or not. If there is an error upon closing, the stream contents may be in an invalid state.

The default option in the case `O=[]`, is `[force(false)]`. In the event that `O` contains conflicting options, the last such option takes precedence.

Examples

```
| ?- open('foo.txt', write, Stream0), close(Stream0, []).
Stream0 = $stream(3) ?

% yes
```

Errors

`instantiation_error` The argument `S` was not instantiated, or, the argument and `O` was not ground.

`existence_error(stream, S)` The argument `S` was not an open stream.

`domain_error(stream_or_alias, S)` The argument `S` was neither an atom nor a stream term.

`type_error(list, O)` The argument `O` was not a list.

`domain_error(close_option, Option)` The argument `O` was a list which contained an invalid option `Option`.

See also

`close/1`.

4.1.49 compare/3

Synopsis

```
compare(?A, +L, +R)
```

Description

Succeeds if the relation `A` holds between the terms `L` and `R`. This predicate behaves as if it were defined as follows:

```
compare(<, A, B) :- A @< B, !.
compare(>, A, B) :- A @> B, !.
compare(=, A, B) :- A == B.
```

Examples

```
| ?- compare(=, 2, 2).
% yes
| ?- compare(<, a, foo(1)).
% yes
| ?- compare(R, 9, 99).
```

```
R = < ?

% yes
```

Errors

None.

See also

'@<'/2, '@>'/2, '=='/2.

4.1.50 compound/1

Synopsis

```
compound(+T)
```

Description

Succeeds if T is a compound term. Note that floating-point numbers have a compound representation.

Examples

```
| ?- compound(compound(Compound)).
Compound = _524624 ?

% yes
| ?- compound(compound).
% no
| ?- compound(3.14).
% yes
```

Errors

None.

See also

None.

4.1.51 concatable_atom/1

Synopsis

```
concatable_atom(+T)
```

Description

Succeeds if `T` is a concatable atom. This definition of concatable atom is:

(i) all atoms are concatable atoms, and

(ii) all variables are concatable atoms.

Examples

```
| ?- concatable_atom(foo).
% yes
| ?- concatable_atom(Foo).
Foo = _524064 ?

% yes
| ?- concatable_atom(99).
% no
```

Errors

None.

See also

`atom/1`, `atom_concat/3`, `var/1`.

4.1.52 Conjunction — `','/2`

Synopsis

```
T1,T2
```

Description

This is the conjunction predicate. It succeeds if both `T1` and `T2` succeed.

Examples

```
| ?- true, true.
% yes
```

Errors

None.

See also

None.

4.1.53 consult/1

Synopsis

```
consult(+F)
```

Description

Opens the file specified by F and loads the terms found in the file into the clause store. Note that since consult uses open/4 , the argument F is in turn passed to absolute_file_name/2 for translation.

Note that this is a consult and not a reconsult. Should you consult the same file twice then you will have two copies of all read terms in the clause store. Any compiled procedures which are loaded with consult/1 replace any previous definitions. In this case consult/1 acts as reconsult/1 does.

Examples

```
| ?- consult('slask/foo').
% yes
```

Errors

instantiation_error The argument F was not instantiated.

See also

absolute_file_name/2, reconsult/1, '.'/2.

4.1.54 convert_char/2

Synopsis

```
convert_char(?C1, ?C2)
```

Description

Succeeds if C2 is the character conversion of C1 which is defined by the flag char_conversion and the predicate current_char_conversion/2. The predicate behaves as if it was defined as follows:

```
convert_char(Char, Converted) :-
        current_prolog_flag(char_conversion, on),
        current_char_conversion(Char, Converted),
        !.
convert_char(Char, Char).
```

Examples

```
| ?- char_conversion('z', 'y').
% yes
| ?- convert_char('z', C).
C = y ?

% yes
| ?- char_conversion('z', 'z').
% yes
| ?- convert_char('z', C).
C = z ?

% yes
```

Errors

See current_char_conversion/2.

See also

char_conversion, char_conversion/2, current_char_conversion/2.

4.1.55 copy_term/2

Synopsis

```
copy_term(+T1, ?T2)
```

Description

Succeeds if T2 is a copy of T1 where each uninstantiated variable in T1 is mapped to a newly allocated variable in T2. The predicate behaves as if it was defined as follows:

```
copy_term(T1, T2) :-
        asserta(dummy_copy_functor(T1)),
        retract(dummy_copy_functor(FreshT1)),
        FreshT1 = T2.
```

Examples

```
| ?- copy_term(foo(A,B), Copy).
A = _523696
B = _524352
Copy = foo(_536064,_536080) ?
```

```
% yes
| ?- copy_term(A,B).
A = _522720
B = _523376 ?

% yes
| ?- copy_term(A, 99).
A = _522720 ?

% yes
```

Errors

None.

See also

None.

4.1.56 current_char_conversion/2

Synopsis

```
current_char_conversion(?T1, ?T2)
```

Description

Succeeds if at some point `char_conversion(T1, T2)` where `T1 \== T2` has been used to update the character conversion table and this conversion is still valid, i.e., hasn't been deleted with `char_conversion(T1, T1)`.

Examples

```
| ?- bagof(T1-T2, current_char_conversion(T1, T2), Bag).
% no
| ?- char_conversion('p', 'q').
% yes
| ?- bagof(T1-T2, current_char_conversion(T1, T2), Bag).
T1 = _521856
T2 = _522848
Bag = [p-q] ?

% yes
| ?- char_conversion('p', 'p').
% yes
| ?- bagof(T1-T2, current_char_conversion(T1, T2), Bag).
```

```
% no
```

Errors

`type_error(character, T)` One of the arguments, `T1` or `T2`, was neither uninstantiated nor a character.

See also

`character/1`, `char_conversion/2`.

4.1.57 current_file_search_path/2

Synopsis

```
current_file_search_path(?A, ?D)
```

Description

Succeeds if at some point `add_file_search_path(A, D)` has been called to link the path alias `A` with the directory `D`.

Examples

```
| ?- current_file_search_path(A, D).
A = prolog
D = /home/bwat/BarrysProlog ? ;

A = misc
D = prolog(misc) ? ;

A = runtime
D = prolog(bin/runtime) ?

% yes
```

Errors

None.

See also

`add_file_search_path/2`, `del_file_search_path/2`.

4.1.58 current_generate_message/1

Synopsis

```
current_generate_message(?F)
```

Description

Succeeds if at some point add_generate_message(F) has been called to add the procedure specified by the predicate indicator F/2.

Examples

```
| ?- current_generate_message(F).
% no
| ?- add_generate_message(foo).
% yes
| ?- current_generate_message(F).
F = foo ?

% yes
| ?- del_generate_message(foo).
% yes
| ?- current_generate_message(F).
% no
```

Errors

None.

See also

add_generate_message/1, del_generate_message/1.

4.1.59 current_input/1

Synopsis

```
current_input(?T)
```

Description

Succeeds if T is the current input stream.

Examples

```
| ?- current_input(S).
S = $stream(0) ?

% yes
```

Errors

`domain_error(stream, T)` The argument `T` was neither uninstantiated nor a stream term.

See also

`set_input/1`.

4.1.60 current_message_hook/1

Synopsis

```
current_message_hook(?F)
```

Description

Succeeds if at some point `add_message_hook(F)` has been called to add the procedure specified by the predicate indicator `F/3`.

Examples

```
| ?- current_message_hook(F).
% no
| ?- add_message_hook(foo).
% yes
| ?- current_message_hook(F).
F = foo ?

% yes
| ?- del_message_hook(foo).
% yes
| ?- current_message_hook(F).
% no
```

Errors

None.

See also

add_message_hook/1, del_message_hook/1.

4.1.61 current_op/3

Synopsis

```
current_op(?P, ?S, ?O)
```

Description

Succeeds if the operator table contains an entry for the operator `O` with precedence `P` and specification S. The operator table is updated with op/3.

Examples

```
| ?- findall((P, S, '-'), current_op(P, S, '-'), L).
P = _522496
S = _523360
L = [(500,yfx,-),(200,fy,-)] ?

% yes
```

Errors

`domain_error(operator_priority, P)` The argument `P` was neither uninstantiated nor an integer between 0 and 1200 (inclusive).

`domain_error(operator_specifier, S)` The argument `S` was neither uninstantiated nor an operator specifier.

`type_error(atom, P)` The argument `O` was neither uninstantiated nor an atom.

See also

op/3.

4.1.62 current_output/1

Synopsis

```
current_output(?T)
```

Description

Succeeds if `T` is the current output stream.

Examples

```
| ?- current_output(S).
S = $stream(1) ?

% yes
```

Errors

`type_error(stream, T)` The argument `T` was neither uninstantiated nor a stream term.

See also

`set_output/1`.

4.1.63 current_portray/1

Synopsis

```
current_portray(?F)
```

Description

Succeeds if at some point `add_portray(F)` has been called to add the procedure specified by the predicate indicator `F/2`.

Examples

```
| ?- current_portray(F).
% no
| ?- add_portray(foo).
% yes
| ?- current_portray(F).
F = foo ?

% yes
| ?- del_portray(foo).
% yes
| ?- current_portray(F).
% no
```

Errors

None.

See also

add_portray/1, del_portray/1.

4.1.64 current_portray_message/1

Synopsis

```
current_portray_message(?F)
```

Description

Succeeds if at some point add_portray_message(F) has been called to add the procedure specified by the predicate indicator F/2.

Examples

```
| ?- current_portray_message(F).
% no
| ?- add_portray_message(foo).
% yes
| ?- current_portray_message(F).
F = foo ?

% yes
| ?- del_portray_message(foo).
% yes
| ?- current_portray_message(foo).
% no
```

Errors

None.

See also

add_portray_message/1, del_portray_message/1.

4.1.65 current_predicate/1

Synopsis

```
current_predicate(?T)
```

Description

Succeeds if T is a predicate indicator which identifies a dynamic predicate. This means that no compiled predicates can be found by T.

Examples

```
| ?- assert(foo).
% yes
| ?- current_predicate(foo/N).
N = 0 ?

% yes
```

Errors

`type_error(predicate_indicator, T)` The argument `T` was neither uninstantiated nor a predicate indicator.

See also

`predicate_indicator/1`.

4.1.66 current_prolog_flag/2

Synopsis

```
current_prolog_flag(?F, ?V)
```

Description

Succeeds if the Prolog flag `F` currently has the value `V`.

Examples

```
| ?- current_prolog_flag(integer_rounding_function, V).
V = toward_zero ?

% yes
```

Errors

`type_error(atom, F)` The argument `F` was neither uninstantiated nor an atom.

See also

`set_prolog_flag/2`.

4.1.67 current_query_class_hook/1

Synopsis

```
current_query_class_hook(?F)
```

Description

Succeeds if at some point add_query_class_hook(F) has been called to add the procedure specified by the predicate indicator F/5.

Examples

```
| ?- current_query_class_hook(F).
% no
| ?- add_query_class_hook(foo).
% yes
| ?- current_query_class_hook(F).
F = foo ?

% yes
| ?- del_query_class_hook(foo).
% yes
| ?- current_query_class_hook(F).
% no
```

Errors

None.

See also

add_query_class_hook/1, del_query_class_hook/1.

4.1.68 current_query_input_hook/1

Synopsis

```
current_query_input_hook(?F)
```

Description

Succeeds if at some point add_query_input_hook(F) has been called to add the procedure specified by the predicate indicator F/3.

Examples

```
| ?- current_query_input_hook(F).
% no
| ?- add_query_input_hook(foo).
% yes
| ?- current_query_input_hook(F).
F = foo ?

% yes
| ?- del_query_input_hook(foo).
% yes
| ?- current_query_input_hook(F).
% no
```

Errors

None.

See also

add_query_input_hook/1, del_query_input_hook/1.

4.1.69 current_query_map_hook/1

Synopsis

```
current_query_map_hook(?F)
```

Description

Succeeds if at some point add_query_map_hook(F) has been called to add the procedure specified by the predicate indicator F/4.

Examples

```
| ?- current_query_map_hook(F).
% no
| ?- add_query_map_hook(foo).
% yes
| ?- current_query_map_hook(F).
F = foo ?

% yes
| ?- del_query_map_hook(foo).
% yes
```

```
| ?- current_query_map_hook(F).
% no
```

Errors

None.

See also

add_query_map_hook/1, del_query_map_hook/1.

4.1.70 current_term_expansion/1

Synopsis

```
current_term_expansion(?A)
```

Description

Succeeds if the predicate identified by the predicate indicator `A/2` has been registered as a term expansion hook.

Examples

```
| ?- add_term_expansion(one).
% yes
| ?- add_term_expansion(two).
% yes
| ?- findall(X, current_term_expansion(X), L).
X = _522304
L = [one,two] ?

% yes
```

Errors

None

See also

add_term_expansion/1, del_term_expansion/1, term_expansion/2.

4.1.71 Cut — '!'/0

Synopsis

```
!
```

Description

Removes all choice-points created since the currently executing predicate was called. The predicates `call/1`, `once/1` and `'\+'/1` block the cut, that is to say the effect of any cut in a term given as an argument to these predicates is limited to any choice-points created by that term. The predicate `'->'/2` will limit the scope of any cuts in its first argument to just that argument, that is to say that these cuts are also blocked. Other predicates such as `'^'/2`, `','/2`, and, `';'/2` do not block cuts.

Examples

```
| ?- fail ; true.
% yes
| ?- !, fail ; true.
% no
| ?- !, fail -> fail ; true.
% yes
| ?- once((!, fail)) ; true.
% yes
```

Errors

None.

See also

None.

4.1.72 `del/3`

Synopsis

```
del(?Element, ?Before, ?After)
```

Description

Succeeds if deleting `Element` from the list `Before` is equal to the list `After`.

Examples

```
| ?- del(A, [1,2,3], L).
A = 1
L = [2,3] ? ;

A = 2
L = [1,3] ? ;
```

```
A = 3
L = [1,2] ? ;

% no
```

Errors

None.

See also

None.

4.1.73 del_file_search_path/2

Synopsis

`del_file_search_path(?A, ?D)`

Description

Deletes any link between the path alias `A` with the directory `D` created by `add_file_search_path(A, D)`. This predicate always succeeds.

Examples

```
| ?- add_file_search_path(a,b).
% yes
| ?- current_file_search_path(a,b).
% yes
| ?- del_file_search_path(a,b).
% yes
| ?- current_file_search_path(a,b).
% no
```

Errors

None.

See also

`add_file_search_path/2`, `current_file_search_path/2`.

4.1.74 del_generate_message/1

Synopsis

```
del_generate_message(+A)
```

Description

Deregisters a generate message hook registered with add_generate_message/1. This predicate always succeeds.

Examples

```
| ?- current_generate_message(F).
% no
| ?- add_generate_message(foo).
% yes
| ?- current_generate_message(F).
F = foo ?

% yes
| ?- del_generate_message(foo).
% yes
| ?- current_generate_message(F).
% no
```

Errors

None.

See also

add_generate_message/1, current_generate_message/1.

4.1.75 del_message_hook/1

Synopsis

```
del_message_hook(+A)
```

Description

Deregisters a message hook registered with add_message_hook/1. This predicate always succeeds.

Examples

```
| ?- current_message_hook(F).
% no
| ?- add_message_hook(foo).
% yes
| ?- current_message_hook(F).
F = foo ?

% yes
| ?- del_message_hook(foo).
% yes
| ?- current_message_hook(F).
% no
```

Errors

None.

See also

add_message_hook/1, current_message_hook/1.

4.1.76 del_portray/1

Synopsis

```
del_portray(+A)
```

Description

Deregisters a portray hook registered with add_portray/1. This predicate always succeeds.

Examples

```
| ?- current_portray(F).
% no
| ?- add_portray(foo).
% yes
| ?- current_portray(F).
F = foo ?

% yes
| ?- del_portray(foo).
% yes
```

```
| ?- current_portray(F).
% no
```

Errors

None.

See also

add_portray/1, current_portray/1.

4.1.77 del_portray_message/1

Synopsis

```
del_portray_message(+A)
```

Description

Deregisters a portray message hook registered with add_portray_message/1. This predicate always succeeds.

Examples

```
| ?- current_portray_message(F).
% no
| ?- add_portray_message(foo).
% yes
| ?- current_portray_message(F).
F = foo ?

% yes
| ?- del_portray_message(foo).
% yes
| ?- current_portray_message(foo).
% no
```

Errors

None.

See also

add_portray_message/1, current_portray_message/1.

4.1.78 del_query_class_hook/1

Synopsis

```
del_query_class_hook(+A)
```

Description

Deregisters a query class hook registered with add_query_class_hook/1. This predicate always succeeds.

Examples

```
| ?- current_query_class_hook(F).
% no
| ?- add_query_class_hook(foo).
% yes
| ?- current_query_class_hook(F).
F = foo ?

% yes
| ?- del_query_class_hook(foo).
% yes
| ?- current_query_class_hook(F).
% no
```

Errors

None.

See also

add_query_class_hook/1, current_query_class_hook/1.

4.1.79 del_query_input_hook/1

Synopsis

```
del_query_input_hook(+A)
```

Description

Deregisters a query input hook registered with add_query_input_hook/1. This predicate always succeeds.

Examples

```
| ?- current_query_input_hook(F).
% no
| ?- add_query_input_hook(foo).
% yes
| ?- current_query_input_hook(F).
F = foo ?

% yes
| ?- del_query_input_hook(foo).
% yes
| ?- current_query_input_hook(F).
% no
```

Errors

None.

See also

add_query_input_hook/1, current_query_input_hook/1.

4.1.80 del_query_map_hook/1

Synopsis

```
del_query_map_hook(+A)
```

Description

Deregisters a query map hook registered with add_query_map_hook/1. This predicate always succeeds.

Examples

```
| ?- current_query_map_hook(F).
% no
| ?- add_query_map_hook(foo).
% yes
| ?- current_query_map_hook(F).
F = foo ?

% yes
| ?- del_query_map_hook(foo).
% yes
```

```
| ?- current_query_map_hook(F).
% no
```

Errors

None.

See also

add_query_map_hook/1, current_query_map_hook/1.

4.1.81 del_term_expansion/1

Synopsis

```
del_term_expansion(+A)
```

Description

Deregisters the predicate identified by the predicate indicator `A/2` as a term expansion hook. This predicate always succeeds.

Examples

```
| ?- add_term_expansion(foo).
% yes
| ?- current_term_expansion(foo).
% yes
| ?- del_term_expansion(foo).
% yes
| ?- current_term_expansion(foo).
% no
```

Errors

None

See also

add_term_expansion/1, current_term_expansion/1, term_expansion/2.

4.1.82 delete_all/3

Synopsis

```
delete_all(?Element, ?Before, ?After)
```

Description

Deletes all occurrences of `Element` from the list `Before` giving the list `After`. The members of `Before` are compared with `Element` using `'='/2`. This predicate is deterministic.

Examples

```
| ?- delete_all(1, [1,2,1,3,1,4], Lst).
Lst = [2,3,4] ?

% yes
```

Errors

None.

See also

`delete_all_equal_terms/3`.

4.1.83 delete_all_equal_terms/3

Synopsis

```
delete_all_equal_terms(?Element, ?Before, ?After)
```

Description

Deletes all occurrences of `Element` from the list `Before` giving the list `After`. The members of `Before` are compared with `Element` using `'=='/2`. This predicate is deterministic.

Examples

```
| ?- delete_all_equal_terms(1, [1,2,1], Lst).
Lst = [2] ?

% yes
| ?- delete_all_equal_terms(_, [1,2,1], Lst).
Lst = [1,2,1] ?

% yes
```

Errors

None.

See also

delete_all/3.

4.1.84 delete_deterministically/3

Synopsis

```
delete_deterministically(?Element, ?Before, ?After)
```

Description

Deletes the first occurrence of `Element` from the list `Before` giving the list `After`. The members of `Before` are compared with `Element` using `'=='/2`. This predicate is a deterministic version of `del/3`.

Examples

```
| ?- delete_deterministically(1, [1,2,1], Lst).
Lst = [2,1] ?

% yes
| ?- delete_deterministically(X, [1,2,1], Lst).
X = 1
Lst = [2,1] ?

% yes
```

Errors

None.

See also

del/3.

4.1.85 Disjunction — ';'/2

Synopsis

```
T1;T2
```

Description

This is the disjunction predicate. It succeeds if at least one of either `T1` or `T2` succeed.

Examples

```
| ?- fail ; true.
% yes
```

Errors

None.

See also

None.

4.1.86 display/1

Synopsis

```
display(+T)
```

Description

Behaves as if it were defined as follows:

```
display(Term) :-
        current_output(S),
        display(S, Term).
```

Examples

See display/2.

Errors

See display/2.

See also

display/2.

4.1.87 display/2

Synopsis

```
display(+S, +T)
```

Description

Behaves as if it were defined as follows:

```
display(Stream, Term) :-
        write_term(Stream, Term, [ignore_ops(true)]).
```

Examples

See write_term/2.

Errors

See write_term/2.

See also

write_term/2.

4.1.88 Dot — '.'/2

Synopsis

```
[File|Files]
```

Description

This is the notation used to (re)consult a list of files. Elements of the list are passed as arguments to consult/1 unless they are prefixed with '-'/1 in which case they are passed as arguments to reconsult/1 instead.

Examples

```
| ?- ['slask/foo', -'slask/foo'].
% yes
```

Errors

None.

See also

consult/1, reconsult/1.

4.1.89 ensure_loaded/1

Synopsis

```
ensure_loaded(+T)
```

Description

If the file `T` has previously been consulted then do nothing. Otherwise, call `consult(T)`.

Examples

```
| ?- ensure_loaded('slask/foo').
% yes
```

Errors

`instantiation_error` The argument `T` was not instantiated.

See also

`consult/1`, `reconsult/1`.

4.1.90 `eval/2`

Synopsis

```
eval(+E, ?N)
```

Description

Evaluates the expression `E` to give the number `N`. A valid expression is either a number, or a compound where all arguments are valid expressions and the functor is one of a predefined set of arithmetic functors.

Those arithmetic functors which represent elementary functions are implemented by internal predicates found in the file `elementary_functions.fasl` and this will need to be loaded before these can be used. The internal predicates implementing the special function's arithmetic functors are found in the file `special_functions.fasl` and again this will have to be loaded before use.

A number evaluates to itself. A list with a single element which is a number evaluates to that number. A compound with an arithmetic functor will first have all of its arguments evaluated before the functor is interpreted and the result calculated.

The precision of a number is the amount of bits used in its representation. The integers are of arbitrary precision which means that if you multiply two integers, then the result may be an integer of greater precision. The largest number which can be represented depends upon how much memory can be allocated by the system. Floating-point numbers are represented by a compound of three arguments (parts): `'$float'(F, E, P)`

`F` The fraction part. This is an integer of fixed precision.

`E` The exponent part. This is an integer of arbitrary precision.

`P` The floating-point precision part. This determines the number of bits used in the part `E` above. This value is a system wide parameter that can be changed via the flag `floating_point_precision`.

Now, if `F` is an integer of `P` bits, then we can imagine that it has the binary point at its far right hand side. Should we shift the binary point left `(P-E)` bits then we would have the floating-point number represented by `'$float'(F, E, P)`. Should `(P-E)` be less than zero then we would shift the point right by `abs(P-E)` bits. Here is an example:

```
| ?- current_prolog_flag(floating_point_precision, P).
P = 64 ?

% yes
| ?- write_canonical(3.14), nl.
'$float'(1448069409786199 8019,2,64)
% yes
| ?- X is 1448069409786199 8019 >> (64-2).
X = 3 ?

% yes
| ?-  X is 1448069409786199 8019 * 100 >> (64-2).
X = 314 ?

% yes
```

Since part `E` is an arbitrary precision integer, the magnitude of a floating-point number's exponent can be arbitrarily large.

The arithmetic functors which represent the bit-wise operations — `'<<'(E1, E2)`, `'\/'(E1, E2)`, `'>>'(E1, E2)`, `'/\'(E1, E2)`, and `'\'(E)` — only work on integers greater than or equal to zero. There is also the issue of word size. With these operations, the smallest multiple of the target machine word size which is large enough to hold all of the given arguments is the result word size. So, if you were to calculate the logical not (negation/inversion) of zero, then on a 64-bit machine the result would be a positive integer of precision 64-bits with all bits set to one, whereas on a 32-bit machine the result would be a positive integer of precision 32-bits with all bits set to one.

What follows is a list of arithmetic functors and their evaluation:

`abs(E)` Calculates the absolute value of the argument.

`'+'(E1, E2)` Calculates the addition of the arguments.

`atan(E)` Calculates the arc tangent of the argument in radians. This is an elementary function found in `elementary_functions.fasl`.

`'/\'(E1, E2)` Calculates the bit-wise and of the two arguments. Both arguments must be integers greater than or equal to zero.

`'\'(E)` Calculates the bit-wise not (negation/inversion) of the argument. The argument must be an integer greater than or equal to zero.

`'<<'(E1, E2)` Calculates the bit-wise left shift of the arguments: `E1` is shifted left `E2` bits. Both arguments must be integers greater than or equal to zero.

`'\/'(E1, E2)` Calculates the bit-wise or of the arguments. Both arguments must be integers greater than or equal to zero.

`'>>'(E1, E2)` Calculates the bit-wise right shift of the arguments: `E1` is shifted right `E2` bits. Both arguments must be integers greater than or equal to zero.

`ceiling(E)` Calculates the smallest integer that is not less than the argument.

`cos(E)` Calculates the cosine of the argument in radians. This is an elementary function found in `elementary_functions.fasl`.

`exp(E)` Calculates the value of natural antilogarithm of the argument. This is an elementary function found in `elementary_functions.fasl`.

`'**'(E1, E2)` Calculates the exponentiation of the arguments: `E1` is raised to the power of `E2`. This is an elementary function found in the file `elementary_functions.fasl`. If `E1` is less than zero, then `E2` must be an integer. If `E1` is zero, then `E2` must be greater than or equal to zero.

`factorial(E)` Calculates the factorial of the argument. The argument must be an integer greater than or equal to zero. This is a special function found in `special_functions.fasl`.

`float(E)` Converts the argument into a floating-point number.

`float_fractional_part(E)` Calculates the fractional part of the floating-point argument.

`float_integer_part(E)` Calculates the integer part of the floating-point argument. The result is a floating-point number.

`'/'(E1, E2)` Calculates the floating-point division of `E1` by `E2`. `E2` must not evaluate to zero.

`'//'(E1, E2)` Calculates the integer division of `E1` by `E2`. Both arguments must be integers. `E2` must not evaluate to zero.

`div(E1, E2)` Calculates the floored division of `E1` by `E2`. Both arguments must be integers. `E2` must not evaluate to zero.

`floor(E)` Calculates the largest integer that is not greater than the argument.

`gamma(E)` Calculates the gamma generalised factorial of the argument. This is a special function found in `special_functions.fasl`.

`ln(E)` Calculates the natural logarithm of the argument. This is an elementary function found in `elementary_functions.fasl`.

`log(E)` Calculates the base 10 logarithm of the argument. This is an elementary function found in `elementary_functions.fasl`.

`mod(E1, E2)` Calculates `E1` modulo `E2`. Both arguments must be integers. `E2` must not evaluate to zero.

`'*'(E1, E2)` Calculates the multiplication of the two arguments.

`rem(E1, E2)` Calculates `E1` remainder `E2`. Both arguments must be integers. `E2` must not evaluate to zero.

`round(E)` Calculates the nearest integer to the argument.

`sign(E)` The result is the sign of the argument.

`'-'(E)` Calculates the unary minus of the argument, i.e., the sign of the argument is reversed.

`sin(E)` Calculates the sine of the argument in radians. This is an elementary function found in `elementary_functions.fasl`.

`sqrt(E)` Calculates the square root of the argument. This is an elementary function found in `elementary_functions.fasl`.

`'-'(E1, E2)` Calculates the subtraction of `E2` from `E1`.

`truncate(E)` Calculates the integer which is equal to `float_integer_part(E)`.

Examples

```
| ?- ensure_loaded(runtime(elementary_functions)).
% yes
| ?- ensure_loaded(runtime(special_functions)).
% yes
| ?- X is 99.9.
X = 99.9 ?
```

```
% yes
| ?- X is "A".
X = 65 ?

% yes
| ?- X is [99].
X = 99 ?

% yes
| ?- X is abs(-2.3).
X = 2.3 ?

% yes
| ?- X is 1.2+5.
X = 6.2 ?

% yes
| ?- X is atan(1).
X = 0.785398 ?

% yes
| ?- X is 129 /\ 7.
X = 1 ?

% yes
| ?- X is \7.
X = 18446744073709551608 ?

% yes
| ?- X is 2 << 3.
X = 16 ?

% yes
| ?- X is 129 \/ 7.
X = 135 ?

% yes
| ?- X is 16 >> 3.
X = 2 ?

% yes
| ?- X is ceiling(1.5).
X = 2 ?
```

```
% yes
| ?- X is cos(1).
X = 0.540302 ?

% yes
| ?- X is exp(1).
X = 2.718282 ?

% yes
| ?- X is 2**3.
X = 8 ?

% yes
| ?- X is factorial(3).
X = 6 ?

% yes
| ?- X is float(3).
X = 3.0 ?

% yes
| ?- X is float_fractional_part(3.14).
X = 0.14 ?

% yes
| ?- X is float_integer_part(3.14).
X = 3.0 ?

% yes
| ?- X is 2/3.
X = 0.666667 ?

% yes
| ?- X is 2//3.
X = 0 ?

% yes
| ?- X is -2 // 3.
X = 0 ?

% yes
| ?- X is 2 div 3.
X = 0 ?
```

```
% yes
| ?- X is -2 div 3.
X = -1 ?

% yes
| ?- X is floor(1.5).
X = 1 ?

% yes
| ?- X is gamma(-1.5).
X = 2.363271 ?

% yes
| ?- X is ln(exp(1)).
X = 1.0 ?

% yes
| ?- X is log(10).
X = 1.0 ?

% yes
| ?- X is 5 mod 3.
X = 2 ?

% yes
| ?- X is -5 mod 3.
X = 1 ?

% yes
| ?- X is 5 * 3.
X = 15 ?

% yes
| ?- X is 5 rem 3.
X = 2 ?

% yes
| ?- X is -5 rem 3.
X = -2 ?

% yes
| ?- X is round(5.3).
X = 5 ?
```

```
% yes
| ?- X is round(-5.3).
X = -5 ?

% yes
| ?- X is -5, Y is -X.
X = -5
Y = 5 ?

% yes
| ?- X is sin(1).
X = 0.841471 ?

% yes
| ?- X is sqrt(2).
X = 1.414214 ?

% yes
| ?- X is 2-3.
X = -1 ?

% yes
| ?- X is truncate(3.14).
X = 3 ?

% yes
```

Errors

`domain_error(not_less_than_zero)` An argument that should have been greater than or equal to zero was not.

`evaluation_error(zero_divisor)` An attempt was made to divide by zero.

`instantiation_error` A given expression was an uninstantiated variable.

`type_error(evaluable, E)` A given expression was not evaluable.

`type_error(evaluable, F/N)` A given compound did not have an arithmetic functor.

`type_error(integer, N)` An argument which should have been an integer was not.

See also

`floating_point_precision`.

4.1.91 Existential quantification — '^'/2

Synopsis

```
?V ^ +T
```

Description

Succeeds if there exists a `V` such that `T` is true. This is equivalent to `call(T)`.

Examples

```
| ?- X^print(hello), nl.
hello
X = _520432 ?

% yes
```

Errors

None.

See also

`call/1`.

4.1.92 expand_term/2

Synopsis

```
expand_term(+T, -E)
```

Description

Expands the the term `T`, using term expansion hooks and Definite Clause Grammar (DCG) translations, giving the result `E`. First `term_expansion/2` is called and the result of this is passed on to the internal DCG expander.

Examples

```
| ?- expand_term((a-->b), E).
E = a(_541648,_541616) :- b(_541648,_541616) ?

% yes
```

Errors

None.

See also

Section 4.2, term_expansion/2.

4.1.93 fail/0

Synopsis

```
fail
```

Description

The predicate which never succeeds.

Examples

```
| ?- fail.
% no
```

Errors

None.

See also

true/0.

4.1.94 file_search_path/2

Synopsis

```
file_search_path(+A, -D)
```

Description

Translates a path alias A into a directory D. The connection between an alias and a term is made with add_file_search_path/2. If this term is an atom then this is also the directory result of file_search_path/2. If, however, this term is a compound with a single argument, then we treat the functor as another alias to be resolved, and the argument as the directory name to be appended onto the result of resolving the new alias. It is the result of the append that becomes the result of add_file_search_path/2.

Examples

```
| ?- add_file_search_path(logging_dir, '/usr/local/logs').
% yes
| ?- file_search_path(logging_dir, D).
D = /usr/local/logs ?

% yes
| ?- add_file_search_path(backup_logging_dir, logging_dir(backups)).
% yes
| ?- file_search_path(backup_logging_dir, D).
D = /usr/local/logs/backups ?

% yes
```

Errors

None.

See also

add_file_search_path/2.

4.1.95 findall/3

Synopsis

```
findall(+Template, +Goal, ?Bag)
```

Description

Similar to bagof/3 but with the following exceptions:

- Should `Goal` be unsatisfiable, `findall/3` does not fail as bagof/3 does. Instead `Bag` is unified with the empty list — `Bag=[]`.
- Any uninstantiated variables free in `Goal` and not in `Template` are automatically existentially quantified. This means `findall/3` does not enumerate solutions for these variables as bagof/3 does.

Examples

```
| ?- bagof(X, member(X-Y, [1-2,3-4,5-6]), L).
X = _521856
Y = 2
L = [1] ? ;
```

```
X = _521856
Y = 4
L = [3] ? ;

X = _521856
Y = 6
L = [5] ? ;

% no
| ?- findall(X, member(X-Y, [1-2,3-4,5-6]), L).
X = _522304
Y = _525584
L = [1,3,5] ?

% yes
| ?- bagof(nothing, fail, Bag).
% no
| ?- findall(nothing, fail, Bag).
Bag = [] ?

% yes
```

Errors

`instantiation_error` The argument `Goal` was not instantiated.

`type_error(callable, Goal)` The argument `Goal` was not callable.

`type_error(list, Bag)` The argument `Bag` was neither uninstantiated nor a list.

See also

bagof/3.

4.1.96 `float/1`

Synopsis

`float(+T)`

Description

Succeeds if `T` is a floating-point number.

Examples

```
| ?- float(9).
% no
| ?- float(9.0).
% yes
```

Errors

None.

See also

None.

4.1.97 flush_output/0

Synopsis

```
flush_output
```

Description

Flushes the current output stream. This predicate behaves as if it was defined as follows:

```
flush_output :-
        current_output(S),
        flush_output(S).
```

Examples

```
| ?- flush_output.
% yes
```

Errors

None.

See also

flush_output/1.

4.1.98 flush_output/1

Synopsis

```
flush_output(+S)
```

Description

Flushes the stream identified by `S`.

Examples

```
| ?- open('test.txt', write, O), print(O, foo),
     flush_output(O), close(O).
O = $stream(3) ?

% yes
```

Errors

`instantiation_error` The argument `S` was not instantiated.

`domain_error(stream_or_alias, S)` The argument `S` was neither a stream term nor an alias atom.

`existence_error(stream, S)` The argument `S` did not identify an existing stream.

`permission_error(output, stream, S)` The argument `S` was not an output stream.

See also

`flush_output/1`.

4.1.99 `format/2`

Synopsis

`format(+C,+A)`

Description

Equivalent to `current_output(S), format(S, C, A)`.

Examples

See `format/3`.

Errors

See `format/3`.

See also

current_output/1, format/3.

4.1.100 format/3

Synopsis

```
format(+S,+C,+A)
```

Description

Formats the control string `C` with the arguments `A` and writes the result onto the stream `S`. `C` is either a list of characters or an atom. `A` is a list.

Each character of the control string is written on the stream unless it is the character `~` which is the escape character. The next character after the escape is the control character and determines the action to be taken. Some actions take an argument given as an element of a list passed via the argument `A`. The order of the elements of this list matches the order of the argument taking control actions in `C`, i.e, the fourth argument taking control action of `C` uses the fourth element of `A`. There is another type of argument, numerical arguments, which are given between the escape and control characters, or, in the arguments list which is signified by placing the character `*` between the escape and control characters. This gives three different methods of argument passing:

`format(S, '~a', [argument])` Control character `a` takes an argument in the arguments list.

`format(S, '~23a', [argument])` Control character `a` takes an argument in the arguments list and a numerical argument 23 in the control string.

`format(S, '~*a', [23,argument])` Control character `a` takes an argument and a numerical argument in the arguments list.

Here is the list of valid control characters (numerical arguments are taken by those characters which are prefixed by `<N>`):

`~a` Interpret the argument as an atom and print it without quoting.

`~<N>a` Interpret the argument as an atom and print its first `N` characters without quoting.

`~c` Interpret the argument as a character code and print it.

`~<N>c` Interpret the argument as a character code and print it `N` times.

`~d` Interpret the argument as a number and print it in decimal.

`~f` Interpret the argument as a floating-point number and print it.

`~<N>f` Interpret the argument as a floating-point number and print it with a precision of `N` digits after the radix point.

`~i` Ignore the argument.

`~k` Pass the argument to `write_canonical/2`.

`~n` Print a newline.

`~<N>n` Print a newline `N` times.

`~N` Print a newline only if the output stream is not at column position zero.

`~p` Pass the argument to `print/2`.

`~q` Pass the argument to `writeq/2`.

`~r` Interpret the argument as an integer and print it in radix 8.

`~<N>r` Interpret the argument as an integer and print it radix `N`. The radix must be between 2 and 36 (inclusive).

`~s` Interpret the argument as a string and print it without quoting.

`~<N>s` Interpret the argument as a string and print its first `N` characters.

`~t` Reserved. Do not use.

`~w` Pass the argument to `write/2`.

`~~` Print the `~` character.

`~@` Pass the argument to `call/1`.

Examples

```
| ?- format("~a~2a~n", [foo, foo]).
foofo
% yes
| ?- format("~*a~n", [3, three]).
thr
% yes
| ?- format("~c~2c~n", [64, 65]).
@AA
% yes
| ?- format("~d~n", [123]).
123
% yes
```

```
| ?- X is 1/3, format("~f ~10f~n", [X, X]).
0.333333 0.3333333333
X = 0.333333 ?

% yes
| ?- format("~i~k~n", [dummy, 1+2]).
+(1,2)
% yes
| ?- format("line~2n~Nline~n", []).
line

line
% yes
| ?- format("~p ~q~n", [' a ', ' a ']).
 a  ' a '
% yes
| ?- format("~r ~16r~n", [24, 24]).
30 18
% yes
| ?- format("~s ~3s~n", ["Hello", "World"]).
Hello Wor
% yes
| ?- format("~w~n", ['$VAR'(16)]).
Q
% yes
| ?- format("~@~n", [print('Hello World')]).
Hello World
% yes
```

Errors

`existence_error(stream, S)` The argument `S` was not an open stream.

`domain_error(empty_arguments, [])` The argument `A` was not of the correct length.

`domain_error(not_optional_argument, C)` An optional argument was given to the format control character `C`.

`domain_error(not_less_than_zero, N)` The numerical argument `N` was less than zero.

`domain_error(number_base, N)` The numerical argument `N` given to `~r` was not a valid base.

`domain_error(stream_or_alias, S)` The argument `S` was not a stream term or an atom stream alias.

`instantiation_error` The arguments `S`, `A`, or `C` were not instantiated.

`type_error(atom, T)` The argument to `~a` was not an atom.

`type_error(atom_or_list, C)` The argument `C` was neither an atom nor a list.

`type_error(character_code, T)` The argument to `~c` was not an character code.

`type_error(format_control, C)` The character `C` is not a valid control character.

`type_error(integer, T)` The given numerical argument was not an integer.

`type_error(list, A)` Either the argument `A` was not a list, or, the argument to `~s` was not a list.

`type_error(number, T)` The argument to `~f` was not a number.

See also

None.

4.1.101 `functor/3`

Synopsis

```
functor(+T,?F,?A)
functor(?T,+F,+A)
```

Description

Succeeds if the term `T` has the principle functor `F` with arity `A`.

Examples

```
| ?- functor(f(a1,a2), F, A).
F = f
A = 2 ?

% yes
| ?- functor(T, f, 2).
T = f(_528960,_528976) ?
```

```
% yes
| ?- functor(three, F, A).
F = three
A = 0 ?

% yes
| ?- functor(3, F, A).
F = 3
A = 0 ?

% yes
| ?- functor(3.0, F, A).
F = $float
A = 3 ?

% yes
```

Errors

`instantiation_error` Either both `T` and `F` were not instantiated, or both `T` and `A` were not instantiated.

`type_error(atomic, F)` The argument `T` was uninstantiated and `A` was instantiated, and `F` was not atomic.

`type_error(atom, F)` The argument `T` was uninstantiated and `A` was instantiated, and `F` was not an atom.

`type_error(integer, A)` The argument `T` was uninstantiated and `A` was instantiated, and `A` was not an integer.

`domain_error(not_less_than_zero, A)` The argument `T` was uninstantiated and `A` was instantiated, and `A` was less than zero.

`representation_error(max_arity)` The argument `T` was uninstantiated and `A` was instantiated, and `A` was not less than or equal to the value of the flag `max_arity`.

See also

None.

4.1.102 `generate_message_line/3`

Synopsis

```
generate_message_line(?A, ?B, ?C).
```

Description

This predicate is used internally by generate_message_lines/3. Behaves as if it were defined as follows:

```
generate_message_line([]) --> [].
generate_message_line([Control-Arguments|T]) -->
        [Control-Arguments],
        generate_message_line(T).
generate_message_line([write_term(Term, Options)|T]) -->
        [write_term(Term, Options)],
        generate_message_line(T).
```

Examples

None.

Errors

None.

See also

generate_message_lines/3.

4.1.103 generate_message_lines/3

Synopsis

```
generate_message_lines(?A, ?B, ?C).
```

Description

This predicate is used internally by print_message/2. Behaves as if it were defined as follows:

```
generate_message_lines([]) --> [].
generate_message_lines([H|T]) -->
        generate_message_line(H),
        [nl],
        generate_message_lines(T).
```

Examples

None.

Errors

None.

See also

generate_message_line/3, print_message/2.

4.1.104 get_byte/1

Synopsis

```
get_byte(?B)
```

Description

Behaves as if it were defined as follows:

```
get_byte(Byte) :-
        current_input(S),
        get_byte(S, Byte).
```

Examples

See get_byte/2.

Errors

See get_byte/2.

See also

current_input/1, get_byte/2.

4.1.105 get_byte/2

Synopsis

```
get_byte(+S, ?B)
```

Description

Reads a byte `B` from the binary stream `S`.

Examples

```
test_get_byte :-
        open('test_file.bin', read, Stream, [type(binary)]),
        get_byte(Stream, 16'de),
        get_byte(Stream, 16'ad),
        get_byte(Stream, 16'be),
        get_byte(Stream, 16'ef),
        skip_to_end_of_binary_file(Stream),
        close(Stream).
```

Errors

`instantiation_error` The argument `S` was not instantiated.

`type_error(in_byte, B)` The argument `B` was instantiated but not a valid input byte.

`permission_error(input, past_end_of_stream, S)` Tried to read past the end of the stream.

`permission_error(input, stream, S)` There is a lack of permission to read from stream `S`.

`permission_error(input, text_stream, S)` The argument `S` refers to a text stream.

`existence_error(stream, S)` The stream `S` does not exist.

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

See also

in_byte/1, unget_byte/2.

4.1.106 get_char/1

Synopsis

```
get_char(?C)
```

Description

Behaves as if it were defined as follows:

```
get_char(Char) :-
        current_input(S),
        get_char(S, Char).
```

Examples

See get_char/2.

Errors

See get_char/2.

See also

current_input/1, get_char/2.

4.1.107 get_char/2

Synopsis

```
get_char(+S, ?C)
```

Description

Reads a character C from the text stream S.

Examples

```
test_get_char :-
        open('test_file.txt', read, Stream),
        get_char(Stream, d),
        get_char(Stream, e),
        get_char(Stream, a),
        get_char(Stream, d),
        skip_to_end_of_text_file(Stream),
        close(Stream).
```

Errors

`instantiation_error` The argument S was not instantiated.

`type_error(in_character, C)` The argument C was instantiated but not a valid input character.

`permission_error(input, past_end_of_stream, S)` Tried to read past the end of the stream.

`permission_error(input, stream, S)` There is a lack of permission to read from stream S.

`permission_error(input, binary_stream, S)` The argument S refers to a binary stream.

`existence_error(stream, S)` The stream `S` does not exist.

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

See also

`unget_char/2`.

4.1.108 get_code/1

Synopsis

```
get_code(?C)
```

Description

Behaves as if it were defined as follows:

```
get_code(Code) :-
        current_input(S),
        get_code(S, Code).
```

Examples

See `get_code/2`.

Errors

See `get_code/2`.

See also

`current_input/1`, `get_code/2`.

4.1.109 get_code/2

Synopsis

```
get_code(+S, ?C)
```

Description

Reads a character code `C` from the text stream `S`.

Examples

```
test_get_code :-
        open('test_file.txt', read, Stream),
        get_code(Stream, 64),
        get_code(Stream, 65),
        get_code(Stream, 66),
        get_code(Stream, 67),
        skip_to_end_of_text_file(Stream),
        close(Stream).
```

Errors

`instantiation_error` The argument `S` was not instantiated.

`representation_error(in_character_code, C)` The argument `C` was instantiated but not a valid input character code.

`permission_error(input, past_end_of_stream, S)` Tried to read past the end of the stream.

`permission_error(input, stream, S)` There is a lack of permission to read from stream `S`.

`permission_error(input, binary_stream, S)` The argument `S` refers to a binary stream.

`existence_error(stream, S)` The stream `S` does not exist.

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

See also

`unget_code/2`.

4.1.110 `ground/1`

Synopsis

`ground(+T)`

Description

Succeeds if `T` is a ground term.

Examples

```
| ?- ground(X).
% no
| ?- ground(x).
% yes
| ?- ground(foo(x)).
% yes
```

Errors

None.

See also

None.

4.1.111 `halt/0`

Synopsis

```
halt
```

Description

Halts execution. The operating system is given a exit status code equivalent to the call `exit(EXIT_SUCCESS)` in the C programming language.

Examples

```
| ?- halt.
```

Errors

None.

See also

`halt/1`.

4.1.112 `halt/1`

Synopsis

```
halt(+N)
```

Description

Halts execution with exit code `N`. The predicate fails if `N` is not an integer that fits into a target machine word. The operating system is given a exit status code equivalent to the call `exit(N)` in the C programming language.

Examples

```
| ?- halt(99).
```

Errors

`instantiation_error` `N` must be ground.

`type_error(integer, N)` `N` must be of type integer.

See also

`halt/0`.

4.1.113 If — '->'/2

Synopsis

```
T1 -> T2
T1 -> T2 ; T3
```

Description

The case `T1 -> T2` is equivalent to `T1 -> T2 ; fail`. The case `T1 -> T2 ; T3` is equivalent to `(call(T1), !, T2) ; T3` where the cut is limited in scope to this goal, i.e, should `T1` succeed and `T2` fail, there will be no backtrack to `T3`.

Examples

```
| ?- true -> print('YES'), nl.
YES
% yes
| ?- fail -> print('NO'), nl.
% no
| ?- true -> print('YES'), nl, fail ; print('NO'), nl.
YES
% no
```

Errors

None.

See also

'!'/0.

4.1.114 in_byte/1

Synopsis

```
in_byte(+B)
```

Description

Succeeds if `B` is a valid input byte. Behaves as if it were defined as follows:

```
in_byte(Byte) :-
        integer(Byte),
        Byte >= -1,
        Byte =< 255.
```

Examples

```
| ?- in_byte(99).
% yes
```

Errors

None.

See also

None.

4.1.115 in_character/1

Synopsis

```
in_character(+C)
```

Description

Succeeds if `C` is a valid input character, that is to say, `C` is either the atom `end_of_file` or a valid character.

Examples

```
| ?- in_character(end_of_file).
% yes
| ?- in_character(f).
% yes
| ?- in_character(ff).
% no
```

Errors

`instantiation_error` The argument `C` was not instantiated.

See also

`character/1`.

4.1.116 `in_character_code/1`

Synopsis

```
in_character_code(+C)
```

Description

Succeeds if `C` is a valid input character code, that is to say, `C` is either -1 or a valid character code.

Examples

```
| ?- in_character_code(-1).
% yes
| ?- in_character_code(99).
% yes
| ?- in_character_code(-99).
% no
```

Errors

None.

See also

`character_code/1`.

4.1.117 infix_op_specifier/1

Synopsis

```
infix_op_specifier(?T)
```

Description

Succeeds if `T` is an infix operator specifier.

Examples

```
| ?- findall(O, infix_op_specifier(O), L).
O = _522304
L = [xfx,xfy,yfx] ?

% yes
```

Errors

None.

See also

current_op/3, op/3, op_specifier/1, postfix_op_specifier/1, prefix_op_specifier/1.

4.1.118 integer/1

Synopsis

```
integer(+T)
```

Description

Succeeds if `T` is an integer.

Examples

```
| ?- integer(9).
% yes
| ?- integer(9.0).
% no
```

Errors

None.

See also

None.

4.1.119 io_mode/1

Synopsis

```
io_mode(?T)
```

Description

Succeeds if `T` is a valid I/O mode used by `open/4`.

Examples

```
| ?- findall(M, io_mode(M), Modes).
M = _522304
Modes = [read,write,append] ?

% yes
```

Errors

None.

See also

`open/4`.

4.1.120 is/2

Synopsis

```
T is E
```

Description

Succeeds if `T` is unified with to the arithmetic evaluation of `E` which was calculated with `eval/2`.

Examples

```
| ?- X is 1+2.
X = 3 ?

% yes
| ?- X is 2**3.
```

```
X = 8 ?

% yes
```

Errors

See eval/2.

See also

eval/2.

4.1.121 key_pair/1

Synopsis

```
key_pair(+T)
```

Description

Succeeds if T is a key pair term. The collection of key pairs consists of all compound terms with principle functor '-'/2.

Examples

```
| ?- key_pair(a-1).
% yes
| ?- key_pair(99).
% no
| ?- key_pair(X).
% no
| ?-
```

Errors

None.

See also

keysort/2.

4.1.122 keysort/2

Synopsis

```
keysort(+T1, ?T2)
```

Description

Succeeds if `T1` is a list of key pair terms and `T2` has all of the members of `T1` in an ascending order — the ordering relation on key pairs being the predicate `'@<'/2` applied to the first elements of the pairs — and `T1` and `T2` have equal lengths, i.e, no merging takes place.

Examples

```
| ?- keysort([2-a, 1-b, 3-c, 1-b], Result).
Result = [1-b,1-b,2-a,3-c] ?

% yes
```

Errors

`instantiation_error` The argument `T1` was not instantiated.

`type_error(list, T1)` The argument `T1` was not a list.

`type_error(key_pair, T1)` The argument `T1` was not a list of key pairs.

See also

`key_pair/1`.

4.1.123 `length/2`

Synopsis

`length(?L, ?N)`

Description

Succeeds if the length of the list `L` is `N`.

Examples

```
| ?- length(L, 3).
L = [_552528,_552576,_552624] ?

% yes
| ?- length([1,2], N).
N = 2 ?

% yes
```

Errors

None.

See also

None.

4.1.124 `listing/0`

Synopsis

```
listing
```

Description

Displays the clauses connected with all of the dynamic predicates in the clause store. This predicate behaves as if it were defined as follows:

```
listing :-
        current_predicate(P),
        listing(P),
        fail.
listing.
```

Examples

None.

Errors

None.

See also

`current_predicate/1`, `listing/1`.

4.1.125 `listing/1`

Synopsis

```
listing(+Pred)
```

Description

This predicate behaves as if it were defined as follows:

```
listing(P) :-
        current_output(S),
        listing(S, P).
```

Examples

See listing/2.

Errors

See listing/2.

See also

current_output/1, listing/2.

4.1.126 listing/2

Synopsis

```
listing(+Stream, +Pred)
```

Description

Displays the clauses connected with the predicate(s) `Pred` on `Stream`. The argument `Pred` can be:

- A variable which is ignored.
- A predicate indicator `F/N`. If `F/N` specifies an existing dynamic predicate then the clauses connected with this predicated are passed on to portray_clause/2. If no such dynamic predicate exists then nothing is done and the call succeeds.
- An atom `A`, `A \== []`, which is equivalent to a call of `listing(Stream, A/0)`.
- A list which has its elements processed one at a time.

Examples

```
| ?- assert((a:-b;c,!)).
% yes
| ?- assert((d(X))).
X = _548128 ?

% yes
| ?- listing([a,d/1]).
a :-
    (   b
    ;   c,
        !
    ).
d(A).
% yes
```

Errors

See portray_clause/2.

See also

portray_clause/2.

4.1.127 max/3

Synopsis

```
max(+A, +B, ?C)
```

Description

Succeeds if `C` is the maximum of `A` and `B`. This predicate behaves as if it were defined as:

```
max(A,B,C) :- A >= B, !, C = A.
max(_,B,B).
```

Examples

```
| ?- max(1,2,Max).
Max = 2 ?

% yes
| ?- max(2,1,Max).
Max = 2 ?
```

```
% yes
```

Errors

Since `A` and `B` are evaluated, errors may be thrown by `eval/2`.

See also

None.

4.1.128 `member/2`

Synopsis

```
member(?Element, ?List)
```

Description

Succeeds if `Element` is a member of `List`.

Examples

```
| ?- member(1, [1,2,3]).
% yes
| ?- member(4, [1,2,3]).
% no
| ?- member(A, B).
A = _22816
B = [_22816|_29616] ?

% yes
```

Errors

None.

See also

None.

4.1.129 `message_hook/3`

Synopsis

```
message_hook(+Severity, +Message, +Lines)
```

Description

Prints a message with a severity prefix. The following are the valid severity values for the argument `Severity`:

`banner` The message has no prefix.

`error` The message is prefixed with `!`.

`help` The message has no prefix.

`informational` The message is prefixed with `%`.

`none` The message has no prefix.

`query` The message has no prefix.

`silent` The message is not output.

`warning` The message is prefixed with `*`.

The arguments `Message` and `Lines` are two different representations of the same message. This predicate only uses `Lines` which are displayed on the stream `user_error`. The reason both are given is to provide a user hook registered with `add_message_hook/1` more information so that it can generate more useful error messages. The usual method of generating `Lines` from `Message` is:

```
    generate_message(Msg, L, []),
    generate_message_lines(Lines, L, [])
```

Examples

```
| ?- Msg = error(instantiation_error,
                  instantiation_error(variable(V), 1)),
     generate_message(Msg, L, []),
     generate_message_lines(Lines, L, []),
     message_hook(error, Msg, Lines).
! ERROR
! Error class : instantiation error
! Goal in error : variable(_628736)
! This exception was thrown because argument number 1 was
  not sufficiently instantiated.
Msg = error(instantiation_error,
            instantiation_error(variable(_628736),1))
V = _628736
L = [ERROR -[],nl,
     Error class : instantiation error-[],nl,
```

```
     Goal in error : ~q-[variable(_628736)],nl,
     This exception was thrown because argument number
     ~q was not sufficiently instantiated.-[1],nl]
Lines = [[ERROR -[]],
         [Error class : instantiation error-[]],
         [Goal in error : ~q-[variable(_628736)]],
         [This exception was thrown because argument
          number ~q was not sufficiently instantiated.
         -[1]]] ?

% yes
```

Errors

None.

See also

add_generate_message/1, generate_message_lines/3.

4.1.130 min/3

Synopsis

```
min(+A, +B, ?C)
```

Description

Succeeds if `C` is the minimum of `A` and `B`. This predicate behaves as if it were defined as:

```
min(A,B,C) :- A =< B, !, C = A.
min(_,B,B).
```

Examples

```
| ?- min(1,2,Min).
Min = 1 ?

% yes
| ?- min(2,1,Min).
Min = 1 ?

% yes
```

Errors

Since `A` and `B` are evaluated, errors may be thrown by `eval/2`.

See also

None.

4.1.131 `name/2`

Synopsis

```
name(+A, ?C)
name(?A, +C)
```

Description

Succeeds if the argument `C` is a list of character codes which represent the argument `A`. `A` can be a number or a symbol.

Examples

```
| ?- name(A, "foo").
A = foo ?

% yes
| ?- name(99, C).
C = [57,57] ?

% yes
```

Errors

`instantiation_error` Both arguments were uninstantiated.

See also

`atom_codes/2`, `number_codes/2`.

4.1.132 `nl/0`

Synopsis

```
nl
```

Description

Equivalent to `current_output(S), nl(S)`.

Examples

See nl/1.

Errors

See nl/1.

See also

current_output/1, nl/1.

4.1.133 nl/1

Synopsis

```
nl(S)
```

Description

Behaves as if it were defined as follows:

```
nl(S) :-
        put_code(S, 13),
        put_code(S, 10),
        flush_output(S).
```

Examples

```
| ?- print(a), nl, print(user_error, b), nl(user_error).
a
b
% yes
```

Errors

`instantiation_error` The argument `S` was not instantiated.

`domain_error(stream_or_alias, S)` The argument `S` was neither a stream term nor an alias.

`existence_error(stream, S)` The argument `S` referred to a nonexistent stream.

`permission_error(output, binary_stream, S` The argument `S` referred to a binary stream.

`permission_error(output, stream, S` The argument `S` referred to an input stream.

See also

flush_output/1, put_code/2.

4.1.134 number/1

Synopsis

```
number(+T)
```

Description

Succeeds if T is a number. A number is either a floating-point number or an integer. This predicate behaves as if it was defined as follows:

```
number(T) :- integer(T), !.
number(T) :- float(T).
```

Examples

```
| ?- number(9).
% yes
| ?- number(9.0).
% yes
| ?- number(nine).
% no
```

Errors

None.

See also

float/1, integer/1.

4.1.135 numbervars/3

Synopsis

```
numbervars(+Term, +Start, -End)
```

Description

The variables in Term are unified with terms of the form '$VAR'(C) where C is a unique integer identifier for each such variable. This identifier takes the form of a counter which has the initial value Start and a final value of End-1.

When the predicate write_term/3 is called with the option argument containing numbervars(true), and a argument which contains a term of the form '$VAR'(C) — where C is a positive integer — then instead of '$VAR'(C), a Prolog variable is written. The name of this variable is unique for each individual C.

If C is less than 26 then the variable name is the single upper case letter identified with the C^{th} element of the English alphabet. The letter "A" is considered to be the zeroth element of the alphabet. Example:

```
| ?- writeq('$VAR'(0)), writeq('$VAR'(25)).
AZ
% yes
```

If C is greater than 25, then the variable name takes the form of a letter followed by a sequence of digits. The letter is calculated to be the n^{th} element of the English alphabet where n is C modulo 26, and the sequence of digits is the decimal representation of the integer division of C and 26. Example:

```
| ?- writeq('$VAR'(26)), writeq('$VAR'(52)), nl.
A1A2
% yes
```

Examples

```
| ?- Term = foo(X,Y,Z), numbervars(Term, 0, Count),
     writeq(Term), nl.
foo(A,B,C)
Term = foo($VAR(0),$VAR(1),$VAR(2))
X = $VAR(0)
Y = $VAR(1)
Z = $VAR(2)
Count = 3 ?

% yes
```

Errors

instantiation_error The argument Start was not instantiated.

type_error(integer, Start) The argument Start was not an integer.

domain_error(not_less_than_zero, Start) The argument Start must not be less than zero.

See also

write_term/3, writeq/2.

4.1.136 nonvar/1

Synopsis

```
nonvar(+T)
```

Description

Succeeds if `T` is not a variable. This predicate behaves as if it was defined as follows:

```
nonvar(T) :- \+ var(T).
```

Examples

```
| ?- nonvar(9).
% yes
| ?- nonvar(foo(X)).
X = _523056 ?

% yes
| ?- nonvar(X).
% no
```

Errors

None.

See also

`var/1`.

4.1.137 '\+'/1

Synopsis

```
'\+'(+T)
```

Description

Succeeds if and only if `call(T)` fails. This is negation by failure. This predicate behaves as if it were defined as follows:

```
'\+'(T) :- call(T) -> fail ; true.
```

Examples

```
| ?- \+ fail.
% yes
| ?- \+ true.
% no
```

Errors

`instantiation_error` The argument `T` was not instantiated.

`type_error(callable, T)` The argument `T` was not a callable term.

See also

None.

4.1.138 nth0/3

Synopsis

```
nth0(?N, +L, +E)
nth0(+N, +L, ?E)
```

Description

Succeeds if the list `L` at index `N` contains the element `E`. The first element has index 0. This predicate is deterministic.

Examples

```
| ?- nth0(N, [a,b,c], c).
N = 2 ?

% yes
| ?- nth0(1, [a,b,c], E).
E = b ?

% yes
```

Errors

None.

See also

nth1/3.

4.1.139 number_base_codes/3

Synopsis

```
number_base_codes(+N, +B, ?L)
```

Description

Succeeds if the name of the number `N` corresponds to the list of character codes `L` represented in base `B`. Acceptable values for `B` satisfy both `2 =< B` and `B =< 36`.

Examples

```
| ?- number_base_codes(99, 16, L), atom_codes(A, L).
L = [49,54,39,54,51]
A = 16'63 ?

% yes
```

Errors

`domain_error(number_base, B)` The argument `B` was an integer but it was either less than `2` or greater than `36`.

`instantiation_error` Either the argument `N` or the argument `B` was an uninstantiated variable.

`type_error(integer, B)` The argument `B` was instantiated but it was not an integer.

`type_error(number, N)` The argument `N` was instantiated but it was not a number.

See also

`number_codes/2`, `name/2`.

4.1.140 number_chars/2

Synopsis

```
number_chars(+N, ?L)
number_chars(?N, +L)
```

Description

Succeeds if the name of the number `N` corresponds to the list of characters `L`.

Examples

```
| ?- number_chars(N, ['3', '.', '1', '4']).
N = 3.14 ?

% yes
| ?- number_chars(3.14, C).
C = [3,.,1,4] ?

% yes
```

Errors

`instantiation_error` Either both arguments were uninstantiated, or, `N` was uninstantiated and `L` was not ground.

`type_error(number, N)` The argument `N` was instantiated but it was not a number.

`type_error(list, L)` The argument `L` was instantiated but it was not a list.

`type_error(character, E)` The argument `L` contained an element `E` which was not a character.

`syntax_error(badly_formed_number)` The argument `L` could not be parsed.

See also

`character/1`, `number_codes/2`.

4.1.141 number_codes/2

Synopsis

```
number_codes(+N, ?L)
number_codes(?N, +L)
```

Description

Succeeds if the name of the number `N` corresponds to the list of character codes `L`.

Examples

```
| ?- number_codes(3.14, C).
C = [51,46,49,52] ?
```

```
% yes
| ?- number_codes(N, "3.14").
N = 3.14 ?

% yes
```

Errors

`instantiation_error` Either both arguments were uninstantiated, or, `N` was uninstantiated and `L` was not ground.

`type_error(number, N)` The argument `N` was instantiated but it was not a number.

`type_error(list, L)` The argument `L` was instantiated but it was not a list.

`representation_error(character_code)` The argument `L` contained an element which was not a character code.

`syntax_error(badly_formed_number)` The argument `L` could not be parsed.

See also

`character_code/1`, `number_chars/2`, `name/2`.

4.1.142 once/1

Synopsis

`once(+T)`

Description

Succeeds if and only if `call(T)` succeeds. This predicate succeeds only once, hence the name. This predicate behaves as if it were defined as follows:

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

Examples

```
| ?- once((true ; true)), fail.
% no
```

Errors

`instantiation_error` The argument `T` was not instantiated.

`type_error(callable, T)` The argument `T` was not a callable term.

See also

None.

4.1.143 `op/3`

Synopsis

```
op(+P, +S, +A)
```

Description

Creates or updates the entry in the operator table for the atom(s) `A` with operator specifier `S` and priority `P`. The operator table is used when parsing and printing terms.

The argument `A` is either an atom or a list of atoms. When it is a list, the table entries are created/updated for each atom in the list. Any such atom is the name of the operator being processed.

The argument `S` is an atom and it specifies the fixity and associativity of the operator. It must be one of the following:

`fx` The operator is prefix and non-associative.

`fy` The operator is prefix and right-associative.

`xfx` The operator is infix and non-associative.

`xfy` The operator is infix and right-associative.

`yfx` The operator is infix and left-associative.

`xf` The operator is postfix and non-associative.

`yf` The operator is postfix and left-associative.

The argument `P` is an integer and it specifies the priority of the operator. The range of this argument is between `0` and `1200` (inclusive). The lower of two priorities has a higher precedence, that is to say it binds more tightly.

It is not possible to call `op/3` to alter the status of the operator `','`. A call of `op/3` where the priority `P` is `0` will remove the entry associated with atom `A` and specifier `S`. There can be no two operators with the same specifier and name. There can be no two operators with the same name where one has a prefix specifier and the other has a postfix specifier.

Examples

```
| ?- op(200, xfx, @).
% yes
| ?- op(300, xfx, &).
% yes
| ?- write_canonical(a @ b & c), nl.
&(@(a,b),c)
```

Errors

`domain_error(operator_priority, P)` The argument `P` was either less than `0` or greater than `1200`.

`domain_error(operator_specifier, S)` The argument `S` was not a valid operator specifier.

`instantiation_error` One of the arguments `P`, `S`, or `A` was not ground.

`permission_error(create, operator, A)` An attempt was made to create the operator `A` but this couldn't be done.

`permission_error(modify, operator, ',')` An attempt was made to modify the operator `','`.

`type_error(atom, S)` The argument `S` was not an atom.

`type_error(integer, P)` The argument `P` was not an integer.

`type_error(list, A)` The argument `A` was not an atom and it was not a list.

See also

`current_op/3`, `infix_op_specifier/1`, `postfix_op_specifier/1`, `prefix_op_specifier/1`.

4.1.144 open/3

Synopsis

```
open(+F, +M, -S)
```

Description

Equivalent to `open(F, M, S, [])`.

Examples

See open/4.

Errors

See open/4.

See also

open/4.

4.1.145 open/4

Synopsis

```
open(+F, +M, -S, +O)
```

Description

Opens the file specified by the path F, mode M, and options O. The result is the stream term S which can be used to input or output data. The argument F is passed to absolute_file_name/2 and it is the result of this that is the specified file.

The argument M must be one of the following:

read This is an input stream. The position of the stream is set to the beginning of the file.

write This is an output stream. If the file does not exist then create it. The position of the stream is set to the beginning of the file.

append This is an output stream. If the file does not exist then create it. The position of the stream is set to the end of the file.

The argument O is a list of terms. Each element of this list must be one of the following:

alias(A) The atom A can be used instead of the stream term S to refer to this stream. Two streams may not have the same alias. In this case a permission error exception is thrown.

character_encoding(ascii) This is an ASCII encoded stream.

character_encoding(latin_1) This is an Latin-1 (ISO 8859-1) encoded stream.

character_encoding(utf_8) This is an Unicode UTF-8 encoded stream.

`character_encoding(utf_16be)` This is an Unicode UTF-16BE encoded stream.

`character_encoding(utf_16le)` This is an Unicode UTF-16LE encoded stream.

`character_encoding(utf_32be)` This is an Unicode UTF-32BE encoded stream.

`character_encoding(utf_32le)` This is an Unicode UTF-32LE encoded stream.

`eof_action(error)` Upon reading past the end of a file, throw a permission error exception.

`eof_action(eof_code)` Upon reading past the end of a file, return an end of file code.

`eof_action(reset)` Upon reading past the end of a file, reposition the stream so that it is not past the end of file.

`reposition(false)` This stream may not be repositioned.

`reposition(true)` This stream may be repositioned.

`type(binary)` This is a binary stream.

`type(text)` This is a text stream.

If any two terms in `O` conflict with each other, then the last of the two terms given takes precedence. The default options are `type(text)`, `reposition(false)`, `eof_action(eof_code)`, and `character_encoding(utf_8)`

Examples

```
test :-
       open('test_file.txt', write, Stream),
       write(Stream, 'hello.'),
       nl(Stream),
       close(Stream).
```

Errors

`domain_error(io_mode, M)` The argument `M` was not a valid I/O mode.

`domain_error(source_sink, F)` The argument `F` was not a valid source/sink.

`domain_error(stream_option, O)` The argument `O` contained an invalid option.

`existence_error(source_sink, F)` The file referenced by `F` does not exist.

`instantiation_error` The arguments `F`, `M`, and `O` were not ground.

`permission_error(open, source_sink, F)` There was a lack of permission to open `F`.

`type_error(atom, M)` The argument `M` was not an atom.

`type_error(list, O)` The argument `O` was not a list.

`type_error(variable, S)` The argument `S` was not a variable.

See also

absolute_file_name/2, close/1, io_mode/1, source_sink/1, stream_property/2.

4.1.146 op_specifier/1

Synopsis

```
op_specifier(?T)
```

Description

Succeeds if `T` is an operator specifier.

Examples

```
| ?- findall(O, op_specifier(O), Ops).
O = _522304
Ops = [fx,fy,xfx,xfy,yfx,xf,yf] ?

% yes
```

Errors

None.

See also

current_op/3, op/3, infix_op_specifier/1, postfix_op_specifier/1, prefix_op_specifier/1.

4.1.147 partial_list/1

Synopsis

```
partial_list(?T)
```

Description

Succeeds if `T` is a partial list. The collection of partial lists is the smallest one satisfying:

(i) all variables are partial lists,

(ii) the atom `[]` is a partial list,

(iii) if `P` is a partial list, then so is `[_|P]`.

Examples

```
| ?- partial_list([_|_]).
% yes
| ?- partial_list(_).
% yes
| ?- partial_list([]).
% yes
```

Errors

None.

See also

var/1.

4.1.148 peek_byte/1

Synopsis

```
peek_byte(?B)
```

Description

Behaves as if it were defined as follows:

```
peek_byte(Byte) :-
        current_input(S),
        peek_byte(S, Byte).
```

Examples

See peek_byte/2.

Errors

See peek_byte/2.

See also

current_input/1, peek_byte/2.

4.1.149 peek_byte/2

Synopsis

```
peek_byte(+S, ?B)
```

Description

Succeeds if B would be the next byte read from the binary stream S.

Examples

```
test_peek_byte :-
        open('test_file.bin', read, Stream, [type(binary)]),
        peek_byte(Stream, Byte),
        get_byte(Stream, Byte),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument S is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream S does not exist.

`instantiation_error` The argument S was not instantiated.

`permission_error(input, past_end_of_stream, S)` Tried to read past the end of the stream.

`permission_error(input, stream, S)` There is a lack of permission to read from stream S.

`permission_error(input, text_stream, S)` The argument S refers to a text stream.

`type_error(in_byte, B)` The argument B was instantiated but not a valid input byte.

See also

in_byte/1, get_byte/2.

4.1.150 peek_char/1

Synopsis

```
peek_char(?C)
```

Description

Behaves as if it were defined as follows:

```
peek_char(Char) :-
        current_input(S),
        peek_char(S, Char).
```

Examples

See peek_char/2.

Errors

See peek_char/2.

See also

current_input/1, peek_char/2.

4.1.151 peek_char/2

Synopsis

```
peek_char(+S, ?C)
```

Description

Succeeds if `C` would be the next character read from the text stream `S`.

Examples

```
test_peek_char :-
        open('test_file.txt', read, Stream),
        peek_char(Stream, C),
        get_char(Stream, C),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` The argument `S` was not instantiated.

`permission_error(input, past_end_of_stream, S)` Tried to read past the end of the stream.

`permission_error(input, stream, S)` There is a lack of permission to read from stream `S`.

`permission_error(input, binary_stream, S)` The argument `S` refers to a binary stream.

`type_error(in_character, C)` The argument `C` was instantiated but not a valid input character.

See also

`get_char/2`.

4.1.152 peek_code/1

Synopsis

```
peek_code(?C)
```

Description

Behaves as if it were defined as follows:

```
peek_code(Code) :-
        current_input(S),
        peek_code(S, Code).
```

Examples

See `peek_code/2`.

Errors

See `peek_code/2`.

See also

`current_input/1`, `peek_code/2`.

4.1.153 peek_code/2

Synopsis

```
peek_code(+S, ?C)
```

Description

Succeeds if `C` would be the next character code read from the text stream `S`.

Examples

```
test_peek_code :-
        open('test_file.txt', read, Stream),
        peek_code(Stream, C),
        get_code(Stream, C),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` The argument `S` was not instantiated.

`permission_error(input, past_end_of_stream, S)` Tried to read past the end of the stream.

`permission_error(input, stream, S)` There is a lack of permission to read from stream `S`.

`permission_error(input, binary_stream, S)` The argument `S` refers to a binary stream.

`representation_error(in_character_code, C)` The argument `C` was instantiated but was not a valid input character code.

`type_error(integer, C)` The argument `C` was instantiated but was not an integer.

See also

None.

4.1.154 phrase/2

Synopsis

```
phrase(+R, ?L)
```

Description

Behaves as if it were defined as follows:

```
phrase(R, L) :-
        phrase(R, L, []).
```

Examples

See phrase/3.

Errors

See phrase/3.

See also

phrase/3.

4.1.155 phrase/3

Synopsis

```
phrase(+R, ?L0, ?L1)
```

Description

Succeeds if there is a list L2 such that append(L2, L1, L0) and L2 belongs to the language of definite clause grammar rule R. To put it another way, some prefix of the list L0 can be generated/accepted by the definite clause grammar rule R and what follows after this prefix is the list L1.

Examples

```
| ?- expand_term((s --> np, vp), T), assert(T).
T = s(_546656,_546624) :- np(_546656,_547920),vp(_547920,_546624) ?

% yes
| ?- expand_term((np --> det, noun), T), assert(T).
T = np(_547552,_547520) :- det(_547552,_548816),noun(_548816,_547520) ?

% yes
```

```
| ?- expand_term((vp --> verb, np), T), assert(T).
T = vp(_547328,_547296) :- verb(_547328,_548592),np(_548592,_547296) ?

% yes
| ?- expand_term((det --> [a]), T), assert(T).
T = det(_546320,_546288) :- _546320 = [a|_546288] ?

% yes
| ?- expand_term((det --> [the]), T), assert(T).
T = det(_546768,_546736) :- _546768 = [the|_546736] ?

% yes
| ?- expand_term((noun --> [cat]), T), assert(T).
T = noun(_546992,_546960) :- _546992 = [cat|_546960] ?

% yes
| ?- expand_term((noun --> [dog]), T), assert(T).
T = noun(_546992,_546960) :- _546992 = [dog|_546960] ?

% yes
| ?- expand_term((noun --> [mouse]), T), assert(T).
T = noun(_547440,_547408) :- _547440 = [mouse|_547408] ?

% yes
| ?- expand_term((verb --> [chases]), T), assert(T).
T = verb(_547664,_547632) :- _547664 = [chases|_547632] ?

% yes
| ?- phrase(s, L, []).
L = [a,cat,chases,a,cat] ?

% yes
| ?- phrase(s, [the, dog, chases, the, cat], []).
% yes
| ?- phrase(s, [the, cat, chases, the, mouse, all, day], L).
L = [all,day] ?

% yes
```

Errors

`instantiation_error` The argument `R` was not instantiated.

`type_error(callable, R)` The argument `R` was not a callable term.

See also

Section 4.2.

4.1.156 portray/2

Synopsis

```
portray(+S, +T)
```

Description

Succeeds if there is a portray hook that has been registered with add_portray/1 that succeeds with the same arguments. This predicate behaves as if it was defined as follows:

```
portray(Stream, Term) :-
        current_portray(Functor),
        Goal =.. [Functor, Stream, Term],
        call(Goal), !.
```

Examples

```
| ?- assert((my_portray(S, foo) :- write(S, goo_not_foo), nl)).
S = _524784 ?

% yes
| ?- add_portray(my_portray).
% yes
| ?- portray(user_output, foo).
goo_not_foo
% yes
```

Errors

None.

See also

add_portray/1, current_portray/1, del_portray/1.

4.1.157 portray_clause/1

Synopsis

```
portray_clause(+C)
```

Description

This predicate behaves as if it was defined as follows:

```
portray_clause(C) :-
        current_output(S),
        portray_clause(S, C).
```

Examples

See portray_clause/2.

Errors

See portray_clause/2.

See also

current_output/1, portray_clause/2.

4.1.158 portray_clause/2

Synopsis

```
portray_clause(+S, +C)
```

Description

Pretty prints the clause `C` on the stream `S`. This predicate is very useful for programs which generate Prolog code.

Examples

```
| ?- current_output(S),
     portray_clause(S, (a(X):-b(X),!,c(X,Y);d(X)->e(X))).
a(A) :-
    (   b(A),
        !,
        c(A,B)
    ;   d(A) ->
        e(A)
    ).
S = $stream(1)
X = _553488
Y = _557056 ?

% yes
```

Errors

This predicate uses write/2. which may raise errors.

See also

None.

4.1.159 postfix_op_specifier/1

Synopsis

```
postfix_op_specifier(?T)
```

Description

Succeeds if T is an postfix operator specifier.

Examples

```
| ?- findall(O, postfix_op_specifier(O), L).
O = _522304
L = [xf,yf] ?

% yes
```

Errors

None.

See also

current_op/3, op/3, op_specifier/1,
infix_op_specifier/1, prefix_op_specifier/1.

4.1.160 predicate_indicator/1

Synopsis

```
predicate_indicator(+P)
```

Description

Succeeds if P is a valid predicate indicator — P unifies with F/N where F is an atom, and N is an integer that is greater than or equal to 0 and less than the value of the flag max_arity.

Examples

```
| ?- predicate_indicator(predicate_indicator/1).
% yes
| ?- predicate_indicator(X).
% no
```

Errors

None.

See also

atom/1, integer/1.

4.1.161 predication/1

Synopsis

```
predication(+T)
```

Description

Succeeds if `T` is a predication. This definition of predication is:

(i) all atoms are predications, and

(ii) all compound terms are predications.

Examples

```
| ?- predication(foo).
% yes
| ?- predication(foo(_,_)).
% yes
```

Errors

None.

See also

atom/1, compound/1.

4.1.162 prefix_op_specifier/1

Synopsis

```
prefix_op_specifier(?T)
```

Description

Succeeds if `T` is an prefix operator specifier.

Examples

```
| ?- findall(O, prefix_op_specifier(O), L).
O = _522304
L = [fx,fy] ?

% yes
```

Errors

None.

See also

`current_op/3`, `op/3`, `op_specifier/1`, `infix_op_specifier/1`, `postfix_op_specifier/1`.

4.1.163 `print/1`

Synopsis

```
print(+T)
```

Description

Behaves as if it were defined as follows:

```
print(Term) :-
        current_output(S),
        print(S, Term).
```

Examples

See `print/2`.

Errors

See `print/2`.

See also

`current_output/1`, `print/2`.

4.1.164 print/2

Synopsis

```
print(+S, +T)
```

Description

Behaves as if it were defined as follows:

```
print(Stream, Term) :-
        write_term(Stream, Term, [portray(true)]).
```

Examples

See write_term/2.

Errors

See write_term/2.

See also

write_term/2.

4.1.165 print_message/2

Synopsis

```
print_message(+S, +M)
```

Description

Prints the message `M` with severity S. An attempt is made to find an appropriate portray message hook. If such a hook exists it is called. Otherwise, the message is generated and if any generate message hooks exist, they are called. Output is sent to the stream `user_error`. The following are the valid severity values for `S`:

`banner` The message has no prefix.

`error` The message is prefixed with `!`.

`help` The message has no prefix.

`informational` The message is prefixed with `%`.

`none` The message has no prefix.

`query` The message has no prefix.

`silent` The message is not output.

`warning` The message is prefixed with `*`.

Examples

```
| ?- print_message(banner, 'I am a banner').
I am a banner
% yes
| ?- print_message(error, 'I am an error').
! I am an error
% yes
| ?- print_message(help, 'I am help').
I am help
% yes
| ?- print_message(informational, 'I am informational').
% I am informational
% yes
| ?- print_message(none, 'I am none').
I am none
% yes
| ?- print_message(query, 'I am a query').
I am a query
% yes
| ?- print_message(silent, 'I am silent').
% yes
| ?- print_message(warning, 'I am a warning').
* I am a warning
% yes
```

Errors

None.

See also

add_portray_message/1, add_generate_message/1.

4.1.166 print_message_lines/3

Synopsis

```
print_message_lines(+Stream, +Severity, +Lines)
```

Description

This procedure is used internally by print_message/2 to print the message Lines with Severity on Stream. This procedure should be used to generate formatted output in a message hook.

Examples

See add_message_hook/1.

Errors

None.

See also

add_message_hook/1, print_message/2.

4.1.167 private_procedure/1

Synopsis

```
private_procedure(+T)
```

Description

Succeeds if the predicate indicator T identifies a private procedure which is equivalent to a static predicate. This predicate behaves as if it was defined as:

```
private_procedure(P) :- procedure_property(P, (static)).
```

Examples

```
| ?- private_procedure(private_procedure/1).
% yes
```

Errors

See procedure_property/2.

See also

procedure_property/2, public_procedure/1.

4.1.168 procedure_property/2

Synopsis

```
procedure_property(+I, ?P)
```

Description

Succeeds if `I` is a predicate indicator which identifies a compiled predicate and `P=static`, or, `I` is a predicate indicator which identifies an interpreted predicate and `P=(dynamic)`

Examples

```
| ?- assert(foo).
% yes
| ?- procedure_property(foo/0, P).
P = dynamic ?

% yes
| ?- procedure_property(procedure_property/2, P).
P = static ?

% yes
```

Errors

`instantiation_error` The argument `I` was not instantiated.

`type_error(predicate_indicator, I)` The argument `I` was not a predicate indicator.

See also

procedure_property/2.

4.1.169 prolog_lexical_digit/1

Synopsis

```
prolog_lexical_digit(+Code)
```

Description

Succeeds if the character code `Code` represents a numerical character that can be used to start a Prolog integer or floating-point number.

Examples

```
| ?- prolog_lexical_digit(0'5).
% yes
| ?- prolog_lexical_digit(5).
% no
| ?- between(0, 16'22ff, Code), prolog_lexical_digit(Code),
     put_code(Code), fail ; nl.
0123456789
Code = _514512 ?

% yes
```

Errors

None.

See also

prolog_lexical_lower_case_letter/1, prolog_lexical_symbol/1, prolog_lexical_upper_case_letter/1, prolog_lexical_ws/1.

4.1.170 prolog_lexical_letter/1

Synopsis

```
prolog_lexical_letter(+Code)
```

Description

Succeeds if the character code `Code` represents an alphabetical character. Behaves as if it were defined as follows:

```
prolog_lexical_letter(CodePoint) :-
        (   prolog_lexical_lower_case_letter(CodePoint)
        ;   prolog_lexical_upper_case_letter(CodePoint)
        ),
        !.
```

Examples

```
| ?- prolog_lexical_letter(0'p).
% yes
| ?- prolog_lexical_letter(0'P).
% yes
```

Errors

None.

See also

prolog_lexical_upper_case_letter/1, prolog_lexical_lower_case_letter/1.

4.1.171 prolog_lexical_lower_case_letter/1

Synopsis

```
prolog_lexical_lower_case_letter(+Code)
```

Description

Succeeds if the character code `Code` represents an alphabetical character that can be used to start a Prolog atom name.

Examples

```
| ?- prolog_lexical_lower_case_letter(0'p).
% yes
| ?- prolog_lexical_lower_case_letter(0'P).
% no
```

If your terminal can display the Latin-1 codepage characters, you can try the following:

```
| ?- between(0, 16'22ff, Code),
     prolog_lexical_lower_case_letter(Code),
     put_code(Code), fail ; nl.
```

Errors

None.

See also

prolog_lexical_digit/1, prolog_lexical_symbol/1, prolog_lexical_upper_case_letter/1, prolog_lexical_ws/1.

4.1.172 prolog_lexical_symbol/1

Synopsis

```
prolog_lexical_symbol(+Code)
```

Description

Succeeds if the character code `Code` represents a symbol (non-alphabetical) character can be used in a Prolog atom name.

Examples

```
| ?- prolog_lexical_symbol(0'=).
% yes
| ?- prolog_lexical_symbol(p).
% no
```

If your terminal can display the Unicode mathematical operators and arrows, then you can try the following:

```
| ?- between(0, 16'22ff, Code), prolog_lexical_symbol(Code),
     put_code(Code), fail ; nl.
```

Errors

None.

See also

`prolog_lexical_digit/1`, `prolog_lexical_lower_case_letter/1`, `prolog_lexical_upper_case_letter/1`, `prolog_lexical_ws/1`.

4.1.173 prolog_lexical_upper_case_letter/1

Synopsis

```
prolog_lexical_upper_case_letter(+Code)
```

Description

Succeeds if the character code `Code` represents an alphabetical character that can be used to start a Prolog variable name.

Examples

```
| ?- prolog_lexical_upper_case_letter(0'P).
% yes
| ?- prolog_lexical_upper_case_letter(0'p).
% no
```

If your terminal can display the Latin-1 codepage characters, you can try the following:

```
| ?- between(0, 16'22ff, Code),
     prolog_lexical_upper_case_letter(Code),
     put_code(Code), fail ; nl.
```

Errors

None.

See also

prolog_lexical_digit/1, prolog_lexical_lower_case_letter/1, prolog_lexical_symbol/1, prolog_lexical_ws/1.

4.1.174 prolog_lexical_ws/1

Synopsis

```
prolog_lexical_ws(+Code)
```

Description

Succeeds if the character code `Code` represents a character can be used as whitespace in Prolog source. All whitespace input is ignored.

Examples

```
| ?- prolog_lexical_upper_case_letter(0'P).
% yes
| ?- prolog_lexical_upper_case_letter(0'p).
% no
```

Errors

None.

See also

prolog_lexical_digit/1, prolog_lexical_lower_case_letter/1, prolog_lexical_symbol/1, prolog_lexical_upper_case_letter/1.

4.1.175 prompt/1

Synopsis

```
prompt(+T)
prompt(-T)
```

Description

This predicate is used to set or query the prompt term displayed when input is read from the user. If `T` is an uninstantiated variable, then it is unified with the prompt term. If `T` is not a variable, then the prompt term is set to `T`. Note that the top-level loop calls `prompt('|:')` before asking for input.

Examples

```
| ?- prompt('OK > '), read(X).
OK > hello.
X = hello ?

% yes
| ?- prompt(T).
T = |:  ?

% yes
```

Errors

None.

See also

None.

4.1.176 public_procedure/1

Synopsis

```
public_procedure(+T)
```

Description

Succeeds if the predicate indicator `T` identifies a public procedure which is equivalent to a dynamic predicate. This predicate behaves as if it was defined as:

```
public_procedure(P) :- procedure_property(P, (dynamic)).
```

Examples

```
| ?- assert(public_pred).
% yes
| ?- public_procedure(public_pred/0).
% yes
```

Errors

See `procedure_property/2`.

See also

`procedure_property/2`, `private_procedure/1`.

4.1.177 put_byte/1

Synopsis

```
put_byte(?B)
```

Description

Behaves as if it were defined as follows:

```
put_byte(Byte) :-
        current_output(S),
        put_byte(S, Byte).
```

Examples

See `put_byte/2`

Errors

See `put_byte/2`

See also

`current_output/1`, `put_byte/2`.

4.1.178 put_byte/2

Synopsis

```
put_byte(+S, ?B)
```

Description

Succeeds if the byte `B` can be written to the binary stream `S`.

Examples

```
test_put_byte :-
        open('test_file.bin', write, Stream, [type(binary)]),
        put_byte(Stream, Byte),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` One of the arguments, `S` or `B`, was not instantiated.

`permission_error(output, stream, S)` There is a lack of permission to write to stream `S`.

`permission_error(output, text_stream, S)` The argument `S` refers to a text stream.

`type_error(byte, B)` The argument `B` was instantiated but not a valid byte.

See also

`byte/1`.

4.1.179 put_char/1

Synopsis

```
put_char(?C)
```

Description

Behaves as if it were defined as follows:

```
put_char(Char) :-
        current_output(S),
        put_char(S, Char).
```

Examples

See `put_char/2`.

Errors

See put_char/2.

See also

current_output/1, put_char/2.

4.1.180 put_char/2

Synopsis

```
put_char(+S, ?C)
```

Description

Succeeds if the character `C` can be written to the text stream `S`.

Examples

```
test_put_char :-
        open('test_file.txt', write, Stream),
        put_char(Stream, C),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` One of the arguments, `S` or `C`, was not instantiated.

`permission_error(output, stream, S)` There is a lack of permission to write to stream `S`.

`permission_error(output, binary_stream, S)` The argument `S` refers to a binary stream.

`type_error(character, C)` The argument `C` was instantiated but not a valid character.

See also

character/1.

4.1.181 put_code/1

Synopsis

```
put_code(?C)
```

Description

Behaves as if it were defined as follows:

```
put_code(Code) :-
        current_output(S),
        put_code(S, Code).
```

Examples

See put_code/2.

Errors

See put_code/2.

See also

current_output/1, put_code/2.

4.1.182 put_code/2

Synopsis

```
put_code(+S, ?C)
```

Description

Succeeds if the character code `C` can be written to the text stream `S`.

Examples

```
test_put_code :-
        open('test_file.txt', write, Stream),
        put_code(Stream, C),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` One of the arguments, `S` or `C`, was not instantiated.

`permission_error(output, stream, S)` There is a lack of permission to write to stream `S`.

`permission_error(output, binary_stream, S)` The argument `S` refers to a binary stream.

`representation_error(character_code, C)` The argument `C` was instantiated but was not a valid character code.

`type_error(integer, C)` The argument `C` was instantiated but was not an integer.

See also

`character_code/1`, `integer/1`.

4.1.183 query_class/5

Synopsis

```
query_class(+Class, ?Prompt, ?InputMethod, ?MapMethod, ?FailureMode)
```

Description

This predicate is used internally by `ask_query/4`. The predicate `Class` names the query class being defined. This will correspond to the first argument of `ask_query/4`. The arguments `Prompt` and `InputMethod` are used in a subsequent call:

```
query_input(InputMethod, Prompt, Input).
```

The argument `MapMethod` is used along with `Input` from the previous call:

```
query_map(MapMethod, Input, Result, Answer),
```

The predicate `ask_query/4` will then use the argument `FailureMode` to determine what to do if `Result` is unified with `failure` in the previous call.

The predefined query classes are:

```
query_class(query, '| ?- ',
           (T-Vs)^term(T, [variable_names(Vs)]), =, query).
query_class(solution, ' ? ',
            line, char([yes-[0'\n], no-";"]), help_query).
query_class(goal, '| :- ', term([]), =, query).
query_class(yes_or_no, ' (y or n) ',
            line, char([yes-"Yy", no-"Nn"]), help_query).
query_class(yes_no_proceed, ' (y, n, p, s, a, or ?) ', line,
            char([yes-"Yy", no-"Nn", proceed-"Pp",
                  suppress-"Ss", abort-"Aa"]),
            help_query).
```

Examples

None.

Errors

None.

See also

ask_query/4, add_query_class_hook/1,
query_input/3, query_map/4.

4.1.184 query_input/3

Synopsis

```
query_input(+InputMethod, +Prompt, ?Input)
```

Description

This predicate is used internally by ask_query/4 where it reads prompted input from the user. The argument `Input` is the final result. The argument `InputMethod` determines how the input is gathered. Possible values for this argument are:

`line` The input is read using query_read_line(user_input, Input).

`term(Options)` The input is read using read_term(Input, Options).

`FinalTermˆterm(Term,Options)` Input is read using read_term(Term, Options) and `Input` is unified with `FinalTerm`.

The argument `Prompt` is a term that is passed to prompt/1 before input is gathered The old prompt is restored upon completion of input.

Examples

```
| ?- query_input(line, 'ok>', Line).
ok>This is a line.
Line = [84,104,105,115,32,105,115,32,97,32,108,105,110,101,46,10] ?

% yes
```

Errors

None.

See also

add_query_input_hook/1, ask_query/4, query_read_line/2

4.1.185 query_map/4

Synopsis

```
query_map(+Method, +Input, ?Result, ?Answer)
```

Description

This predicate is used internally by ask_query/4. The argument `Input` is the output from a call to query_input/3 which is a list of character codes. This is mapped to the argument `Answer` using a method indicated by `Method`. The success of this mapping is indicated by unifying `Result` with either `success` or `failure`. The argument `Method` is one of the following:

`char(Pairs)` Here `Pairs` should be a list of key pairs. The keys are terms with lists of character codes as corresponding values. This map method will search `Pairs` for the first character code, say `Code`, in `Input` that is either a newline (`Code =:= 10`) or non-whitespace (`Code > 32`). The corresponding key is unified with `Answer`.

`=` The arguments `Answer` and `Input` are unified.

Examples

```
| ?- query_map(char([yes-"Yy", no-"Nn"]), "yeah", Answer, Result).
Answer = success
Result = yes ?

% yes
| ?- query_map(char([yes-"Yy", no-"Nn"]), "maybe", Answer, Result).
```

```
Answer = failure
Result = _642464 ?

% yes
```

Errors

None.

See also

ask_query/4, add_query_map_hook/1, key_pair/1.

4.1.186 query_read_line/2

Synopsis

```
query_read_line(+Stream, ?Line)
```

Description

This predicate reads a sequence of character codes from `Stream` and accumulates these codes in `Line`. Reading stops when an end of file code (`-1`) or a newline code (`10`) is encountered. `Line` includes a final newline code but not an end of file code.

Examples

```
| ?- query_read_line(user_input, Line).
|: This is a line.
Line = [84,104,105,115,32,105,115,32,97,32,108,105,110,101,46,10] ?

% yes
```

Errors

See get_code/2.

See also

get_code/2, query_input/3.

4.1.187 read/1

Synopsis

```
read(?T)
```

Description

Behaves as if it were defined as follows:

```
read(Term) :-
        current_input(S),
        read(S, Term).
```

Examples

See read/2.

Errors

See read/2.

See also

current_input/1, read/2.

4.1.188 read/2

Synopsis

```
read(+S, ?T)
```

Description

Behaves as if it were defined as follows:

```
read(S, Term) :-
        read_term(S, Term, []).
```

Examples

See read_term/3.

Errors

See read_term/3.

See also

read_term/3.

4.1.189 read_term/2

Synopsis

```
read_term(?T, +O)
```

Description

Behaves as if it were defined as follows:

```
read_term(Term, Options) :-
        current_input(S),
        read_term(S, Term, Options).
```

Examples

See read_term/3.

Errors

See read_term/3.

See also

current_input/1, read_term/3.

4.1.190 read_term/3

Synopsis

```
read_term(+S, ?T, +O)
```

Description

A term `T` is read from the the stream `S` with auxiliary information determined by the list of options `O`. The valid options are:

`position(P)` `P` is unified with the position of the stream `S` corresponding to the start of the term that was read.

`singletons(V)` `V` is unified with a list of `A=B` pairs such that `atom(A),var(B)` is true and `A` is the name of a variable that appears only once in the term and `B` is the corresponding variable.

`variables(V)` `V` is unified with a list of all the variables read.

`variable_names(V)` `V` is unified with a list of `A=B` pairs such that `atom(A),var(B)` is true and `A` is the name of a variable that appears in the term and `B` is the corresponding variable.

Examples

```
| ?- read_term(T, [position(P), singletons(S),
                   variables(V), variable_names(N)]).
|: foo(A, B, B).
T = foo(_565216,_566080,_566080)
P = $stream_position(0,1,77,0)
S = [A = _565216]
V = [_565216,_566080]
N = [A = _565216,B = _566080] ?

% yes
```

Errors

`domain_error(read_option, Option)` The element `Option` of the list `O` was not a valid read option.

`domain_error(stream_or_alias, S)` The argument `S` was not a valid stream term or alias.

`existence_error(stream, S)` The argument `S` does not refer to an open stream.

`instantiation_error` Either one of the arguments, `T` or `O`, was not instantiated, or, `O` was a list where one of the elements was a variable.

`lexical_error(end_of_file)` The input stream ended in the middle of a token.

`lexical_error(strange_character(Char))` The character code `Char` was found inside a token.

`permission_error(input, binary_stream, S)` The stream `S` was not a text stream.

`permission_error(input, past_end_of_stream, S)` An attempt was made to read past the end of the stream `S`.

`permission_error(input, stream, S)` There was no permission to read from stream `S`.

`syntax_error(badly_formed_list)` A list was not ended with a close bracket.

`syntax_error(bracket_expected_in_arguments)` There was a missing bracket in the input term.

`syntax_error(cannot_start_an_expression(Token))` The token `Token` started an expression.

`syntax_error(expression_expected)` There was a missing expression.

`syntax_error(follows_expression(Token))` The token `Token` immediately follows an expression.

`syntax_error(not_all_read)` Not all of the term was read at the end if the input.

`syntax_error(operator_expected_after_expression)` Two adjacent terms were read.

`syntax_error(prefix_operator_precedence(Op, Prec))` The operator `Op` with precedence `Prec` immediately followed a prefix operator.

`syntax_error(token_or_operator_expected(Token))` The token `Token` was expected but not read.

`type_error(list, O)` The argument `O` was not a list.

See also

None.

4.1.191 `reconsult/1`

Synopsis

```
reconsult(+F)
```

Description

Opens the file specified by `F` and loads the terms found in the file into the clause store. Note that since reconsult uses `open/4`, the argument `F` is in turn passed to `absolute_file_name/2` for translation.

Note that this is a reconsult and not a consult. Should you reconsult the same file twice then you will have only one copy of all read terms in the clause store.

Examples

```
| ?- reconsult('slask/foo').
% yes
```

Errors

`instantiation_error` The argument `F` was not instantiated.

See also

absolute_file_name/2, consult/1, '.'/2.

4.1.192 repeat/0

Synopsis

```
repeat
```

Description

Succeeds and creates a looping choice-point. Behaves as if defined as follows:

```
repeat.
repeat :- repeat.
```

Examples

```
skip_to_end_of_text_file(Stream) :-
        repeat,
        get_char(Stream, _),
        at_end_of_stream(Stream), % failure loops at repeat/0
        !.                        % remove choicepoint
```

Errors

None.

See also

None.

4.1.193 retract/1

Synopsis

```
retract(+T)
```

Description

Removes a dynamic clause from the clause store which unifies with the argument `T`. If the argument is not a compound term with a principle functor of `':-'/2` then this call will behave as that of a new call of `retract((T :- true))`.

This predicate will succeed as many times as there are clauses in the clause store which unify with `T`.

Examples

```
| ?- assert(g(1)), assert(g(2)).
% yes
| ?- g(X).
X = 1 ? ;

X = 2 ? ;

% no
| ?- retract(g(X)), fail.
% no
| ?- clause(g(_), C).
% no
```

Errors

`instantiation_error` The head of the argument clause `T` was not instantiated.

`permission_error(access, static_procedure, F/N)` The head of the argument clause `T` has a predicate indicator of `F/N` and this identifies a static procedure.

`type_error(callable, H)` The head of the argument clause `T` was `H` and this is not a callable term.

See also

`assert/1`, `callable_term/1`.

4.1.194 `retractall/1`

Synopsis

```
retractall(+T)
```

Description

Removes all dynamic clauses from the clause store which have a head term that unifies with the argument `T`. Behaves as if it were defined as follows:

```
retractall(Head) :-
        retract((Head :- _)),
        fail
    ;
        true.
```

Examples

```
| ?- assert(foo(1)), assert(foo(2)).
% yes
| ?- listing(foo/1).
foo(1).
foo(2).
% yes
| ?- retractall(foo(_)).
% yes
| ?- listing(foo/1).
% yes
```

Errors

See `retract/1`.

See also

`retract/1`.

4.1.195 `reverse/2`

Synopsis

```
reverse(?L1, ?L2)
```

Description

Succeeds if the list `L1` is the reverse of `L2`.

Examples

```
| ?- reverse(A, [1,2,3]).
A = [3,2,1] ?

% yes
| ?- reverse(A, B).
A = []
B = [] ? ;

A = [_29760]
B = [_29760] ? ;

A = [_29760,_29840]
B = [_29840,_29760] ? ;
```

```
A = [_29760,_29840,_29920]
B = [_29920,_29840,_29760] ?
```

Errors

None.

See also

None.

4.1.196 seek/4

Synopsis

```
seek(+Stream, +Offset, +Whence, -Result)
```

Description

Sets the byte offset of the stream indicated by `Stream` to a new byte offset determined by `Offset` and `Whence`, then unifies this new offset with `Result`. `Stream` can be either a binary or a text stream. The byte offset is not necessarily the same as the character or character code offset such as in the case of a Unicode text stream. The argument `Offset` is an integer value greater than or equal to zero. The argument `Whence` is an atom and it determines how `Offset` is to be used. Possible values for `Whence` are:

`bof` The stream offset is set to the beginning of the file plus `Offset`.

`current` The stream offset is set to the current position of the file plus `Offset`.

`eof` The stream offset is set to the end of the file plus `Offset`.

Examples

```
| ?- open('test_seek.txt', write, Stream1,
          [alias(alias), reposition(true)]),
     seek(Stream1, 0, bof, 0),
     seek(Stream1, 3, bof, 3),
     seek(Stream1, 0, eof, 0),
     seek(Stream1, 0, bof, 0),
     write(Stream1, 'efg'),
     seek(Stream1, 0, current, 3),
     close(Stream1).
Stream1 = $stream(3) ?
```

```
% yes
```

Errors

`domain_error(not_less_than_zero, Offset)` The argument `Offset` was an integer less than zero.

`domain_error(seek_whence, Whence)` The argument `Whence` was not valid.

`domain_error(stream_or_alias, Stream)` The argument `Stream` was not a valid stream term or alias atom.

`existence_error(stream, Stream` The argument `Stream` refers to a nonexistent stream.

`instantiation_error` One of the arguments `Stream`, `Offset`, or `Whence` was not instantiated.

`permission_error(reposition, stream, Stream)` Either the argument `Stream` refers to a stream that was not opened with the option `reposition(true)`, or, the underlying file system did not permit the changing of the byte offset.

`type_error(atom, Whence)` The argument `Whence` was not an atom.

`type_error(integer, Offset)` The argument `Offset` was not an integer.

See also

open/4, set_stream_position/2.

4.1.197 set_input/1

Synopsis

```
set_input(+T)
```

Description

Instructs the system to use the stream identified by the argument `T` as the default input stream.

Examples

```
| ?- current_input(S).
S = $stream(0) ?

% yes
| ?- open('foo.txt', read, InStream),
     set_input(InStream),
     current_input(InStream),
     close(InStream).
InStream = $stream(3) ?

% yes
| ?- current_input(S).
S = $stream(0) ?

% yes
```

Errors

`domain_error(stream_or_alias, T)` The argument `T` was not a valid stream term or alias atom.

`existence_error(stream, T)` The argument `T` identified a nonexistent stream.

`instantiation_error` The argument `T` was not instantiated.

`permission_error(input, stream, T)` There was no permission to use the argument `T` as an input stream.

See also

None.

4.1.198 `set_output/1`

Synopsis

`set_output(+T)`

Description

Instructs the system to use the stream identified by the argument `T` as the default output stream.

Examples

```
| ?- current_output(S).
S = $stream(1) ?

% yes
| ?- open('foo.txt', write, OutStream),
     set_output(OutStream),
     current_output(OutStream),
     close(OutStream).
OutStream = $stream(3) ?

% yes
| ?- current_output(S).
S = $stream(1) ?

% yes
```

Errors

`domain_error(stream_or_alias, T)` The argument `T` was not a valid stream term or alias atom.

`existence_error(stream, T)` The argument `T` identified a nonexistent stream.

`instantiation_error` The argument `T` was not instantiated.

`permission_error(output, stream, T)` There was no permission to use the argument `T` as an output stream.

See also

None.

4.1.199 `set_prolog_flag/2`

Synopsis

```
set_prolog_flag(+Flag, +Term)
```

Description

Sets the value of the predefined Prolog flag `Flag` to `Term`.

Examples

```
| ?- current_prolog_flag(F,V).
F = debug
V = off ?

% yes
| ?- set_prolog_flag(debug, on).
% yes
| ?- current_prolog_flag(debug, V).
V = on ?

% yes
| ?- set_prolog_flag(debug, off).
% yes
| ?- current_prolog_flag(debug, V).
V = off ?

% yes
```

Errors

`domain_error(flag_value, Term)` The argument `Term` was not a valid flag value for the argument `Flag`.

`domain_error(prolog_flag, Flag)` The argument `Flag` was not a valid prolog flag.

`instantiation_error` One of the arguments, `Flag` or `Term`, was not instantiated.

`permission_error(modify, flag, Flag)` There was no permission to modify the flag `Flag`.

`type_error(atom, Flag)` The argument `Flag` was not an atom.

See also

`current_prolog_flag/2`.

4.1.200 `set_stream_position/2`

Synopsis

`set_stream_position(+Stream, +Position)`

Description

Sets the position of the stream `Stream` to the position `Position`. The argument `Position` is an internal structure that should be configured with the following predicates:

`stream_position_byte_count(Position, Value)` The number of bytes read from or written to the stream is `Value`.

`stream_position_character_count(Position, Value)` The number of characters read from or written to the stream is `Value`.

`stream_position_line_count(Position, Value)` The number of lines read from or written to the stream is `Value`. The line count for a newly opened file is 1. A line count of 0 is not valid.

`stream_position_line_position(Position, Value)` The position in the current line is `Value`. The first position in a line is 0.

The argument `Position` is used to set the following internal counters that are updated upon the completion of a stream I/O operation:

`byte_count` Updated upon `get_byte/2`, `put_byte/2`, and `unget_byte/2`.

`character_count` Updated upon `get_code/2`, `put_code/2`, `unget_code/2`, `get_char/2`, `put_char/2`, and `unget_char/2`.

`line_count` Updated upon `get_code/2`, `put_code/2`, `unget_code/2`, `get_char/2`, `put_char/2`, and `unget_char/2`.

`line_position` Updated upon `get_code/2`, `put_code/2`, `unget_code/2`, `get_char/2`, `put_char/2`, and `unget_char/2`.

As can be seen, it makes little sense changing the value of `byte_count` for a text stream, and likewise, changing the `character_count` for a binary stream is meaningless. It should be noted that changing the position of a Unicode text stream could place the current position in the middle of a character code and this could lead to errors.

Examples

```
| ?- open('test_set_stream_position',
          write,
          OutStream,
          [reposition(true)]),
     stream_position_byte_count(Pos0, 1),
     stream_position_line_count(Pos0, 2),
     stream_position_character_count(Pos0, 3),
```

```
     stream_position_line_position(Pos0, 4),
     set_stream_position(OutStream, Pos0),
     stream_property(OutStream, position(Pos1)),
     stream_position_byte_count(Pos1, 1),
     stream_position_line_count(Pos1, 2),
     stream_position_character_count(Pos1, 3),
     stream_position_line_position(Pos1, 4),
     close(OutStream).
OutStream = $stream(3)
Pos0 = $stream_position(1,2,3,4)
Pos1 = $stream_position(1,2,3,4) ?

% yes
```

Errors

`domain_error(stream_position, Position)` The argument `Position` was not a valid stream position.

`domain_error(stream_or_alias, Stream)` The argument `Stream` was not a valid stream term or alias atom.

`existence_error(stream, Stream)` The argument `Stream` identified a nonexistent stream.

`instantiation_error` One of the arguments, `Stream` or `Property`, was not instantiated.

`permission_error(reposition, stream, Stream)` There was no permission to reposition the stream `Stream`.

See also

`seek/4`, `stream_position_byte_count/2`, `stream_position_character_count/2`, `stream_position_line_count/2`, `stream_position_line_position/2`.

4.1.201 `setof/3`

Synopsis

```
setof(+Template, +Goal, ?Set)
```

Description

Similar to `bagof/3` but whereas that predicate computes a multiset, this predicate computes a sorted set. Thus, `setof/3` behaves as if it had the following definition:

```
setof(Template, Goal, Set) :-
        bagof(Template, Goal, Bag),
        sort(Bag, Set).
```

Examples

```
| ?- bagof(X, member(X, [3,2,1,1,2,3]), Result).
X = _524928
Result = [3,2,1,1,2,3] ?

% yes
| ?- setof(X, member(X, [3,2,1,1,2,3]), Result).
X = _524928
Result = [1,2,3] ?

% yes
```

Errors

`instantiation_error` The argument `Goal` was uninstantiated.

`type_error(callable, Goal)` The argument `Goal` was not callable.

`type_error(list, Set)` The argument `Set` was neither a list nor a partial list.

See also

`bagof/3`, `findall/3`, `sort/2`.

4.1.202 sort/2

Synopsis

`sort(+T1, ?T2)`

Description

Succeeds if `T2` is a list representation of the sorted contents of the list `T1` where duplicate elements have been removed. The ordering relation used is `'@<'/2`.

Examples

```
| ?- sort([3,2,1,1,2,3], L).
L = [1,2,3] ?
```

```
% yes
```

Errors

`instantiation_error` The argument `T1` was not instantiated.

`type_error(list, T1)` The argument `T1` was not a list.

See also

None.

4.1.203 `source_sink/1`

Synopsis

```
source_sink(+T)
```

Description

Succeeds if the given argument is a term which can be used to specify a source or a sink. This predicate behaves as if it was defined as follows:

```
source_sink(Source_sink) :-
    (   atom(Source_sink) ->
        true
    ;   compound(Source_sink),
        Source_sink =.. [_, Arg],
        source_sink(Arg)
    ).
```

Examples

```
| ?- source_sink(file).
% yes
| ?- source_sink(runtime(file)).
% yes
```

Errors

None.

See also

`absolute_file_name/3`.

4.1.204 `statistics/1`

Synopsis

```
statistics(+S)
```

Description

Succeeds if `S` is a key-value list describing the state of the Prolog system. There is one key-value pair for each monitored part of the system. The following is a description of all the keys and their values.

`wall_clock-T T` is the number of milliseconds since midnight before January 1^{st} 1970.

`global_stack_used-B B` is the number of bytes of the global stack that is currently being used.

`global_stack_free-B B` is the number of bytes of the global stack that is currently unused.

`local_stack_used-B B` is the number of bytes of the local stack that is currently being used.

`local_stack_free-B B` is the number of bytes of the local stack that is currently unused.

`trail_used-B B` is the number of bytes of the trail that is currently being used.

`trail_free-B B` is the number of bytes of the trail that is currently unused.

`clause_store_used-B B` is the number of bytes of the clause store that is currently being used.

`clause_store_free-B B` is the number of bytes of the clause store that is currently unused.

`code_used-B B` is the number of bytes of the code section that is currently being used.

`code_free-B B` is the number of bytes of the code section that is currently unused.

`constants_used-B B` is the number of bytes of the constants section that is currently being used.

`constants_free-B B` is the number of bytes of the constants section that is currently unused.

`strings_used-B B` is the number of bytes of the strings section that is currently being used.

`strings_free-B B` is the number of bytes of the strings section that is currently unused.

`functors_used-B B` is the number of bytes of the functors section that is currently being used.

`functors_free-B B` is the number of bytes of the functors section that is currently unused.

`clause_store_gc_count-C C` is the number of times the clause store has been garbage collected.

`data_section_gc_count-C C` is the number of times the local and global stacks have been garbage collected.

Examples

```
| ?- statistics(Stats).
Stats = [wall_clock-1379264168244, global_stack_used-515952,
global-stack_free-83370112, local_stack_used-9088,
local_stack_free-67099776, trail_used-488,
trail_free-67108376, clause_store_used-61376,
clause_store_free-2035776, code_used-217456,
code_free-44688, constants_used-37008,
constants_free-225136, strings_used-19369,
strings_free-111703, functors_used-27168,
functors_free-103896, clause_store_gc_count-0,
data_section_gc_count-0] ?

% yes
```

Errors

None.

See also

None.

4.1.205 `stream/1`

Synopsis

```
stream(+T)
```

Description

Succeeds if `T` is either a stream term or a variable.

Examples

```
| ?- stream_alias(user_input, S), stream(S).
S = $stream(0) ?

% yes
| ?- stream(V).
V = _525152 ?

% yes
```

Errors

None.

See also

None.

4.1.206 `stream_alias/2`

Synopsis

```
stream_alias(+A, ?S)
```

Description

Succeeds if the atom `A` is an alias for the stream identified by the stream term `S`. That is to say, `alias(A)` is a property of `S`.

Examples

```
| ?- stream_alias(user_input, S).
S = $stream(0) ?

% yes
```

Errors

None.

See also

None.

4.1.207 stream_position_byte_count/2

Synopsis

```
stream_position_byte_count(+Position, ?Count).
```

Description

This predicate succeeds if `Position` is a stream position term and `Count` is the number of bytes read from or written to the stream.

Examples

See set_stream_position/2.

Errors

None.

See also

set_stream_position/2, stream_property/2.

4.1.208 stream_position_character_count/2

Synopsis

```
stream_position_character_count(+Position, ?Count).
```

Description

This predicate succeeds if `Position` is a stream position term and `Count` is the number of characters read from or written to the stream.

Examples

See set_stream_position/2.

Errors

None.

See also

set_stream_position/2, stream_property/2.

4.1.209 stream_position_line_count/2

Synopsis

```
stream_position_line_count(+Position, ?Count).
```

Description

This predicate succeeds if `Position` is a stream position term and `Count` is the number of lines written to or read from the stream. A newly opened stream has a line count of 1. Giving the first line a count of 1 makes more sense to humans than calling it line 0.

Examples

See set_stream_position/2.

Errors

None.

See also

set_stream_position/2, stream_property/2.

4.1.210 stream_position_line_position/2

Synopsis

```
stream_position_line_position(+Position, ?Count).
```

Description

This predicate succeeds if `Position` is a stream position term and `Count` is the offset from the beginning of the current line in characters.

Examples

See set_stream_position/2.

Errors

None.

See also

set_stream_position/2, stream_property/2.

4.1.211 stream_property/1

Synopsis

```
stream_property(+T)
```

Description

Succeeds if the argument `T` is a valid stream property term. Behaves as if it were defined as follows:

```
stream_property(file_name(_)).
stream_property(mode(_)).
stream_property(input).
stream_property(output).
stream_property(alias(_)).
stream_property(position(_)).
stream_property(end_of_stream(_)).
stream_property(eof_action(_)).
stream_property(reposition(_)).
stream_property(type(_)).
stream_property(character_encoding(_)).
```

Examples

```
| ?- stream_property(type(nonsense)).
% yes
```

Errors

None.

See also

stream_property/2.

4.1.212 stream_property/2

Synopsis

```
stream_property(+S, ?T)
```

Description

Succeeds if the stream `S` has property `T`. This is the list of valid properties:

`file_name(F)` The atom that was used to open the stream was `F`. See open/4.

`mode(M)` `M` is either `read`, `write`, or `append`.

`input` This is an input stream.

`output` This is an output stream.

`alias(A)` One of the aliases of the stream is `A`.

`position(P)` . The current position of the stream is `P`. See `set_stream_position/2`.

`end_of_stream(S)` Designates the stream position in terms of the end of the stream. `S` is one of

`at` The stream is currently at its end.

`past` The stream is currently past its end.

`not` The stream is neither at, nor past, its end.

`eof_action(A)` Designates what should happen should an program try to input from the stream when the current position is past the end. `A` is one of

`error` An exception is thrown.

`eof_code` An error code is returned.

`reset` The stream is reset so that it is no longer past the end of stream.

`reposition(B)` Designates whether or not the stream can be repositioned. `B` is either `true` or `false`.

`type(T)` Designates the type of the stream. `T` is either `binary` or `text`.

`character_encoding(E)` Designates how the stream characters are encoded — obviously meaningless for binary streams. `E` is one of

`ascii` This is an ASCII encoded stream.

`latin_1` This is an Latin-1 (ISO 8859-1) encoded stream.

`utf_8` This is an Unicode UTF-8 encoded stream.

`utf_16be` This is an Unicode UTF-16BE encoded stream.

`utf_16le` This is an Unicode UTF-16LE encoded stream.

`utf_32be` This is an Unicode UTF-32BE encoded stream.

`utf_32le` This is an Unicode UTF-32LE encoded stream.

Examples

```
| ?- stream_alias(user_input, S),
     findall(P, stream_property(S, P), Ps).
S = $stream(0)
P = _532144
Ps = [mode(read),input,alias(user_input),
      eof_action(reset),reposition(false),
      type(text),character_encoding(utf_8),
      file_name(user_input),
      position($stream_position(0,7,219,0)),
      end_of_stream(not)] ?

% yes
```

Errors

`domain_error(stream, S)` The argument `S` was not a valid stream term.

`domain_error(stream_property, T)` The argument `T` was not a valid stream property term.

See also

`open/4`, `stream_property/1`.

4.1.213 sub_atom/5

Synopsis

```
sub_atom(+Atom, ?Before, ?Length, ?After, ?SubAtom)
```

Description

Succeeds if the name of the atom `Atom` contains the name of the atom `SubAtom` of length `Length` characters which starts at index `Before` and ends at index `After`.

Examples

```
| ?- sub_atom(ab, Before, Length, After, SubAtom).
Before = 0
Length = 0
After = 2
SubAtom =  ? ;

Before = 0
```

```
Length = 1
After = 1
SubAtom = a ? ;

Before = 0
Length = 2
After = 0
SubAtom = ab ? ;

Before = 1
Length = 0
After = 1
SubAtom =  ? ;

Before = 1
Length = 1
After = 0
SubAtom = b ? ;

Before = 2
Length = 0
After = 0
SubAtom =  ? ;

% no
```

Errors

`instantiation_error` The argument `Atom` was not instantiated.

`domain_error(not_less_than_zero, After)` The argument `After` was an integer less than zero.

`domain_error(not_less_than_zero, Before)` The argument `Before` was an integer less than zero.

`domain_error(not_less_than_zero, Length)` The argument `Length` was an integer less than zero.

`type_error(atom, Atom)` The argument `Atom` was not an atom.

`type_error(atom, SubAtom)` The argument `SubAtom` was not an atom.

`type_error(integer, After)` The argument `After` was not an integer.

`type_error(integer, Before)` The argument `Before` was not an integer.

`type_error(integer, Length)` The argument `Length` was not an integer.

See also

atom_index/3, atom_length/2.

4.1.214 subsumes_chk/2

Synopsis

```
subsumes_chk(+T1, +T2)
```

Description

Same as subsumes_term/2.

Examples

See subsumes_term/2.

Errors

See subsumes_term/2.

See also

subsumes_term/2.

4.1.215 subsumes_term/2

Synopsis

```
subsumes_term(+T1, +T2)
```

Description

Succeeds if there is a substitution that when applied to `T1` unifies it with `T2`. The variables of `T1` are not bound.

Examples

```
| ?- subsumes_term(foo(A, B), foo(1,2)).
A = _524592
B = _525456 ?

% yes
```

Errors

None

See also

None.

4.1.216 `system_error/0`

Synopsis

```
system_error
```

Description

Exits the Prolog system with a system error code.

Examples

None.

Errors

None.

See also

None.

4.1.217 Term comparison

Synopsis

```
+L == +R
+L \== +R
+L @< +R
+L @=< +R
+L @> +R
+L @>= +R
```

Description

These predicates compare the terms `L` and `R` with respect to the term ordering. Term ordering is defined as follows:

- Variables precede floating-point numbers.
- Floating-point numbers precede integers.
- Integers precede atoms.
- Atoms precede compound terms.

- Two variables are ordered according to their memory addresses.
- Two numbers are ordered arithmetically.
- Two atoms are ordered according to the lexicographical ordering of their names. The null atom is the least atom.
- Two compound terms are ordered according to their arity. Two compound terms of the same arity are ordered according to the lexicographical ordering of their principle functor names. Two compound terms with the same arity and principle functor are ordered according to the term ordering of the first argument that they do not share.

With this ordering in mind we can say that a term is less than, equal to, or greater than another term. These relations are offered via the following binary predicates:

`L == R` `L` equal to `R`.

`L \== R` `L` not equal to `R`.

`L @< R` `L` less than `R`.

`L @=< R` `L` less than or equal to `R`.

`L @> R` `L` greater than `R`.

`L @>= R` `L` greater than or equal to `R`.

Examples

```
| ?- a @< b.
% yes
| ?- 3 == 3.0.
% no
| ?- 3 == 3.
% yes
```

Errors

Errors are not thrown by these predicates.

See also

`compare/3`, `setof/3`, `sort/2`.

4.1.218 term_expansion/2

Synopsis

```
term_expansion(+T1, -T2)
```

Description

Succeeds if there is a term expansion hook which succeeds with the same arguments. This predicate behaves as if it was defined as follows:

```
term_expansion(TermIn, TermOut) :-
        current_term_expansion(Functor),
        Goal =.. [Functor, TermIn, TermOut],
        call(Goal), !.
```

Examples

```
| ?- assert(inconsistent_expander(false, true)).
% yes
| ?- add_term_expansion(inconsistent_expander).
% yes
| ?- false.
% yes
```

Errors

None

See also

add_term_expansion/1, current_term_expansion/1, del_term_expansion/1.

4.1.219 throw/1

Synopsis

```
throw(+T)
```

Description

This predicate searches for the nearest exception handler installed by catch/3 which can handle the argument T. Control is then passed to this handler.

Examples

```
| ?- catch(throw(foo), E, format("Cought exception ~q~n", [E])).
Cought exception foo
E = foo ?

% yes
```

Errors

`instantiation_error` The argument `T` was not instantiated.

See also

`catch/3`.

4.1.220 `true/0`

Synopsis

```
true
```

Description

The predicate which always succeeds.

Examples

```
| ?- true.
% yes
```

Errors

None.

See also

`fail/0`.

4.1.221 `unget_byte/1`

Synopsis

```
unget_byte(+B)
```

Description

Behaves as if it were defined as follows:

```
unget_byte(Byte) :-
        current_input(S),
        unget_byte(S, Byte).
```

Examples

See unget_byte/2.

Errors

See unget_byte/2.

See also

current_input/1, unget_byte/2.

4.1.222 unget_byte/2

Synopsis

```
unget_byte(+S, +B)
```

Description

Places a byte `B` into the binary stream `S` so that it will be the next byte read from that stream.

Examples

```
test_unget_byte :-
        open('test_file.bin', read, Stream, [type(binary)]),
        unget_byte(Stream, 16'11),
        get_byte(Stream, 16'11),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` The argument `S` was not instantiated.

`permission_error(input, stream, S)` There is a lack of permission to read from stream `S`.

`permission_error(input, text_stream, S)` The argument `S` refers to a text stream.

`representation_error(byte)` The argument `B` was instantiated but not a valid byte.

`type_error(integer, B)` The argument `B` was instantiated but not a valid integer.

See also

`byte/1`, `integer/1`.

4.1.223 unget_char/1

Synopsis

```
unget_char(+C)
```

Description

Behaves as if it were defined as follows:

```
unget_char(Char) :-
        current_input(S),
        unget_char(S, Char).
```

Examples

See `unget_char/2`.

Errors

See `unget_char/2`.

See also

`current_input/1`, `unget_char/2`.

4.1.224 unget_char/2

Synopsis

```
unget_char(+S, +C)
```

Description

Places a character `C` into the text stream `S` so that it will be the next character read from that stream.

Examples

```
test_unget_char :-
        open('test_file.txt', read, Stream),
        unget_char(Stream, d),
        get_char(Stream, d),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` One of the arguments, `S` or `C`, was not instantiated.

`permission_error(input, stream, S)` There is a lack of permission to read from stream `S`.

`permission_error(input, binary_stream, S)` The argument `S` refers to a binary stream.

`type_error(character, C)` The argument `C` was instantiated but not a valid character.

See also

`character/1`.

4.1.225 `unget_code/1`

Synopsis

```
unget_code(+C)
```

Description

Behaves as if it were defined as follows:

```
unget_code(Code) :-
        current_input(S),
        unget_code(S, Code).
```

Examples

See unget_code/2.

Errors

See unget_code/2.

See also

current_input/1, unget_code/2.

4.1.226 unget_code/2

Synopsis

```
unget_code(+S, +C)
```

Description

Places a character code `C` into the text stream `S` so that it will be the next character code read from that stream.

Examples

```
test_unget_code :-
        open('test_file.txt', read, Stream),
        unget_code(Stream, 64),
        get_code(Stream, 64),
        close(Stream).
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a valid stream term or stream alias.

`existence_error(stream, S)` The stream `S` does not exist.

`instantiation_error` One of the arguments, `S` or `C`, was not instantiated.

`permission_error(input, stream, S)` There is a lack of permission to read from stream `S`.

`permission_error(input, binary_stream, S)` The argument `S` refers to a binary stream.

`representation_error(character_code, C)` The argument `C` was instantiated but not a character code.

`type_error(integer, C)` The argument `C` was instantiated but not an integer.

See also

`character_code/1`.

4.1.227 `'\='/2`

Synopsis

```
+T1 \= +T2
```

Description

Succeeds if the term `T1` cannot be unified with the term `T2`. Behaves as if defined as follows:

```
'\\='(T1, T2) :- T1=T2, !, fail.
'\\='(_, _).
```

Examples

```
| ?- 9 \= 19.
% yes
| ?- X \= x.
% no
```

Errors

None.

See also

`'='/2`.

4.1.228 `'='/2`

Synopsis

```
+T1 = +T2
```

Description

Succeeds if the term `T1` can be unified with the term `T2`. This predicate does not perform the occurs check, i.e., `X=f(X)` succeeds. Behaves as if defined as follows:

```
'='(T,T).
```

Examples

```
| ?- X = x.
X = x ?

% yes
| ?- 9 = 19.
% no
```

Errors

None.

See also

'\='/2, unify_with_occurs_check/2.

4.1.229 unify_with_occurs_check/2

Synopsis

```
unify_with_occurs_check(+T1, +T2)
```

Description

Succeeds if the term `T1` can be unified with the term `T2`. This predicate performs the occurs check, i.e., `X=f(X)` fails.

Examples

```
| ?- unify_with_occurs_check(x, X).
X = x ?

% yes
| ?- unify_with_occurs_check(X, f(X)).
% no
```

Errors

None.

See also

'='/2.

4.1.230 '=..'/2

Synopsis

```
+T1 =.. +T2
```

Description

Succeeds if the term `T1` can be constructed from the elements of the list `T2` where the head of `T2` is the principle functor of `T1` and the remaining elements of `T2` are the arguments of `T1` in the correct order. This predicate considers atomic terms to be compounds of arity zero. Floating-point numbers are processed as the compound terms that represent them.

Examples

```
| ?- foo(1,2,3) =.. L.
L = [foo,1,2,3] ?

% yes
| ?- T =.. [a,b].
T = a(b) ?

% yes
| ?- 99 =.. L.
L = [99] ?

% yes
| ?- foo =.. L.
L = [foo] ?

% yes
| ?- 3.14 =.. L.
L = [$float,1448069409786199 8019,2,64] ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `T2` was the atom `[]`.

`instantiation_error` Either both `T1` and `T2` are not instantiated, or, both `T1` and the head of the list `T2` are not instantiated.

`representation_error(max_arity)` The length of the list `L2` was too long.

`type_error(atom, H)` The head of the list `T2` is `H` which is not an atom and `T2` contains more than one element.

`type_error(atomic, H)` The argument `T2` is a single element list `[H]` and `H` is not atomic.

`type_error(list, L)` The argument `T2` was not a proper list. The problematic sub-list is `L`.

See also

functor/3.

4.1.231 var/1

Synopsis

```
var(+T)
```

Description

Succeeds if `T` is an uninstantiated variable.

Examples

```
| ?- var(99).
% no
| ?- var(X).
X = _524480 ?

% yes
| ?- X = 99, var(X).
% no
```

Errors

None.

See also

nonvar/1.

4.1.232 well_formed_body_term/1

Synopsis

```
well_formed_body_term(+T)
```

Description

Succeeds if `T` is a well formed body term. This collection of well formed body terms is the smallest one satisfying:

(i) all variables are well formed body terms,

(ii) all predications where the principle functor is not one of `';'/2`, `','/2`, or `'->'/2`, are well formed body terms,

(iii) if `T1` and `T2` are well formed body terms then so are

- the conjunction of `T1` and `T2`, `(T1 , T2)`, and
- the disjunction of `T1` and `T2`, `(T1 ; T2)`, and
- the implication of `T1` and `T2`, `(T1 -> T2)`.

Examples

```
| ?- well_formed_body_term(X).
X = _525344 ?

% yes
| ?- well_formed_body_term((a,b)).
% yes
| ?- well_formed_body_term((a;b)).
% yes
| ?- well_formed_body_term((a->b)).
% yes
| ?- well_formed_body_term(99).
% no
```

Errors

None.

See also

predication/1, var/1.

4.1.233 write/1

Synopsis

```
write(+T)
```

Description

Behaves as if it were defined as follows:

```
write(Term) :-
        current_output(S),
        write(S, Term).
```

Examples

See write/2.

Errors

See write/2.

See also

current_output/1, write/2.

4.1.234 write/2

Synopsis

```
write(+S, +T)
```

Description

Behaves as if it were defined as follows:

```
write(S, Term) :-
        write_term(S, Term, [numbervars(true)]).
```

Examples

See write_term/2.

Errors

See write_term/2.

See also

write_term/2.

4.1.235 writeq/1

Synopsis

```
writeq(+T)
```

Description

Behaves as if it were defined as follows:

```
writeq(Term) :-
        current_output(S),
        writeq(S, Term).
```

Examples

See writeq/2.

Errors

See writeq/2.

See also

current_output/1, writeq/2.

4.1.236 writeq/2

Synopsis

```
writeq(+S, +T)
```

Description

Behaves as if it were defined as follows:

```
writeq(S, Term) :-
        write_term(S, Term, [quoted(true), numbervars(true)]).
```

Examples

See write_term/2.

Errors

See write_term/2.

See also

write_term/2.

4.1.237 write_canonical/1

Synopsis

```
write_canonical(+T)
```

Description

Behaves as if it were defined as follows:

```
write_canonical(Term) :-
        current_output(S),
        write_canonical(S, Term).
```

Examples

See write_canonical/2.

Errors

See write_canonical/2.

See also

current_output/1, write_canonical/2.

4.1.238 write_canonical/2

Synopsis

```
write_canonical(+S, +T)
```

Description

Behaves as if it were defined as follows:

```
write_canonical(S, Term) :-
        write_term(S, Term, [quoted(true), ignore_ops(true)]).
```

Examples

See write_term/2.

Errors

See write_term/2.

See also

write_term/2.

4.1.239 write_term/2

Synopsis

```
write_term(+T, +O)
```

Description

Behaves as if it were defined as follows:

```
write_term(T, O) :-
        current_output(S),
        write_term(S, T, O).
```

Examples

See write_term/3.

Errors

See write_term/3.

See also

current_output/1, write_term/3.

4.1.240 write_term/3

Synopsis

```
write_term(+S, +T, +O)
```

Description

Writes the term T to the stream S in accordance with the elements of the list of options O. The list of valid options is:

ignore_ops(true) All prefix, infix, and postfix operators are written in so called functional notation whereby the principle functor precedes a list of arguments which is contained in brackets. Lists are written in terms of '.'/2.

ignore_ops(false) Any prefix, infix, or postfix operator (see current_op/3) is written in accordance with its fixity and precedence.

numbervars(true) All terms of the form '$VAR'(N) where N is a number are output as variables. See numbervars/3.

numbervars(false) All terms of the form '$VAR'/1 are not given special treatment.

portray(true) Portray hooks can override the writing of terms. See add_portray/1.

portray(false) No portray hooks are called.

`quoted(true)` All atoms are written using their names but any names that could not be recognised by `read_term/3` will be quoted.

`quoted(false)` All atoms are written using their names, there is not automatic quoting.

If two conflicting options are given in the list — that is two options with the same principle functor — then the last of the two takes precedence. The default set of options is `[ignore_ops(false), numbervars(false), portray(false), quoted(false)]`. Any option passed in the argument `O` takes precedence over a conflicting default.

Examples

```
| ?- write_term(['a b', 1+2, '$VAR'(26)], []), nl.
[a b,1+2,$VAR(26)]
% yes
| ?- write_term(['a b', 1+2, '$VAR'(26)],
                [ignore_ops(true),
                 numbervars(true),
                 quoted(true)]),
     nl.
.('a b',.(+(1,2),.(A1,[])))
% yes
```

Errors

`domain_error(stream_or_alias, S)` The argument `S` is not a stream term or a stream alias.

`domain_error(write_option, Option)` One of the elements of the argument `O`, `Option`, was not a valid write option.

`existence_error(stream, S)` , Either the argument `S` does not refer to a currently open stream, or, `S` is not a current stream alias.

`instantiation_error` Either one of the arguments `S` or `O` is not instantiated, or, an element of `O` was not ground.

`permission_error(output, binary_stream, S)` The argument `S` refers to a binary stream.

`permission_error(output, stream, S)` The argument `S` refers to an input stream.

`type_error(list, O)` The argument `O` is not a list.

See also

add_portray/1, current_op/3, numbervars/3, read_term/3.

4.1.241 version/0

Synopsis

```
version
```

Description

Uses print_message/2 to display version information for Barry's Prolog and the Barry's Prolog Abstract Machine.

Examples

```
| ?- version.
Abstract Machine version: P1A01.
Prolog version: P1A01.
% yes
```

Errors

None.

See also

None.

4.2 Definite Clause Grammar

A Definite Clause Grammar (DCG) is a formalism similar to the well known Context Free Grammar (CFG) [DM93]. Just like a CFG, a DCG describes a language using rules, terminals, and non-terminals. However, where the right-hand side of a CFG rule is defined as a sequence of terminals and non-terminals, a DCG's right-hand side can, in addition, contain Prolog queries and certain extra-grammatical operators such as '!'/0 or '->'/2 which behave in the same way in grammar rules as their extra-logical namesakes do in Prolog goals. Also, the left-hand side of a CFG rule is constrained to be a single non-terminal, but a DCG rule can have an optional list of terminals following the non-terminal on its left-hand side.

When input as a read term, a DCG rule is automatically expanded into a Prolog clause by expand_term/2. This gives the programmer a convenient language to define language acceptors and generators.

4.2.1 Motivation

Consider the following clause, `det/2`, which succeeds if its first argument is a list which begins with the symbol `the` — an English language determiner — and the second argument is the first argument minus this determiner prefix.

```
det([the|Rest], Rest).
```

Consider also these clauses which deal with some English nouns in the same way.

```
noun([cat|Rest], Rest).
noun([dog|Rest], Rest).
```

If we take the conjunction of `det/2` and `noun/2` and feed the second argument of `det/2` into the first argument of `noun/2`, then we would have the body of a clause to recognise or generate simple English noun phrases:

```
noun_phrase(Input0, Rest) :-
        det(Input0, Input1),
        noun(Input1, Rest).
```

Here's an example of what we've done:

```
| ?- noun_phrase(A, B).
A = [the,cat|_524032]
B = _524032 ? ;

A = [the,dog|_524032]
B = _524032 ? ;

% no
```

Here we do something similar for verbs and verb phrases:

```
verb([chases|Rest], Rest).
verb_phrase(Input0, Rest) :-
        verb(Input0, Input1),
        noun_phrase(Input1, Rest).
```

Building upon what we've done so far, We can easily recognise or generate sentences:

```
sentence(Input) :-
        noun_phrase(Input, Rest),
        verb_phrase(Rest, []).
```

Example:

```
| ?- sentence([the, cat, chases, the, dog]).
% yes
```

We can generate or recognise the same simple English language subset using a DCG like this:

```
dcg_det --> [the].
dcg_noun --> [cat].
dcg_noun --> [dog].
dcg_verb --> [chases].
dcg_noun_phrase --> dcg_det, dcg_noun.
dcg_verb_phrase --> dcg_verb, dcg_noun_phrase.
dcg_sentence --> dcg_noun_phrase, dcg_verb_phrase.
```

The language defined is:

```
| ?- dcg_sentence(S, []).
S = [the,cat,chases,the,cat] ? ;

S = [the,cat,chases,the,dog] ? ;

S = [the,dog,chases,the,cat] ? ;

S = [the,dog,chases,the,dog] ? ;

% no
```

As can be seen, specifying a language with a DCG is much simpler than writing Prolog code by hand and, quite surprisingly, there is very little if any loss in performance. In fact the Prolog code created for the DCG grammar rules is *very* similar to the hand crafted examples. Compare the code for noun phrases that we wrote by hand with that generated by `expand_term/2`:

```
| ?- listing([noun_phrase/2, dcg_noun_phrase/2]).
noun_phrase(A,B) :-
    det(A,C),
    noun(C,B).
dcg_noun_phrase(A,B) :-
    dcg_det(A,C),
    dcg_noun(C,B).
% yes
```

4.2.2 DCG Grammar

This section describes the syntax of a DCG rule in terms of DCG rules. The consequence of this is that if you want to understand the DCG formalism from this description, you will have to either

- already understand the DCG formalism, or
- translate the following grammar into another formalism, e.g., Backus-Naur Form by replacing `'-->'` with `'::='`, and `['X']` with `X`.

The grammar of the DCG rule is as follows:

```
dcg_rule --> dcg_nonterminal, [','], ['['], dcg_terminals,
             ['-->'], dcg_body.
dcg_rule --> dcg_nonterminal, ['-->'], dcg_body.

dcg_body --> dcg_body_disjunction.

dcg_body_disjunction --> dcg_body_conditional, [';'], dcg_body.
dcg_body_disjunction --> dcg_body_conditional, ['|'], dcg_body.
dcg_body_disjunction --> dcg_body_conjunction, [';'], dcg_body.
dcg_body_disjunction --> dcg_body_conjunction, ['|'], dcg_body.
dcg_body_disjunction --> dcg_body_conjunction.

dcg_body_conditional --> dcg_body_conjunction,
                         ['->'], dcg_body_conjunction.

dcg_body_conjunction --> dcg_body_not, [','],
                         dcg_body_conjunction.
dcg_body_conjunction --> dcg_body_not.

dcg_body_not --> ['\+'], dcg_body_not.
dcg_body_not --> dcg_body_terminal.

dcg_body_terminal --> [V], {var(V)}.
dcg_body_terminal --> [phrase(T)].
dcg_body_terminal --> [call(T)].
dcg_body_terminal --> [{T}].
dcg_body_terminal --> ['[]'].
dcg_body_terminal --> ['['], dcg_terminals.
dcg_body_terminal --> ['!'].
dcg_body_terminal --> ['('], dcg_body, [')'].
dcg_body_terminal --> dcg_nonterminal.

dcg_nonterminal --> [Term], {callable_term(Term)}.

dcg_terminals --> [Term], [']'].
dcg_terminals --> [Term], [','], dcg_terminals.
```

4.2.3 DCG Expansion

We can best see what the expansion process does by looking at the output of `expand_term/2` for each DCG grammar non-terminal.

Rules

Rules are the starting point.

```
dcg_rule --> dcg_nonterminal, [','], ['['], dcg_terminals,
             ['-->'], dcg_body.
dcg_rule --> dcg_nonterminal, ['-->'], dcg_body.
```

The head of the clause which is generated for this rule is built from the non-terminal on the left of the arrow. This rule can be referenced on the right-hand side of rules by using this non-terminal.

```
| ?- expand_term((a --> b), T).
T = a(_542064,_542032) :- b(_542064,_542032) ?

% yes
```

The terminals which can follow the non-terminal on the left-hand side of the arrow allow you to constrain the second list argument.

```
| ?- expand_term((a, [b] --> c), T).
T = a(_543408,_543376) :- c(_543408,_544608),_543376 = [b|_544608] ?

% yes
| ?- expand_term((a, [b,c] --> d), T).
T = a(_544032,_544000) :- d(_544032,_545232),_544000 = [b,c|_545232] ?

% yes
| ?- expand_term((a, [b] --> []), T), assert(T).
T = a(_546704,_546672) :- _546704 = _547808,_546672 = [b|_547808] ?

| ?- a(A, B).
A = _520960
B = [b|_520960] ?

% yes
```

Disjunctions

A body can contain a disjunction which is essentially a grammatical alternative.

```
dcg_body --> dcg_body_disjunction.
dcg_body_disjunction --> dcg_body_conditional, [';'], dcg_body.
dcg_body_disjunction --> dcg_body_conditional, ['|'], dcg_body.
dcg_body_disjunction --> dcg_body_conjunction, [';'], dcg_body.
dcg_body_disjunction --> dcg_body_conjunction, ['|'], dcg_body.
dcg_body_disjunction --> dcg_body_conjunction.
```

These five disjunction rules are, in fact, only three rules because the first two rules are exactly the same, and the the third and fourth rules are exactly the same. The reason for this is that the symbol `'|'` is never a functor returned by `read_term/3`, it is only treated as a synonym for `';'` which can be seen from this example:

```
| ?- read(X).
|: a | b.
X = a ; b ?

% yes
```

The expansion is straightforward:

```
| ?- expand_term((a --> b ; c), T).
T = a(_554640,_554608) :- b(_554640,_554608) ; c(_554640,_554608) ?

% yes
```

Conditionals

The grammar operator `'->'/2` behaves in the same way as the Prolog predicate with the same name but it must be the topmost grammar operator on the left hand side of a `';'/2` — you can't have a `'->'/2` without a `';'/2`. The grammar rule for the conditional forces `'->'/2` to be non-associative.

```
dcg_body_conditional --> dcg_body_conjunction,
                         ['->'], dcg_body_conjunction.
```

Conjunctions

As in a Prolog clause, `','/2` denotes conjunction which is equivalent to a grammatical concatenation. The second conjunction rule shows that this is optional.

```
dcg_body_conjunction --> dcg_body_not, [','],
                         dcg_body_conjunction.
dcg_body_conjunction --> dcg_body_not.
```

Negations

The optional grammar operator `'\+'/1` behaves as its namesake does in Prolog.

```
dcg_body_not --> ['\+'], dcg_body_not.
dcg_body_not --> dcg_body_terminal.
```

It is translated verbatim

```
| ?- expand_term((a --> \+ b), T).
T = a(_553552,_553520) :- \+ b(_553552,_554784),_553552 = _553520 ?

% yes
```

Terminals

Note: we're confusing the term "terminals" — which usually means symbols from some alphabet — with atomic DCG terms. We'll call the symbols "actual terminals".

The terminals are, perhaps unsurprisingly, described by the largest grammatical category. A terminal can be a single variable — the operator `'{}'/1` is described later.

```
dcg_body_terminal --> [V], {var(V)}.
```

But as we see from the expansion, the variable will have to be instantiated before the call to phrase/3 or an exception will be thrown!

```
| ?- expand_term((a --> B), T).
B = _524768
T = a(_544064,_544032) :- phrase(_524768,_544064,_544032) ?

% yes
```

Using the grammar operator `phrase/1` we can inline a call to the predicate phrase/3.

```
dcg_body_terminal --> [phrase(T)].
```

From the example that follows, we see that `phrase/1` expands into phrase/3 where the extra two arguments are the representation of the list being processed.

```
| ?- expand_term((a --> phrase(b)), T).
T = a(_544544,_544512) :- phrase(b,_544544,_544512) ?

% yes
```

The two extra arguments that represent the list being processed can passed on to Prolog predicates with the grammar operator `call/1`.

```
dcg_body_terminal --> [call(T)].
```

The expansion adds the two extra arguments to build a call of `call/3`.

```
| ?- expand_term((a --> call(c)), T).
T = a(_544096,_544064) :- call(c,_544096,_544064) ?

% yes
```

Ordinary Prolog goals are introduced with the grammar operator `'{}'/1`. This operator's argument is a Prolog goal.

```
dcg_body_terminal --> [{T}].
```

The goal is just copied verbatim into the expanded term.

```
| ?- expand_term((a --> {b}), T).
T = a(_543008,_542976) :- b,_543008 = _542976 ?

% yes
```

The empty sequence is denoted by `[]`.

```
dcg_body_terminal --> ['[]'].
```

It's expansion is a simple unification.

```
| ?- expand_term((a --> []), T).
T = a(_542080,_542048) :- _542080 = _542048 ?

% yes
```

Note that these two read terms are not the same:

```
a.
a --> [].
```

According to [O'K90] confusing the two is a common mistake.

Actual terminals are Prolog terms in a list.

```
dcg_body_terminal --> ['['], dcg_terminals.
```

Expansion of actual terminals constrains the two list arguments, i.e., the list prefix is specified.

```
| ?- expand_term((a --> [b,c]), T).
T = a(_543216,_543184) :- _543216 = [b,c|_543184] ?

% yes
```

Non-determinism can be controlled with the grammar operator `'!'/0` which behaves just as the predicate with the same name does in Prolog goals.

```
dcg_body_terminal --> ['!'].
```

The expansion is straightforward and comparing the following two examples we can see the DCG cut really is the Prolog cut.

```
| ?- expand_term((a --> !, b), T).
T = a(_542800,_542768) :- (!,_542800 = _544032),b(_544032,_542768) ?

% yes
| ?- expand_term((a --> {!}, b), T).
T = a(_543248,_543216) :- (!,_543248 = _544480),b(_544480,_543216) ?

% yes
```

The DCG grammar is designed to give the grammar operators relative priorities in line with the Prolog operator table. Parentheses can be used to override this.

```
dcg_body_terminal --> ['('], dcg_body, [')'].
```

Examples:

```
| ?- expand_term((a --> (b)), T).
T = a(_542496,_542464) :- b(_542496,_542464) ?

% yes
| ?- expand_term((a --> b, (c ; d), e), T).
T = a(_556800,_556768) :- b(_556800,_558064),
                          (c(_558064,_559440) ; d(_558064,_559440)),
                          e(_559440,_556768) ?

% yes
```

A rule (a) is referenced by another rule (b) when the non-terminal on the left-hand side of (a) is used in the right hand side of (b).

```
dcg_body_terminal --> dcg_nonterminal.
```

A non-terminal on the right-hand side is expanded into the appropriate call.

```
| ?- expand_term((a --> a), T).
T = a(_542064,_542032) :- a(_542064,_542032) ?

% yes
```

Terminal Lists

As shown above actual terminals are provided in a list.

```
dcg_terminals --> [Term], [']'].
dcg_terminals --> [Term], [','], dcg_terminals.
```

The following two rules are equivalent:

```
rule_a --> [one, two, three].
rule_b --> [one], [two], [three].
```

Non-terminals

As we can see from its grammar rule, a non-terminal is a callable term:

```
dcg_nonterminal --> [Term], {callable_term(Term)}.
```

This callable term is the basis of the procedure name for this rule. Each term has two arguments added to it and these are equivalent to the two list arguments that our hand crafted Prolog clauses had in the examples given earlier.

```
| ?- expand_term((a --> b), T).
T = a(_542064,_542032) :- b(_542064,_542032) ?

% yes
| ?- expand_term((a(1,2) --> b), T).
T = a(1,2,_545408,_545376) :- b(_545408,_545376) ?

% yes
```

4.3 Flags

4.3.1 `bounded`

Description

This flag is specified by both the ISO Prolog and the ISO/IEC 10967 standards to show whether or not bounded integer arithmetic is implemented. The flag value of `false` indicates that we have arbitrary precision integers, that is to say, the maximum and minimum integers are not bounded by some predefined constants.

Default Value

`false`

Possible Values

This is a constant flag and cannot be changed from the default.

4.3.2 char_conversion

Description

A switch which enables or disables the conversion of characters in unquoted terms read by read_term/3. Characters are converted according to the character conversion table which is described by current_char_conversion/2.

Default Value

on

Possible Values

on Unquoted terms are subject to character conversion.

off Unquoted terms are not subject to character conversion.

4.3.3 char_escapes

Description

A switch which controls the behaviour of read_term/3 when it encounters the backslash character (ASCII 92, '\') in a symbol. When enabled, the next character is interpreted as an escape character.

Default Value

on

Possible Values

on Enable the interpretation of '\' as an escape character.

off Disable the interpretation of '\' as an escape character.

4.3.4 collapse_multiple_minuses

Description

A switch which controls the behaviour of read_term/3 when it encounters any atoms which have names which comprise only of three or more minus (-) characters. When set to on, any such atom is reported as ---.

Default Value

`off`

Possible Values

`on` Any atom whose name comprises only of three or more minus characters is read as `---`.

`off` No special handling of atom names is performed.

4.3.5 `discontiguous_clauses_warnings`

Description

A switch which controls the behaviour of `consult/1`, `reconsult/1`, and `compile_file/2` when it encounters any procedure where the clauses are not all adjacent to one another.

Default Value

`on`

Possible Values

`on` Print a warning message when a discontiguous clause is detected.

`off` No warnings are printed.

4.3.6 `double_quotes`

Description

The value of this flag determines what happens when `read_term/3` encounters a double quoted sequence of characters.

Default Value

`codes`

Possible Values

`atom` The sequence of characters is to be interpreted as an atom name.

`chars` The sequence of characters is to be interpreted as a list of one character atoms.

`codes` The sequence of characters is to be interpreted as a list of character codes.

4.3.7 floating_point_output_format

Description

The value of this flag determines the format used to write floating-point numbers with write_term/3.

Default Value

decimal

Possible Values

decimal The text written comprises of a integral part and a fractional part separated by a radix point. Example: 12.34. Here *decimal* does not mean base 10 representation it just means "not scientific format". See the flag number_output_base for the base representation.

scientific The text written comprises of a single digit integral part, a radix point, a fractional part, the character e, then an exponent part. Example: 1.23e4

4.3.8 floating_point_output_precision

Description

The value of this flag is the number of base digits after the radix point that are written by write_term/3. A value of N means generate all digits. A value of -N means generate N digits.

Default Value

-6

Possible Values

Any integer less than or equal to 0.

4.3.9 floating_point_precision

Description

The value of this flag is the number of bits of precision used for the fraction part of a floating-point number. See eval/2.

Default Value

64

Possible Values

Any integer greater than `0`.

4.3.10 `integer_rounding_function`

Description

This is a flag specified by the ISO Prolog standard to show the direction in which integer division rounds.

Default Value

`toward_zero`

Possible Values

This is a constant flag and cannot be changed from the default.

4.3.11 `max_arity`

Description

The value of this flag is the maximum arity of any predicate or compound term.

Default Value

`255`

Possible Values

This is a constant flag and cannot be changed from the default.

4.3.12 `modulo`

Description

This is a flag specified by the ISO/IEC 10967 standard to be `false` since the flag `bounded` is `false`.

Default Value

`false`

Possible Values

This is a constant flag and cannot be changed from the default.

4.3.13 number_output_base

Description

This flag determines the base which numbers are to be written with write_term/3.

Default Value

10

Possible Values

Any integer greater than or equal to 2, and less than or equal to 36.

4.3.14 singleton_var_warnings

Description

A switch which controls the behaviour of consult/1, reconsult/1, and compile_file/2 when it encounters any singleton variables found in a term read as input.

Default Value

on

Possible Values

on Print a warning message with a list of singleton variables for each term read.

off No warnings are printed.

4.3.15 unknown

Description

The value of this flag determines the action taken when an unknown procedure is called.

Default Value

error

Possible Values

`error` The exception `existence_error(predicate, F/N)` is thrown where `F/N` is the predicate indicator of the unknown procedure.

`warning` A warning is printed on the stream `user_error`, the call then fails.

`fail` The call silently fails.

Chapter 5

The assoc library

The predicates in this library implement an association list data type using binary trees. Each element of an association list is a pair: a key and a value. The key is used to search for the associated value. To use the predicates in this library, you will need to load the `assoc.fasl` file. One way to do this is with the following call: `ensure_loaded(runtime(assoc))`.

An empty association list is the empty binary tree and this is represented by the atom `t`. In the case of a non-empty association list, each element of the list is represented by one node of the binary tree. The binary tree node structure has the form `t(Key, Value, LeftTree, RightTree)`. For any key `K` found in `LeftTree` we know that `K @< Key`. For any key `K` found in `RightTree` we know that `K @> Key`. There is no other key `K` in `LeftTree` or `RightTree` such that `K == Key`.

5.1 Predicates

5.1.1 assoc_to_list/2

Synopsis

```
assoc_to_list(+Assoc, -List)
assoc_to_list(-Assoc, +List)
```

Description

Succeeds if the list `List` contains all of the `Key` and `Value` pairs in the association list `Assoc`.

Examples

```
| ?- put_assoc(b, t, b, Assoc1),
     put_assoc(a, Assoc1, a, Assoc2),
     put_assoc(c, Assoc2, c, Assoc3),
```

```
     assoc_to_list(Assoc3, Lst).
Assoc1 = t(b,b,t,t)
Assoc2 = t(b,b,t(a,a,t,t),t)
Assoc3 = t(b,b,t(a,a,t,t),t(c,c,t,t))
Lst = [a-a,b-b,c-c] ?

% yes
| ?- assoc_to_list(Assoc, [a-a, b-b, c-c]).
Assoc = t(b,b,t(a,a,t,t),t(c,c,t,t)) ?

% yes
```

Errors

None.

See also

None.

5.1.2 get_assoc/3

Synopsis

```
get_assoc(+Key, +Assoc, -Value)
get_assoc(-Key, +Assoc, +Value)
get_assoc(-Key, +Assoc, -Value)
```

Description

Succeeds if the association list `Assoc` associates `Key` with `Value`.

Examples

```
| ?- put_assoc(key0, t, value0, Assoc0),
     put_assoc(key1, Assoc0, value1, Assoc1),
     get_assoc(key0, Assoc1, Value0),
     get_assoc(Key1, Assoc1, value1),
     findall(Key-Value,
             get_assoc(Key, Assoc1, Value),
             Associations).
Assoc0 = t(key0,value0,t,t)
Assoc1 = t(key0,value0,t,t(key1,value1,t,t))
Value0 = value0
Key1 = key1
Key = _573344
```

```
Value = _574336
Associations = [key0-value0,key1-value1] ?

% yes
```

Errors

None.

See also

put_assoc/4.

5.1.3 map_assoc/3

Synopsis

```
map_assoc(+Pred, +OldAssoc, -NewAssoc)
```

Description

For each element (`Key`, `Value`) of `OldAssoc`, there is an element (`Key`, `NewValue`) of `NewAssoc` such that the call `Pred(Value, NewValue)` succeeds. The two association lists are structurally identical with the exception of the values.

Examples

```
| ?- assert(map_assoc_pred(a, 1)).
% yes
| ?- assert(map_assoc_pred(b, 2)).
% yes
| ?- assert(map_assoc_pred(c, 3)).
% yes
| ?- put_assoc(b, t, b, Assoc2),
     put_assoc(a, Assoc2, a, Assoc3),
     put_assoc(c, Assoc3, c, Assoc4),
     map_assoc(map_assoc_pred, Assoc4, Assoc5),
     get_assoc(a, Assoc5, 1),
     get_assoc(b, Assoc5, 2),
     get_assoc(c, Assoc5, 3).
Assoc2 = t(b,b,t,t)
Assoc3 = t(b,b,t(a,a,t,t),t)
Assoc4 = t(b,b,t(a,a,t,t),t(c,c,t,t))
Assoc5 = t(b,2,t(a,1,t,t),t(c,3,t,t)) ?

% yes
```

Errors

None.

See also

None.

5.1.4 put_assoc/4

Synopsis

```
put_assoc(+Key, +OldAssoc, +Value, -NewAssoc)
```

Description

Succeeds if the association list `NewAssoc` is the result of inserting the pair (`Key`, `Value`) into the association list `OldAssoc`. Should `Key` already be associated with a value in `OldAssoc`, the insertion operation updates the existing association.

Examples

```
| ?- put_assoc(key, t, value, Assoc0),
     get_assoc(key, Assoc0, Value0),
     put_assoc(key, Assoc0, new_value, Assoc1),
     get_assoc(key, Assoc1, Value1).
Assoc0 = t(key,value,t,t)
Value0 = value
Assoc1 = t(key,new_value,t,t)
Value1 = new_value ?

% yes
```

Errors

None.

See also

get_assoc/3.

Chapter 6

The bup library

To use the predicates in this library, you will need to load the `bup.fasl` file. One way to do this is with the following call: `ensure_loaded(runtime(bup))`.

This library implements a parser generator as described in the paper [MTH+83]. The main differences between parsers generated by BUP and those generated by a DCG are:

- BUP creates a bottom-up parser which can process a left-recursive grammar. A DCG parser is of the top-down type and cannot process these grammars.
- BUP can split input processing into grammar rule application and dictionary look-up. This permits the independent processing of grammar terminals. For example, a natural language parser may make use of a dictionary too large to store in the grammar itself, so the system could store the dictionary in a file and search this file for words when needed.
- BUP can be configured to create a parser that automatically caches input sequences and the corresponding parse information. This cache eliminates recomputation when the input sequence is seen again.
- Unlike a DCG grammar, a BUP grammar cannot handle empty productions at this time.

It is an error for a BUP grammar to contain a cycle.

To create a parser, the BUP compiler (`bup_compile/2`) is given the name of a file containing a grammar and the name of a file where the Prolog clauses which implement the corresponding parser are to be written. The user can then consult the Prolog clauses and use the `bup_goal/4` predicate to parse.

The input to the compiler is in the form of Prolog terms with principle functor `'::='/2`. The format is the same as that of a DCG with the exception that `'-->'/2` is replaced by `'::='/2`. Here is an example BUP grammar:

```
s(s(NP,VP)) ::= np(NP), vp(VP), ['.'].

np(np(D,N)) ::= det(D), noun(N).

vp(vp(V,NP)) ::= verb(V), np(NP).

det(d(the)) ::= [the].
det(d(a)) ::= [a].

noun(n(cat)) ::= [cat].
noun(n(dog)) ::= [dog].
noun(n(mouse)) ::= [mouse].

verb(v(chases)) ::= [chases].
verb(v(scares)) ::= [scares].
```

If the above grammar was stored in the file `'grammar.bup'`, we could compile, load and use the grammar with the following:

```
| ?- ensure_loaded(runtime(bup)).
% yes
| ?- bup_compile('grammar.bup', 'grammar.pl').
BUP compilation
input file  : grammar.bup
output file : grammar.pl
Finished
% yes
| ?- ['grammar.pl'].
% yes
| ?- ensure_loaded(runtime(readin)).
% yes
| ?- read_in(S), bup_goal(s, [Tree], S, []).
|: The cat chases the mouse.
S = [the,cat,chases,the,mouse,.]
Tree = s(np(d(the),n(cat)),vp(v(chases),np(d(the),n(mouse)))) ?

% yes
```

In the above example we've used `bup_compile/2` which gives us the default options for BUP parser generation. There is a BUP compile predicate, `bup_compile/3`, which takes a list of options as its third argument. These options cover the caching of parse results to avoid recomputation, and the segregation of grammar rules and dictionary.

The grammar we've been using has only seven terminal symbols, {the, a, cat, dog, mouse, chases, scares}, and each of these is defined in the grammar.

If we wanted to be able to parse thousands of terminal symbols then the grammar would be quite large. To solve this problem we can separate the grammar rules and the dictionary. Here a dictionary means a set of terminal symbols. To do this we instruct the BUP compiler to generate a parser that calls a dictionary look-up predicate when terminals aren't found in the grammar. We could then write this predicate to dynamically parse these terminals, e.g., search a large file on disk containing lots of words looking for the grammar terminal passed as an argument to the predicate.

The problem with this solution is that terminal processing now takes longer time, potentially *much* longer. If we are prepared to sacrifice some space, we could use a further option which caches terminals recognised by our dictionary look-up predicate in the clause store to avoid recomputation, meaning that the next time the terminal is processed our predicate isn't called as the terminal is found in the cache.

Since caching results makes no sense if there is no dictionary look-up predicate, we are left with the following options which can be passed to bup_compile/3 — should no such option be given, there is no dictionary look-up and no caching of results.

`lookup_dictionary(Functor, off)`
: Call the predicate `Functor/4` when a terminal is not found in the grammar. Do not cache successes.

`lookup_dictionary(Functor, on)`
: Call the predicate `Functor/4` when a terminal is not found in the grammar. Cache successes to avoid recomputation.

The following is an example of using a predicate function for dictionary look-up. We first compile the grammar given earlier to use dictionary look-up, and we tell the compiler that we will define a predicate `vocabulary/4` that the parser should call. We then load the compiled grammar.

```
| ?- bup_compile('grammar.bup', 'grammar.pl',
                 [dictionary_lookup(vocabulary, on)]).
BUP compilation
input file  : grammar.bup
output file : grammar.pl
Finished
% yes
| ?- [grammar].
* Singletons
* File : '/home/bwat/BarrysProlog/bin/linux_x86_64/grammar.pl'
* Position (start of term) : line number 8, column number 0.
* Singleton variables : [H,G,F,E]
* Singletons
```

```
* File : '/home/bwat/BarrysProlog/bin/linux_x86_64/grammar.pl'
* Position (start of term) : line number 20, column number 0.
* Singleton variables : [E]
% yes
```

We now need to define `vocabulary/4`. For the purposes of this example we'll just add a `noun` and a `verb`. The format of the predicate is specified to be

```
vocabulary(+NonTerminal, -ArgumentList, +Input, +Remainder)
```

`Input` and `Remainder` together form a difference list. It is the difference between these lists that is the terminal which is to be parsed as a type of `NonTerminal` with its argument in the list `ArgumentList`.

We can see from the following two clauses how to state that "horse" is a `noun`, and "dislikes" is a `verb`.

```
| ?- assert(vocabulary(noun, [n(horse)], [horse|Rest], Rest)).
Rest = _553920 ?
% yes
| ?- assert(vocabulary(verb, [v(dislikes)], [dislikes|Rest], Rest)).
Rest = _554880 ?

% yes
```

We now have everything in place, so we can test the new dictionary entries defined by `vocabulary/4`.

```
| ?- bup_goal(s, [Tree], [the, horse, dislikes, the, dog, '.'], []).
Tree = s(np(d(the),n(horse)),vp(v(dislikes),np(d(the),n(dog)))) ?

% yes
```

There is another option that can be given which caches parse results. There are in fact two caches:

- A cache of successful parses. Each entry corresponds to a call of `bup_goal/4` — described later — which succeeded.
- A cache of failed parses. Each entry corresponds to a non-terminal and an input sequence.

Goal caching is controlled by these two options:

`cache_goals(off)`
: Do not cache parse results.

`cache_goals(on)`
: Cache parse results.

The lack of such an option is equivalent to passing `cache_goals(off)`.

6.1 Predicates

6.1.1 bup_compile/2

Synopsis

```
bup_compile(+FileInput, +FileOutput)
```

Description

Behaves as if it were defined as:

```
bup_compile(FileInput, FileOutput) :-
        bup_compile(FileInput, FileOutput, []).
```

Examples

See bup_compile/3.

Errors

See bup_compile/3.

See also

bup_compile/3.

6.1.2 bup_compile/3

Synopsis

```
bup_compile(+FileInput, +FileOutput, +Options)
```

Description

A BUP grammar is read in from `FileInput`. The parser source code, which is generated in accordance with `Options`, is written to `FileOutput`. The argument `Options` is a list of terms. Each element of this list must be one of the following:

`dictionary_lookup(Functor, off)` Terminals not found in the parser's grammar are searched for by calling the predicate `Functor/4`.

`dictionary_lookup(Functor, on)` Terminals not found in the parser's grammar are searched for by calling the predicate `Functor/4`. All terminals successfully found are cached in the clause store using the predicate bup_wf_dict/4.

`cache_goals(off)` No goals are cached.

`cache_goals(on)` Successful goals are cached in the clause store using the predicate `bup_wf_goal/4`. Unsuccessful goals are cached in the clause store using the predicate `bup_fail_goal/2`.

The default option in the case `Options=[]`, is `[cache_goals(off)]`. In the event that `Options` contains conflicting options, the last such option in the list takes precedence.

Examples

```
| ?- bup_compile('grammar.bup', 'grammar.pl',
                 [dictionary_lookup(vocabulary, on)]).
BUP compilation
input file  : grammar.bup
output file : grammar.pl
Finished
% yes
```

Errors

`bup_error(could_not_open, Filename, 0)` The file identified by `Filename` could not be opened.

`bup_error(could_not_read, Filename, Position)` Input could not be read from the grammar in file `Filename` at stream position `Position`.

`bup_error(empty_production, Filename, Position)` There is an empty production in the grammar in file `Filename` at stream position `Position`.

`bup_error(grammar_cycle, NonTerminal)` The input grammar contains at least one cycle. The atoms in the list `NonTerminal` are part of a cycle.

`domain_error(bup_compile_option, Option)` The argument `Option` was not a valid BUP compiler option.

`instantiation_error` One of the arguments was not fully instantiated.

See also

`stream_position_byte_count/2`, `stream_position_character_count/2`, `stream_position_line_count/2`, `stream_position_line_position/2`.

6.1.3 bup_fail_goal/2

Synopsis

```
bup_fail_goal(?NonTerminal, ?Input)
```

Description

This dynamic predicate is asserted by the parser when the compile option `cache_goals(on)` is given. The parser then uses this predicate as a cache of pairs of grammar non-terminals and input sequences. Such a pair indicates a failure to parse the input sequence as being of the type described by the non-terminal. This cache is used to avoid recomputation in subsequent parses. You may wish to use `retractall/1` to remove this predicate and flush the cache.

Examples

Here we assume that we have compiled the file `'grammar.bup'`, defined above, with the option `cache_goals(on)`.

```
| ?- bup_goal(s, A, [the, cat, chases, the, rat, '.'], []).
% no
| ?- listing(bup_fail_goal/2).
bup_fail_goal(noun,[rat,.]).
bup_fail_goal(np,[the,rat,.]).
bup_fail_goal(vp,[chases,the,rat,.]).
bup_fail_goal(s,[the,cat,chases,the,rat,.]).
% yes
```

Errors

None.

See also

`bup_compile/3`, `retractall/1`.

6.1.4 bup_goal/4

Synopsis

```
bup_goal(?NonTerminal, ?ArgumentList, ?Input, ?Remainder)
```

Description

Succeeds if `Input` and `Remainder` form a difference list and the actual difference can be parsed as a `NonTerminal` with grammar arguments `ArgumentList`.

Examples

Here we assume we have loaded the grammar contained in `'grammar.pl'` which was defined above:

```
| ?- bup_goal(np, Args, [the, cat, chases, the, mouse, '.'], X).
Args = [np(d(the),n(cat))]
X = [chases,the,mouse,.] ?

% yes
```

Errors

None.

See also

bup_compile/3.

6.1.5 bup_wf_dict/4

Synopsis

```
bup_wf_dict(?NonTerminal, ?ArgumentList, ?Input, ?Remainder)
```

Description

This dynamic predicate is asserted by the parser when the grammar was compiled with the option `dictionary_lookup(Functor, on)`. The predicate `Functor/4` is assumed to be defined elsewhere by the programmer. The parser then uses this predicate as a cache of bup_goal/4 calls which succeed. This cache is used to avoid recomputation in subsequent parses. You may wish to use retractall/1 to remove this predicate and flush the cache.

Examples

Here we assume that we have compiled the file `'grammar.bup'`, defined above, with the option `dictionary_lookup(vocabulary, on)`.

```
| ?- assert(vocabulary(noun, [n(vole)], [vole|X], X)).
X = _553600 ?

% yes
| ?- bup_goal(s, Args, [the, cat, chases, the, vole, '.'], []).
Args = [s(np(d(the),n(cat)),vp(v(chases),np(d(the),n(vole))))] ?

% yes
| ?- listing(bup_wf_dict/4).
bup_wf_dict(det,[d(the)],[the|A],A).
bup_wf_dict(noun,[n(cat)],[cat|A],A).
bup_wf_dict(verb,[v(chases)],[chases|A],A).
```

```
bup_wf_dict(noun,[n(vole)],[vole|A],A).
% yes
```

Errors

None.

See also

bup_compile/3, retractall/1.

6.1.6 bup_wf_goal/4

Synopsis

```
bup_wf_goal(?NonTerminal, +ArgumentList, ?Input, ?Remainder)
```

Description

This dynamic predicate is asserted by the parser when the compile option `cache_goals(on)` is given. The parser then uses this predicate as a cache arguments to bup_goal/4 calls which succeed. This cache is used to avoid recomputation in subsequent parses. You may wish to use retractall/1 to remove this predicate and flush the cache.

Examples

Here we assume that we have compiled the file `'grammar.bup'`, defined above, with the option `cache_goals(on)`.

```
| ?- bup_goal(s, A, [the, cat, chases, the, mouse, '.'], []).
A = [s(np(d(the),n(cat)),vp(v(chases),np(d(the),n(mouse))))] ?

% yes
| ?- listing(bup_wf_goal/4).
bup_wf_goal(noun,
            [n(cat)],
            [cat,chases,the,mouse,.],
            [chases,the,mouse,.]).
bup_wf_goal(noun,[n(mouse)],[mouse,.],[.]).
bup_wf_goal(np,
            [np(d(the),n(mouse))],
            [the,mouse,.],
            [.]).
bup_wf_goal(vp,
            [vp(v(chases),np(d(the),n(mouse)))],
```

```
            [chases,the,mouse,.],
            [.]).
bup_wf_goal(s,
             [s(np(d(the),n(cat)),
                vp(v(chases),np(d(the),n(mouse))))],
             [the,cat,chases,the,mouse,.],
             []).
% yes
```

Errors

None.

See also

bup_compile/3, retractall/1.

Chapter 7

The graphs library

The predicates in this library are intended to help the programmer implement graph theoretical algorithms. To use the predicates in this library, you will need to load the `graphs.fasl` file. One way to do this is with the following call: `ensure_loaded(runtime(graphs))`.

In this library, two representations of graphs are used:

- The P-representation which is a list of pairs where each element in the pair is a single vertex. Example: `[a-b, b-c, b-a, c-a]`.
- The S-representation which is a list of pairs where the first element of the pair is a vertex and the second element is a list of immediate neighbour vertices of the first element. The list of pairs is sorted (see `keysort/2`), as is the list of neighbours (see `sort/2`). Example: `[a-[b],b-[a,c],c-[a]]`.

The reason for the two representations is that they are each suited for different uses. Some algorithms are easier to express if the S-representation is used, and graphs are easier for a human to give as input in P-representation.

7.1 Predicates

7.1.1 `compose/3`

Synopsis

```
compose(+G1, +G2, ?G3)
```

Description

Succeeds if the graph `G3` is the composition of the graphs `G1` and `G2`. All graphs are in S-representation.

Examples

```
| ?- p_to_s_graph([u-v], S1),
     p_to_s_graph([u-v, v-w],S2),
     compose(S1, S2, S3).
S1 = [u-[v],v-[]]
S2 = [u-[v],v-[w],w-[]]
S3 = [u-[w],v-[],w-[]] ?

% yes
```

Errors

None.

See also

None.

7.1.2 p_member/3

Synopsis

```
p_member(?V1, ?V2, +G)
```

Description

Succeeds if there is an edge from vertex `V1` to vertex `V2` in the graph `G` which is given in P-representation.

Examples

```
| ?- p_member(V1, V2, [a-b, b-c]).
V1 = a
V2 = b ? ;

V1 = b
V2 = c ? ;

% no
```

Errors

None.

See also

s_member/3.

7.1.3 p_to_s_graph/2

Synopsis

```
p_to_s_graph(+P, ?S)
```

Description

This predicate takes a graph `P` in P-representation and unifies `S` with the equivalent S-representation graph.

Examples

```
| ?- p_to_s_graph([a-b, b-c, c-d, a-c],S), s_to_p_graph(S, P).
S = [a-[b,c],b-[c],c-[d],d-[]]
P = [a-b,a-c,b-c,c-d] ?

% yes
```

Errors

None.

See also

s_to_p_graph/2.

7.1.4 p_transpose/2

Synopsis

```
p_transpose(+G1, ?G2)
```

Description

This predicate takes a graph `G1` in P-representation, transposes that graph, and unifies `G2` with the result in P-representation.

Examples

```
| ?- p_transpose([a-b, b-c, c-d, a-c],S).
S = [b-a,c-b,d-c,c-a] ?

% yes
```

Errors

None.

See also

s_transpose/2.

7.1.5 s_member/3

Synopsis

```
s_member(?V1, ?V2, +G)
```

Description

Succeeds if there is an edge from vertex `V1` to vertex `V2` in the graph `G` which is given in S-representation.

Examples

```
| ?- p_to_s_graph([a-b, b-c], S), s_member(V1, V2, S).
S = [a-[b],b-[c],c-[]]
V1 = a
V2 = b ? ;

S = [a-[b],b-[c],c-[]]
V1 = b
V2 = c ? ;

% no
```

Errors

None.

See also

p_member/3.

7.1.6 s_to_p_graph/2

Synopsis

```
s_to_p_graph(+S, ?P)
```

Description

This predicate takes a graph `S` in S-representation and unifies `P` with the equivalent P-representation graph.

Examples

```
| ?- p_to_s_graph([a-b, b-c, c-d, a-c],S), s_to_p_graph(S, P).
S = [a-[b,c],b-[c],c-[d],d-[]]
P = [a-b,a-c,b-c,c-d] ?

% yes
```

Errors

None.

See also

p_to_s_graph/2.

7.1.7 s_to_p_trans/2

Synopsis

```
s_to_p_trans(+S, ?P)
```

Description

This predicate takes a graph `S` in S-representation, transposes that graph, and unifies `P` with the result in P-representation.

Examples

```
| ?- p_to_s_graph([a-b, b-c, c-d, a-c],S), s_to_p_trans(S, P).
S = [a-[b,c],b-[c],c-[d],d-[]]
P = [b-a,c-a,c-b,d-c] ?

% yes
```

Errors

None.

See also

s_to_p_graph/2.

7.1.8 s_transpose/2

Synopsis

```
s_transpose(+G1, ?G2)
```

Description

This predicate takes a graph `G1` in S-representation, transposes that graph, and unifies `G2` with the result in S-representation.

Examples

```
| ?- p_to_s_graph([a-b, b-c, c-d, a-c],S),
     s_transpose(S, S2),
     s_to_p_graph(S2, Pt).
S = [a-[b,c],b-[c],c-[d],d-[]]
S2 = [a-[],b-[a],c-[a,b],d-[c]]
Pt = [b-a,c-a,c-b,d-c] ?

% yes
```

Errors

None.

See also

s_to_p_trans/2.

7.1.9 top_sort/2

Synopsis

```
top_sort(+G, ?L)
```

Description

Succeeds if the topological sort of the graph `G` in S-representation is the list of vertices `L`.

Examples

```
p_to_s_graph([a-b, b-d, c-d, d-e],S), top_sort(S, Sorted).
S = [a-[b],b-[d],c-[d],d-[e],e-[]]
Sorted = [a,b,c,d,e] ?

% yes
```

Errors

None.

See also

None.

7.1.10 vertices/2

Synopsis

```
vertices(+G, ?V)
```

Description

This predicate takes a graph `G` in S-representation and unifies `V` with a list of the vertices found in that graph.

Examples

```
| ?- p_to_s_graph([a-b, b-c, c-d, a-c],S), vertices(S, V).
S = [a-[b,c],b-[c],c-[d],d-[]]
V = [a,b,c,d] ?

% yes
```

Errors

None.

See also

None.

7.1.11 warshall/2

Synopsis

```
warshall(+G1, ?G2)
```

Description

This predicate takes a graph `G1` in S-representation and unifies `G2` with a graph in S-representation which is the transitive closure of `G1`.

Examples

```
| ?- p_to_s_graph([a-b, b-c, c-d, a-c],S),
     warshall(S, Closure),
     s_to_p_graph(Closure, ClosureP).
S = [a-[b,c],b-[c],c-[d],d-[]]
Closure = [a-[b,c,d],b-[c,d],c-[d],d-[]]
ClosureP = [a-b,a-c,a-d,b-c,b-d,c-d] ?

% yes
```

Errors

None.

See also

None.

Chapter 8

The `lists` library

To use the predicates in this library, you will need to load the `lists.fasl` file. One way to do this is with the following call: `ensure_loaded(runtime(lists))`.

8.1 Predicates

8.1.1 `correspond/4`

Synopsis

```
correspond(?E1, ?L1, ?L2, ?E2)
```

Description

Succeeds if the position of element `E1` in list `L1` corresponds to the position of element `E2` in list `L2`. This predicate is deterministic.

Examples

```
| ?- correspond(1, [1,2], [a,b], E).
E = a ?

% yes
| ?- correspond(E, [1,2], [a,b], b).
E = 2 ?

% yes
```

Errors

None.

See also

None.

8.1.2 delete/3

Synopsis

```
delete(?L1, ?E, ?L2)
```

Description

Succeeds is the list `L2` is the result of deleting all occurrences of `E` from the list `L1`. This predicate is deterministic.

Examples

```
| ?- delete([1,1,1], 1, L).
L = [] ?

% yes
| ?- delete([1,2,3], 4, L).
L = [1,2,3] ?

% yes
```

Errors

None.

See also

None.

8.1.3 last/2

Synopsis

```
last(?E, ?L)
```

Description

Succeeds if `E` is the last element of the list `L`.

Examples

```
| ?- last(E, [1,2,3]).
E = 3 ?

% yes
| ?- last(E, []).
% no
```

Errors

None.

See also

None.

8.1.4 nextto/3

Synopsis

```
nextto(?E1, ?E2, ?L)
```

Description

Succeeds if the list `L` contains the elements `E1` and `E2` and that these two elements are positioned next to each other.

Examples

```
| ?- nextto(A, B, [1,2,3]).
A = 1
B = 2 ? ;

A = 2
B = 3 ? ;

% no
```

Errors

None.

See also

None.

8.1.5 nmember/3

Synopsis

```
nmember(?E, ?L, ?N)
```

Description

Succeeds if the list `L` contains the element `E` at index `N`.

Examples

```
| ?- nmember(E, [a,b,c], N).
E = a
N = 1 ? ;

E = b
N = 2 ? ;

E = c
N = 3 ? ;

% no
```

Errors

None.

See also

None.

8.1.6 nmembers/3

Synopsis

```
nmembers(?Ns, ?L, ?Es)
```

Description

Succeeds if the list `L` contains the elements of the list `Es` at the corresponding indices of the list `Ns`.

Examples

```
| ?- nmembers([2], [a,b,c], L).
L = [b] ?
```

```
% yes
| ?- nmembers([3,2,1], [a,b,c], L).
L = [c,b,a] ?

% yes
```

Errors

None.

See also

None.

8.1.7 nth1/3

Synopsis

```
nth1(?N, +L, +E)
nth1(+N, +L, ?E)
```

Description

Succeeds if the list `L` at index `N` contains the element `E`. The first element has index 1. This predicate is deterministic.

Examples

```
| ?- nth1(N, [a,b,c], c).
N = 3 ?

% yes
| ?- nth1(2, [a,b,c], E).
E = b ?

% yes
```

Errors

None.

See also

nth0/3.

8.1.8 nth0/4

Synopsis

```
nth0(?N, +L1, +E, ?L2)
nth0(+N, ?L1, ?E, ?L2)
```

Description

Succeeds if the list `L1` at index `N` contains the element `E` and the remainder of the list is `L2`. The first element has index 0. This predicate is deterministic.

Examples

```
| ?- nth0(3, L, x, [a,b,c,d]).
L = [a,b,c,x,d] ?

% yes
| ?- nth0(N, [a,b,c,d], c, L).
N = 2
L = [a,b,d] ?

% yes
```

Errors

None.

See also

nth1/4.

8.1.9 nth1/4

Synopsis

```
nth1(?N, +L1, +E, ?L2)
nth1(+N, ?L1, ?E, ?L2)
```

Description

Succeeds if the list `L1` at index `N` contains the element `E` and the remainder of the list is `L2`. The first element has index 1. This predicate is deterministic.

Examples

```
| ?- nth1(4, L, x, [a,b,c,d]).
L = [a,b,c,x,d] ?

% yes
| ?- nth1(N, [a,b,c,d], c, L).
N = 3
L = [a,b,d] ?

% yes
```

Errors

None.

See also

nth0/4.

8.1.10 numlist/3

Synopsis

```
numlist(+Low, +High, ?L)
```

Description

Succeeds if the list `L` is the sequence of numbers from `Low` to `High` sorted in numerical order.

Examples

```
| ?- numlist(1, 3, L).
L = [1,2,3] ?

% yes
| ?- numlist(4, 3, L).
% no
```

Errors

Since `Low` and `High` are evaluated, errors may be thrown by eval/2.

See also

None.

8.1.11 perm/2

Synopsis

```
perm(?L1, ?L2)
```

Description

Succeeds if the two arguments are lists which are permutations of each other.

Examples

```
| ?- perm([a,b,c], L).
L = [a,b,c] ? ;

L = [a,c,b] ? ;

L = [b,a,c] ? ;

L = [b,c,a] ? ;

L = [c,a,b] ? ;

L = [c,b,a] ? ;

% no
```

Errors

None.

See also

None.

8.1.12 perm2/4

Synopsis

```
perm2(?T1, ?T2, ?T3, ?T4)
```

Description

Succeeds if the first two arguments are a permutation of the second two arguments. This predicate is defined as follows:

```
perm2(T1, T2, T1, T2).
perm2(T1, T2, T2, T1).
```

Examples

```
| ?- assert((zero(A*B) :- perm2(0,_,A,B))).
A = _548608
B = _549280 ?

% yes
| ?- zero(5*0).
% yes
| ?- zero(A*B).
A = 0
B = _547856 ? ;

A = _547184
B = 0 ? ;

% no
```

Errors

None.

See also

None.

8.1.13 `remove_dups/2`

Synopsis

`remove_dups(+L1, ?L2)`

Description

The list `L2` is the result of taking the list `L1` and removing all duplicated elements. You will want the list `L1` to be ground to avoid problems with unintended bindings. This predicate is defined as follows:

```
remove_dups(L1, L2) :-
        sort(L1, L2).
```

Examples

```
| ?- remove_dups([1,1,1], L).
L = [1] ?

% yes
```

```
| ?- remove_dups([1,2,3], L).
L = [1,2,3] ?

% yes
```

Errors

See sort/2.

See also

sort/2.

8.1.14 rev/2

Synopsis

```
rev(?L1, ?L2)
```

Description

Behaves as if it were defined as follows:

```
rev(L1, L2) :- reverse(L1, L2).
```

Examples

See reverse/2.

Errors

See reverse/2.

See also

reverse/2.

8.1.15 same_length/2

Synopsis

```
same_length(?L1, ?L2)
```

Description

Succeeds if the lists L1 and L2 have the same length.

Examples

```
| ?- same_length([a,b,c], L).
L = [_554624,_554672,_554720] ?

% yes
| ?- same_length([1,2,3], [a,b,c]).
% yes
```

Errors

None.

See also

None.

8.1.16 `select/4`

Synopsis

`select(?E1, ?L1, ?E2, ?L2)`

Description

Succeeds if the lists `L1` and `L2` have the same length and differ only in a certain list position where `L1` has the element `E1` and `L2` has `E2`.

Examples

```
| ?- select(1, [1,1,1], b, X).
X = [b,1,1] ? ;

X = [1,b,1] ? ;

X = [1,1,b] ? ;

% no
```

Errors

None.

See also

`selectchk/4`.

8.1.17 selectchk/4

Synopsis

```
selectchk(?E1, ?L1, ?E2, ?L2)
```

Description

This is a deterministic version of select/4. This predicate behaves as if it were defined as follows:

```
selectchk(E1, L1, E2, L2) :-
        once(select(E1, L1, E2, L2)).
```

Examples

```
| ?- selectchk(1, [1,1,1], b, X).
X = [b,1,1] ? ;

% no
```

Errors

None.

See also

select/4.

8.1.18 select/3

Synopsis

```
select(?E1, ?L1, ?L2)
```

Description

This predicate behaves as if it were defined as follows:

```
select(E1, L1, L2) :-
        del(E1, L1, L2).
```

Examples

```
| ?- select(E, [1,2,3], L).
E = 1
L = [2,3] ? ;

E = 2
L = [1,3] ? ;

E = 3
L = [1,2] ? ;

% no
```

Errors

None.

See also

del/3, selectchk/3.

8.1.19 selectchk/3

Synopsis

```
selectchk(?E1, ?L1, ?L2)
```

Description

This is a deterministic version of select/3. This predicate behaves as if it were defined as follows:

```
selectchk(E1, L1, L2) :-
        once(select(E1, L1, L2)).
```

Examples

```
| ?- selectchk(E, [1,2,3], L).
E = 1
L = [2,3] ? ;

% no
```

Errors

None.

See also

select/3.

8.1.20 shorter_list/2

Synopsis

```
shorter_list(?L1, ?L2)
```

Description

Succeeds if the list L1 is shorter in length than the list L2. This predicate can be used to exhaustively solve for L1 given L2 but not vice versa.

Examples

```
| ?- shorter_list(L, [1,2,3]).
L = [] ? ;

L = [_555216] ? ;

L = [_555216,_555264] ? ;

% no
| ?- shorter_list([1,2], L).
L = [_554544,_554592,_554640|_554656] ? ;

% no
```

Errors

None.

See also

None.

8.1.21 subseq/3

Synopsis

```
subseq(?L1, ?L2, ?L3)
```

Description

Succeeds if the list `L2` is a sub-sequence of the list `L1` and those elements not in `L2` are found in the list `L3`. The order of the elements in `L1` is preserved in `L1` and `L2`.

Examples

```
| ?- subseq([a,b], L1, L2).
L1 = []
L2 = [a,b] ? ;

L1 = [b]
L2 = [a] ? ;

L1 = [a]
L2 = [b] ? ;

L1 = [a,b]
L2 = [] ? ;

% no
```

Errors

None.

See also

None.

8.1.22 `subseq0/2`

Synopsis

```
subseq0(?L1, ?L2)
```

Description

Succeeds if the list `L2` is a not necessarily proper sub-sequence of the list `L1`.

Examples

```
| ?- subseq0([1,2], L).
L = [1,2] ? ;

L = [2] ? ;
```

```
L = [] ? ;

L = [1] ? ;

% no
```

Errors

None.

See also

subseq1/2.

8.1.23 subseq1/2

Synopsis

```
subseq1(?L1, ?L2)
```

Description

Succeeds if the list L2 is a proper sub-sequence of the list L1.

Examples

```
| ?- subseq1([1,2], L).
L = [2] ? ;

L = [] ? ;

L = [1] ? ;

% no
```

Errors

None.

See also

subseq0/2.

8.1.24 sumlist/2

Synopsis

```
sumlist(+L, ?S)
```

Description

Succeeds if the sum of all of the elements of the list `L` is a equal to `S`.

Examples

```
| ?- sumlist([1,2,3], S).
S = 6 ?

% yes
| ?- sumlist([], S).
S = 0 ?

% yes
```

Errors

Since the elements of the list are evaluated, errors may be thrown by `eval/2`.

See also

None.

Chapter 9

The ordset library

The predicates in this library implement a set data type using sorted lists (ordered according to '@<'/2). To use the predicates in this library, you will need to load the ordset.fasl file. One way to do this is with the following call: ensure_loaded(runtime(ordset))

The advantage with ordered sets is that certain operations take much fewer instructions. As an example, consider the insertion of an element E1 into an ordered set: when comparing E1 to each element of the list to avoid having duplicate entries, we know that as soon as we find an element E2 in the list such that E2@>E1, we have compared all of the elements we need to.

9.1 Predicates

9.1.1 list_to_ord_set/2

Synopsis

```
list_to_ord_set(+L, -O)
```

Description

This predicate converts the list L into an ordered set O. This is equivalent to a call of sort(L,O).

Examples

```
| ?- list_to_ord_set([3,3,2,2,1,1], O).
O = [1,2,3] ?

% yes
```

Errors

None.

See also

sort/2.

9.1.2 ord_all_nonempty_subsets/2

Synopsis

```
ord_all_nonempty_subsets(+O1, ?O2)
```

Description

This predicate behaves as if it were defined as:

```
ord_all_nonempty_subsets(O1, O2) :-
        ord_powerset(O1, P),
        ord_subtract(P, [[]], O2).
```

Examples

```
| ?- ord_powerset([1,2,3], O).
O = [[],[1],[1,2],[1,2,3],[1,3],[2],[2,3],[3]] ?

% yes
| ?- ord_all_nonempty_subsets([1,2,3], O).
O = [[1],[1,2],[1,2,3],[1,3],[2],[2,3],[3]] ?

% yes
```

Errors

None.

See also

ord_powerset/2, ord_subtract/3.

9.1.3 ord_all_subsets/2

Synopsis

```
ord_all_subsets(+O1, ?O2)
```

Description

This predicate behaves as if it were defined as:

```
ord_allsubsets(O1, O2) :-
        ord_powerset(O1, O2).
```

Examples

See ord_powerset/2.

Errors

None.

See also

ord_all_subsets/3, ord_powerset/2.

9.1.4 ord_all_subsets/3

Synopsis

```
ord_all_subsets(+O1, +N, ?O2)
```

Description

This predicate succeeds if O2 is the ordered set of all subsets of the ordered set O1 which are of cardinality N.

Examples

```
| ?- ord_all_subsets([1,2,3], 2, O).
O = [[1,2],[1,3],[2,3]] ?

% yes
| ?- ord_all_subsets([1,2,3], 0, O).
O = [[]] ?

% yes
```

Errors

None.

See also

ord_all_subsets/2.

9.1.5 ord_all_unordered_pairs/3

Synopsis

```
ord_all_unordered_pairs(+O1, +O2, ?O3)
```

Description

Succeeds if the ordered set O3 contains all unordered pairs of the ordered sets O1 and O2. Each pair is itself an ordered set which means that the order of the first two arguments does not matter.

Examples

```
| ?- ord_all_unordered_pairs([1,2,3], [a,b], R).
R = [[1,a],[1,b],[2,a],[2,b],[3,a],[3,b]] ?

% yes
| ?- ord_all_unordered_pairs([a,b], [1,2,3], R).
R = [[1,a],[1,b],[2,a],[2,b],[3,a],[3,b]] ?

% yes
```

Errors

None.

See also

ord_product/3.

9.1.6 ord_disjoint/2

Synopsis

```
ord_disjoint(+O1, +O2)
```

Description

Succeeds if the two ordered sets O1 and O2 have no element in common.

Examples

```
| ?- ord_disjoint([1,2,3], [4,5,6]).
% yes
| ?- ord_disjoint([1,2,3], [3,4,5]).
% no
```

Errors

None.

See also

None.

9.1.7 ord_insert/3

Synopsis

```
ord_insert(+O1, +E, ?O2)
```

Description

Succeeds if the ordered set `O2` is equal to the ordered set `O1` with the element `E` inserted into it.

Examples

```
| ?- ord_insert([1,2,3], 4, O).
O = [1,2,3,4] ?

% yes
| ?- ord_insert([1,2,3], 2, [1,2,3]).
% yes
```

Errors

None.

See also

None.

9.1.8 ord_intersect/2

Synopsis

```
ord_intersect(+O1, +O2)
```

Description

Succeeds if the ordered sets O1 and O2 have at least one element in common.

Examples

```
| ?- ord_intersect([1,2,3], [3,4,5]).
% yes
| ?- ord_intersect([1], [2]).
% no
```

Errors

None.

See also

ord_intersect/3.

9.1.9 ord_intersect/3

Synopsis

```
ord_intersect(+O1, +O2, ?O3)
```

Description

Succeeds if the ordered set O3 is the set of all common elements of the ordered sets O1 and O2.

Examples

```
| ?- ord_intersect([1,2,3], [3,4,5], O).
O = [3] ?

% yes
| ?- ord_intersect([1,2,3], [4,5,6], O).
O = [] ?

% yes
```

Errors

None.

See also

ord_intersect/2.

9.1.10 ord_powerset/2

Synopsis

```
ord_powerset(+O1, ?O2)
```

Description

Succeeds if the ordered set O2 is the powerset of the ordered set O1.

Examples

```
| ?- ord_powerset([], P).
P = [[]] ?

% yes
| ?- ord_powerset([1,2,3], P).
P = [[],[1],[1,2],[1,2,3],[1,3],[2],[2,3],[3]] ?

% yes
```

Errors

None.

See also

None.

9.1.11 ord_product/3

Synopsis

```
ord_product(+O1, +O2, ?O3)
```

Description

Succeeds if the ordered set O3 is the Cartesian product of the ordered sets O1 and O2.

Examples

```
| ?- ord_product([1,2,3], [a,b], R).
R = [[1,a],[1,b],[2,a],[2,b],[3,a],[3,b]] ?

% yes
| ?- ord_product([a,b], [1,2,3], R).
R = [[a,1],[a,2],[a,3],[b,1],[b,2],[b,3]] ?
```

```
% yes
```

Errors

None.

See also

ord_all_unordered_pairs/3.

9.1.12 ord_seteq/2

Synopsis

```
ord_seteq(+O1, +O2)
```

Description

Succeeds if the ordered sets O1 and O2 are equal.

Examples

```
| ?- ord_seteq([1,2,3], [1,2,3]).
% yes
| ?- ord_seteq([1,2,3], [1,2,3,4]).
% no
```

Errors

None.

See also

None.

9.1.13 ord_subset/2

Synopsis

```
ord_subset(+O1, +O2)
```

Description

Succeeds if the ordered set O1 is a subset of the ordered set O2.

Examples

```
| ?- ord_subset([2,3], [1,2,3]).
% yes
| ?- ord_subset([1,2,3], [2,3]).
% no
```

Errors

None.

See also

None.

9.1.14 ord_subtract/3

Synopsis

```
ord_subtract(+O1, +O2, ?O3)
```

Description

Succeeds if the ordered set O3 is the difference between the ordered sets O1 and O2.

Examples

```
| ?- ord_subtract([1,2,3], [2,3], O).
O = [1] ?

% yes
| ?- ord_subtract([1,2,3], [4,5], [1,2,3]).
% yes
```

Errors

None.

See also

ord_symdiff/3.

9.1.15 ord_symdiff/3

Synopsis

```
ord_symdiff(+O1, +O2, ?O3)
```

Description

Succeeds if the ordered set O3 is the symmetric difference between the ordered sets O1 and O2.

Examples

```
| ?- ord_symdiff([1,2,3], [2,3], O).
O = [1] ?

% yes
| ?- ord_symdiff([2,3], [1,2,3], O).
O = [1] ?

% yes
| ?- ord_subtract([2,3], [1,2,3], O).
O = [] ?

% yes
```

Errors

None.

See also

ord_subtract/3.

9.1.16 ord_union/3

Synopsis

```
ord_union(+O1, +O2, ?O3)
```

Description

Succeeds if the ordered set O3 is the union of the ordered sets O1 and O2.

Examples

```
| ?- ord_union([1,2,3], [2,3], O).

O = [1,2,3] ?

% yes
```

Errors

None.

See also

None.

Chapter 10

The `printtree` library

To use the predicates in this library, you will need to load the `printtree.fasl` file. One way to do this is with the following call:
`ensure_loaded(runtime(printtree))`.

10.1 Predicates

10.1.1 `print_tree/1`

Synopsis

`print_tree(+Term)`

Description

Behaves as if it were defined as:

```
print_tree(Term) :-
        current_input(Stream),
        print_tree(Stream, Term).
```

Examples

See `print_tree/2`.

Errors

None.

See also

`print_tree/2`.

10.1.2 print_tree/2

Synopsis

```
print_tree(+Stream, +Term)
```

Description

Prints the argument `Term` as a tree on `Stream`.

Examples

```
| ?- print_tree(s(np(det(the),noun(man)),vp(v(sees)))).
         s
      ___|___
     /       \
    np       vp
   __|_       |
  /    \      |
 det  noun    v
  |     |     |
  |     |     |
 the   man  sees


% yes
| ?- print_tree(user_output, 99).
 99


% yes
| ?- print_tree(user_output, 99.9).
               $float
           _______|______
          /           |  \
 14397107288778001613  7  64


% yes
```

Errors

None.

See also

None.

Chapter 11

The `readin` library

To use the predicates in this library, you will need to load the `readin.fasl` file. One way to do this is with the following call:
`ensure_loaded(runtime(readin))`.

11.1 Predicates

11.1.1 `read_in/1`

Synopsis

`read_in(-Words)`

Description

Behaves as if it were defined as:

```
read_in(Words) :-
        current_input(Stream),
        read_in(Stream, Words).
```

Examples

See `read_in/2`.

Errors

See `read_in/2`.

See also

`read_in/2`.

11.1.2 read_in/2

Synopsis

```
read_in(+Stream, -Words)
```

Description

Reads input from `Stream` until a line is input which terminates with either an end of file, or one of the characters `.`, `!`, or `?`. The argument `Words` is a list of atoms or numbers which are constructed from the characters given as input as follows:

- A whitespace character code which satisfies `prolog_lexical_ws/1` is ignored.
- A character code that either satisfies `prolog_lexical_symbol/1`, or is one of `;` or `!`, is turned into a one character atom.
- A sequence of character codes, all of which satisfy `prolog_lexical_digit/1`, is turned into a number.
- A sequence of character codes, all of which satisfy `prolog_lexical_letter/1`, is turned into an atom. Any of the character codes in the sequence that satisfy `prolog_lexical_upper_case_letter/1` are converted into the equivalent lower case letter code before the atom is built.

Examples

```
| ?- current_input(Stream), read_in(Stream, Words).
|: This is a line of input. This is another
|  line of input.
Stream = $stream(0)
Words = [this,is,a,line,of,input,.,this,is,another,line,of,input,.] ?

% yes
| ?- current_input(Stream), read_in(Stream, Words).
|: 2 4 6 8 10!
Stream = $stream(0)
Words = [2,4,6,8,10,!] ?

% yes
```

Errors

See `get_code/2`.

See also

`get_code/2`.

Chapter 12

The `readsent` library

To use the predicates in this library you will need to load the `readsent.fasl` file. One way to do this is with the following call:
`ensure_loaded(runtime(readsent))`.

This library implements predicates that may be useful for reading sentences of natural languages. The code for these predicates is based on the Edinburgh public domain Prolog library.

12.1 Predicates

12.1.1 `case_shift/2`

Synopsis

```
case_shift(+MixedCaseCodes, ?LowerCaseCodes)
```

Description

The input list of character codes `MixedCaseCodes` is converted into the output list `LowerCaseCodes` where each element of the input list which is an upper case alphabetic character code is converted into the equivalent lower case character code. All other character codes are not altered.

This predicate behaves as if it were defined as follows:

```
case_shift([Upper|Mixeds], [Lower|Lowers]) :-
        is_upper(Upper),
        Lower is Upper-"A"+"a",
        !,
        case_shift(Mixeds, Lowers).
case_shift([Lower|Mixeds], [Lower|Lowers]) :-
        case_shift(Mixeds, Lowers).
case_shift([], []).
```

Examples

```
| ?- case_shift("AaBb", Codes).
Codes = [97,97,98,98] ?

% yes
```

Errors

As each element of `MixedCaseCodes` is evaluated, errors may be thrown by `eval/2`.

See also

`is_upper/1`.

12.1.2 chars_to_atom/3

Synopsis

```
chars_to_atom(?Codes, +Input, ?InputRest)
chars_to_atom(+Codes, ?Input, ?InputRest)
```

Description

Succeeds if `Codes` is the difference between the lists `Input` and `InputRest`, and each element of `Codes` represents an alphabetic letter.

This predicate behaves as if it were defined as follows:

```
chars_to_atom([L|Ls]) --> [L], {is_letter(L)}, chars_to_atom(Ls).
chars_to_atom([]) --> [].
```

Examples

```
| ?- chars_to_atom("foo", A, B).
A = [102,111,111|_554608]
B = _554608 ?

% yes
| ?- chars_to_atom(Codes, "foo 123", Rest).
Codes = [102,111,111]
Rest = [32,49,50,51] ?

% yes
```

Errors

See `is_letter/1`.

See also

is_letter/1.

12.1.3 chars_to_integer/3

Synopsis

```
chars_to_integer(?Codes, +Input, ?InputRest)
chars_to_integer(+Codes, ?Input, ?InputRest)
```

Description

Succeeds if `Codes` is the difference between the lists `Input` and `InputRest`, and each element of `Codes` represents a decimal digit.

This predicate behaves as if it were defined as follows:

```
chars_to_integer([D|Ds]) -->
      [D],
      {is_digit(D)},
      chars_to_integer(Ds).
chars_to_integer([]) --> [].
```

Examples

```
| ?- chars_to_integer("123", A, B).
A = [49,50,51|_555088]
B = _555088 ?

% yes
| ?- chars_to_integer(Codes, "123 foo", Rest).
Codes = [49,50,51]
Rest = [32,102,111,111] ?

% yes
```

Errors

See is_digit/1.

See also

is_digit/1.

12.1.4 chars_to_string/3

Synopsis

```
chars_to_string(?Codes, +Input, ?InputRest)
chars_to_string(+Codes, ?Input, ?InputRest)
```

Description

Succeeds if `Codes` is the difference between the lists `Input` and `InputRest`, and the last element of `Codes` is a double quote character code (ASCII 34). Two adjacent double quotes in the difference list correspond to one double quote in `Codes`. This predicate is used to parse the remainder of string literals when the leading double quote has already been consumed.

This predicate behaves as if it were defined as follows:

```
chars_to_string([34|Cs]) --> [34, 34], !, chars_to_string(Cs).
chars_to_string([]) --> [34], !.
chars_to_string([C|Cs]) --> [C], chars_to_string(Cs).
```

Examples

```
% yes
| ?- chars_to_string("foo", A, B).
A = [102,111,111,34|_554928]
B = _554928 ?

% yes
| ?- chars_to_string(Codes, "foo""", Rest).
Codes = [102,111,111]
Rest = [] ?

% yes
```

Errors

None.

See also

None.

12.1.5 chars_to_words/2

Synopsis

```
chars_to_words(+Codes, ?Words)
```

Description

Behaves as if it were defined as follows:

```
chars_to_words(Codes, Words) :-
        chars_to_words(Words, Codes, []).
```

Examples

See `chars_to_words/3`.

Errors

`instantiation_error` `Codes` must be ground.

As each element of `Codes` is evaluated, errors may be thrown by `eval/2`.

See also

`chars_to_words/3`.

12.1.6 chars_to_words/3

Synopsis

```
chars_to_words(?Words, +Codes, ?CodesRest)
```

Description

The character codes in the list `Codes` are scanned and transformed into the list of terms `Words`. The remainder of `Codes` which could not be scanned is unified with `CodesRest`, i.e. `Codes` and `CodesRest` form a difference list. Each term element of the list `Words` is one of the following:

`apost` Corresponds to an apostrophe (ASCII 39) which was not followed by an 's' or 'S' (ASCII 83, or 115).

`aposts` Corresponds to an apostrophe (ASCII 39) which was followed by an 's' or 'S' (ASCII 83, or 115).

`atom(A)` Corresponds to a sequence of upper and/or lower case letters. The argument `A` is the atom whose name is represented by this sequence where all upper case letters have been shifted to lower case letters.

`integer(I)` Corresponds to a sequence of decimal digits. The argument `I` is the integer representation of these digits.

`string(S)` Corresponds to a double quote (ASCII 34) followed by a sequence of character codes terminated by a double quote. Any adjacent occurrences of the double quote character code which are seen when forming the sequence lead to a single double quote being added to the sequence and the formation of the sequence continues.

`Atom` The single character atom `Atom` corresponds to a character code which has a value greater than 32 and which is neither an alphabetic character code, a decimal digit character code, an apostrophe (ASCII 39), nor a double quote (ASCII 34).

Examples

```
| ?- chars_to_words(Words, "'quote' quote's", Rest).
Words = [apost,atom(quote),apost,atom(quote),aposts]
Rest = [] ?

% yes
| ?- chars_to_words(Words, "123 foo!", Rest).
Words = [integer(123),atom(foo),!]
Rest = [] ?

% yes
```

Errors

If `Codes` is not ground, then an endless loop will be entered. As each element of `MixedCaseCodes` is evaluated, errors may be thrown by `eval/2`.

See also

None.

12.1.7 `is_digit/1`

Synopsis

```
is_digit(+Code)
```

Description

Succeeds if `Code` represents an decimal digit.

Examples

```
| ?- is_digit(0'3).
% yes
```

Errors

As `Code` is evaluated, errors may be thrown by `eval/2`.

See also

None.

12.1.8 `is_endfile/1`

Synopsis

```
is_endfile(?Code)
```

Description

Succeeds if `Code` is the end of file character code which is -1.

Examples

```
| ?- is_endfile(Code).
Code = -1 ?

% yes
```

Errors

None.

See also

None.

12.1.9 `is_layout/1`

Synopsis

```
is_layout(+Code)
```

Description

Succeeds if `Code` represents an unprintable layout character which is any code value less than or equal to 32.

Examples

```
| ?- is_layout(32).
% yes
```

Errors

As `Code` is evaluated, errors may be thrown by `eval/2`.

See also

None.

12.1.10 `is_letter/1`

Synopsis

```
is_letter(+Code)
```

Description

Succeeds if `Code` represents an English language letter of either upper or lower case.

Examples

```
| ?- is_letter(0'a).
% yes
| ?- is_letter(0'B).
% yes
```

Errors

As `Code` is evaluated, errors may be thrown by `eval/2`.

See also

None.

12.1.11 `is_lower/1`

Synopsis

```
is_lower(+Code)
```

Description

Succeeds if `Code` represents an English language letter of lower case.

Examples

```
| ?- is_lower(0'c).
% yes
| ?- is_lower(0'C).
% no
```

Errors

As `Code` is evaluated, errors may be thrown by `eval/2`.

See also

None.

12.1.12 `is_newline/1`

Synopsis

```
is_newline(?Code)
```

Description

Succeeds if `Code` is the newline code (ASCII 10).

Examples

```
| ?- is_newline(X).
X = 10 ?

% yes
```

Errors

None.

See also

None.

12.1.13 `is_paren/2`

Synopsis

```
is_punct(+LeftCode, +RightCode)
```

Description

Succeeds if the pair `LeftCode`, `RightCode` represents one of the following pairs characters: `(`, `)`; `[`, `]`; or `{`, `}`,

Examples

```
| ?- is_paren(0'{, 0'}).
% yes
```

Errors

None.

See also

None.

12.1.14 is_period/1

Synopsis

```
is_period(+Code)
```

Description

Succeeds if `Code` represents one of the following characters `!`, `.`, or `?`.

Examples

```
| ?- is_period(0'!).
% yes
```

Errors

None.

See also

None.

12.1.15 is_punct/1

Synopsis

```
is_punct(+Code)
```

Description

Succeeds if `Code` represents one of the following characters: `:`, `;`, or `,`.

Examples

```
| ?- is_punct(0';).
% yes
```

Errors

None.

See also

None.

12.1.16 `is_upper/1`

Synopsis

```
is_upper(+Code)
```

Description

Succeeds if `Code` represents an English language letter of upper case.

Examples

```
| ?- is_upper(0'c).
% no
| ?- is_upper(0'C).
% yes
```

Errors

As `Code` is evaluated, errors may be thrown by `eval/2`.

See also

None.

12.1.17 `read_line/1`

Synopsis

```
read_line(?Codes)
```

Description

This predicate behaves as if it were defined as follows:

```
read_line(Codes) :-
        current_input(Stream),
        read_line(Stream, Codes).
```

Examples

See read_line/2.

Errors

See read_line/2.

See also

read_line/2.

12.1.18 read_line/2

Synopsis

```
read_line(+Stream, ?Codes)
```

Description

Reads a list of character codes `Codes` from `Stream` until the newline code (ASCII 10) is seen. Note that `Codes` contains the newline code.

This predicate behaves as if it were defined as follows:

```
read_line(Stream, Codes) :-
        is_newline(Newline),
        read_until(Stream, [Newline], Codes).
```

Examples

```
| ?- read_line(Codes).
|: 123 foo
Codes = [49,50,51,32,102,111,111,10] ?

% yes
```

Errors

See read_until/2.

See also

is_newline/1, read_until/2.

12.1.19 read_sent/1

Synopsis

```
read_sent(-Words)
```

Description

Behaves as if it were defined as:

```
read_sent(Words) :-
        current_input(Stream),
        read_sent(Stream, Words).
```

Examples

See read_sent/2.

Errors

See read_sent/2.

See also

read_sent/2.

12.1.20 read_sent/2

Synopsis

```
read_sent(+Stream, -Words)
```

Description

This predicate will read character codes from `Stream` until a code representing either `.`, `!`, or `?` is seen. The remaining codes up to and including the newline code are then consumed. This predicate behaves as if it were defined as follows:

```
read_sent(Stream, Words) :-
        read_until(Stream, "!?.", Chars),
        is_newline(Newline),
        read_until(Stream, [Newline], _),
        !,
```

```
        chars_to_words(Chars, Words),
        !.
```

Examples

```
| ?- current_input(S), read_sent(S, Words).
|: This is 1 sentence.
S = $stream(0)
Words = [atom(this),atom(is),integer(1),atom(sentence),.] ?

% yes
```

Errors

See read_until/3.

See also

is_newline/1, read_sentence/1, read_until/3.

12.1.21 read_sentence/1

Synopsis

```
read_sentence(-Words)
```

Description

Behaves as if it were defined as:

```
read_sentence(Words) :-
        current_input(Stream),
        read_sent(Stream, Words).
```

Examples

See read_sent/2.

Errors

See read_sent/2.

See also

read_sent/2.

12.1.22 read_sentence/2

Synopsis

```
read_sentence(+Stream, -Words)
```

Description

Behaves as if it were defined as:

```
read_sentence(Stream, Words) :-
        read_sent(Stream, Words).
```

Examples

See read_sent/2.

Errors

See read_sent/2.

See also

read_sent/2.

12.1.23 read_until/2

Synopsis

```
read_until(+Delimiters, ?Codes)
```

Description

Behaves as if it were defined as:

```
read_until(Delimiters, Codes) :-
        current_input(Stream),
        read_until(Stream, Delimiters, Codes).
```

Examples

See read_until/3.

Errors

See read_until/3.

See also

read_until/3.

12.1.24 read_until/3

Synopsis

```
read_until(+Stream, +Delimiters, ?Codes)
```

Description

Reads a list of character codes `Codes` from `Stream` until one of the codes in the list `Delimiters` is seen. The character code -1, which represents an end of file, is always considered to be a delimiter. Note, `Codes` contains the delimiter code.

Examples

```
| ?- current_input(S), read_until(S, [13, 10], Codes).
|: Hello
S = $stream(0)
Codes = [72,101,108,108,111,10] ?

% yes
```

Errors

See get_code/2.

See also

None.

12.1.25 trim_blanks/2

Synopsis

```
trim_blanks(+Codes, ?TrimmedCodes)
```

Description

Removes all leading and trailing layout character codes in `Codes` and unifies the result with `TrimmedCodes`. Here a layout character code is defined by is_layout/1.

Examples

```
| ?- trim_blanks("  123 4", Trimmed).
Trimmed = [49,50,51,32,52] ?

% yes
```

Errors

See is_layout/1.

See also

is_layout/1.

Chapter 13

The statistics library

The predicates in this library implement common statistics functions. To use these predicates you will need to load the `statistics.fasl` file. One way to do this is with the following call: `ensure_loaded(runtime(statistics))`.

13.1 Predicates

13.1.1 chi_squared_cdf/3

Synopsis

```
chi_squared_cdf(+X, +N, -Answer)
```

Description

`Answer` is the probability that `Y=<X` where `Y` is a random variable distributed according to the χ^2 distribution with `N` degrees of freedom.

Examples

```
| ?- chi_squared_cdf(3, 3, Answer).
Answer = 0.608375 ?

% yes
```

Errors

`domain_error(not_less_than_one, N)` The argument `N` was less than `1`.

`domain_error(not_less_than_zero, X)` The argument `X` was less than `0`.

`instantiation_error` Either the argument `X` or the argument `N` was an uninstantiated variable.

`type_error(integer, N)` The argument `N` did not evaluate to an integer.

Also, as the arguments `X` and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.2 chi_squared_pdf/3

Synopsis

```
chi_squared_pdf(+X, +N, -Answer)
```

Description

`Answer` is the probability density at `Y=:=X` where `Y` is a random variable distributed according to the χ^2 distribution with `N` degrees of freedom.

Examples

```
| ?- chi_squared_pdf(0, 2, Answer).
Answer = 0.5 ?

% yes
```

Errors

`domain_error(not_less_than_one, N)` The argument `N` was less than `1`.

`domain_error(not_less_than_zero, X)` The argument `X` was less than `0`.

`instantiation_error` Either the argument `X` or the argument `N` was an uninstantiated variable.

`type_error(integer, N)` The argument `N` did not evaluate to an integer.

Also, as the arguments `X` and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.3 chi_squared_quantile/3

Synopsis

```
chi_squared_quantile(+X, +N, -Answer)
```

Description

`Answer` is the `X` quantile (`0<=X, X<=1`) of the χ^2 distribution with `N` degrees of freedom.

Examples

```
| ?- chi_squared_quantile(0.995, 10, A).
A = 25.188173 ?

% yes
| ?- chi_squared_quantile(0.005, 10, A).
A = 2.159888 ?
```

Errors

`domain_error(chi_squared_out_of_bounds, (X, N))` One of the following holds for the arguments `X` and `N`:

- `X<0.1, N<2`
- `X<0.01, N<3`

`domain_error(not_greater_than_one, X)` The argument `X` was greater than `1`.

`domain_error(not_less_than_one, N)` The argument `N` was less than `1`.

`domain_error(not_less_than_zero, X)` The argument `X` was less than `0`.

`instantiation_error` Either the argument `X` or the argument `N` was an uninstantiated variable.

`type_error(integer, N)` The argument `N` did not evaluate to an integer.

Also, as the arguments `X` and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.4 f_cdf/4

Synopsis

```
f_cdf(+X, +M, +N, -Answer)
```

Description

`Answer` is the probability that `Y=<X` where `Y` is a random variable distributed according to the F distribution with `M` (numerator) and `N` (denominator) degrees of freedom.

Examples

```
| ?- f_cdf(1, 5, 2, Answer).
Answer = 0.431201 ?

% yes
```

Errors

`domain_error(not_less_than_one, I)` Either the argument `M` or the argument `N` was less than `1`.

`domain_error(not_less_than_zero, X)` The argument `X` was less than `0`.

`instantiation_error` At least one of the arguments `X`, `M`, or `N` was an uninstantiated variable.

`type_error(integer, I)` Either the argument `M` or the argument `N` did not evaluate to an integer.

Also, as the arguments `X`, `M`, and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.5 f_pdf/4

Synopsis

```
f_pdf(+X, +M, +N, -Answer)
```

Description

`Answer` is the probability density at `Y=:=X` where `Y` is a random variable distributed according to the F distribution with `M` (numerator) and `N` (denominator) degrees of freedom.

Examples

```
| ?- f_pdf(1, 100, 100, Answer).
Answer = 1.989731 ?

% yes
| ?- f_pdf(3, 1, 1, Answer).
Answer = 0.045944 ?

% yes
```

Errors

`domain_error(not_less_than_one, I)` Either the argument `M` or the argument `N` was less than `1`.

`domain_error(not_less_than_zero, X)` The argument `X` was less than `0`.

`instantiation_error` At least one of the arguments `X`, `M`, or `N` was an uninstantiated variable.

`type_error(integer, I)` Either the argument `M` or the argument `N` did not evaluate to an integer.

Also, as the arguments `X`, `M`, and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.6 `f_quantile/4`

Synopsis

```
f_quantile(+P, +M, +N, -Zp)
```

Description

`Zp` is the `P` quantile (`0<=P, P<=1`) of the F distribution with `M` (numerator) and `N` (denominator) degrees of freedom.

Examples

```
| ?- f_quantile(0.9, 9, 9, Zp).
Zp = 2.440334 ?

% yes
```

Errors

`domain_error(not_greater_than_one, P)` The argument `P` was greater than `1`.

`domain_error(not_less_than_one, I)` Either the argument `M` or the argument `N` was less than `1`.

`domain_error(not_less_than_zero, P)` The argument `P` was less than `0`.

`instantiation_error` At least one of the arguments `P`, `M`, or `N` was an uninstantiated variable.

`type_error(integer, I)` Either the argument `M` or the argument `N` did not evaluate to an integer.

Also, as the arguments `P`, `M`, and `S` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.7 `normal_cdf/4`

Synopsis

```
normal_cdf(+X, +M, +S, -Answer)
```

Description

`Answer` is the probability that `Y=<X` where `Y` is a random variable distributed according to the normal distribution with mean `M` and variance `S`.

Examples

```
| ?- normal_cdf(0, 0, 1, Answer).
Answer = 0.5 ?

% yes
```

Errors

`domain_error(not_less_than_zero, S)` The argument `S` was less than `0`.

`instantiation_error` At least one of the arguments `X`, `M`, or `S` was an uninstantiated variable.

Also, as the arguments `X`, `M`, and `S` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.8 normal_pdf/4

Synopsis

```
normal_pdf(+X, +M, +S, -Answer)
```

Description

`Answer` is the probability density at `Y=:=X` where `Y` is a random variable distributed according to the normal distribution with mean `M` and variance `S`.

Examples

```
| ?- normal_pdf(0, 0, 1, Answer).
Answer = 0.398942 ?

% yes
| ?- normal_pdf(0, 0, 4, Answer).
Answer = 0.199471 ?

% yes
```

Errors

`domain_error(not_less_than_zero, S)` The argument `S` was less than `0`.

`instantiation_error` At least one of the arguments `X`, `M`, or `S` was an uninstantiated variable.

Also, as the arguments `X`, `M`, and `S` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.9 normal_quantile/4

Synopsis

```
normal_quantile(+P, +M, +S, -Zp)
```

Description

`Zp` is the `P` quantile (`0<=P, P<=1`) of the normal distribution with mean `M` and variance `S`.

Examples

```
| ?- normal_quantile(0.9, 70, 16, Zp).
Zp = 75.126915 ?

% yes
```

Errors

`domain_error(not_greater_than_one, P)` The argument `P` was greater than `1`.

`domain_error(not_less_than_zero, N)` Either the argument `P` or the argument `S` was less than `0`.

`instantiation_error` At least one of the arguments `X`, `M`, or `S` was an uninstantiated variable.

Also, as the arguments `P`, `M`, and `S` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.10 population_mean_confidence_interval/4

Synopsis

```
population_mean_confidence_interval(+L, +Alpha, -Low, -High)
```

Description

This predicate calculates confidence interval (`Low`, `High`) with significance level `Alpha` of the list of numbers `L`. This means that the population mean lies within (`Low`, `High`) with a probability of `1-Alpha`.

Examples

```
| ?- population_mean_confidence_interval([1,2,3,4,5,6,7,8,9,10],
                                         0.5, Low, High).
Low = 4.827195
High = 6.172805 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` Either the argument `L` or the argument `Alpha` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` and the argument `Alpha` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.11 `sample_absolute_deviation/3`

Synopsis

```
sample_absolute_deviation(+L, +C, -Result)
```

Description

`Result` is the arithmetic mean of the absolute deviations of the members of the list of numbers `L` from the value `C`. The value `C` should be some central value, e.g., the mean or median of `L`.

Examples

```
| ?- L = [1,2,3,4,5,6,7,8,9,10],
     sample_arithmetic_mean(L, Mean),
     sample_absolute_deviation(L, Mean, D).
L = [1,2,3,4,5,6,7,8,9,10]
Mean = 5.5
D = 2.5 ?

% yes
| ?- L = [1,2,3,4,5,6,7,8,9,10],
     sample_median(L, Median),
     sample_absolute_deviation(L, Median, D).
L = [1,2,3,4,5,6,7,8,9,10]
Median = 5.0
D = 2.5 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` Either the argument `L` or the the argument `C` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the argument `C` and the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_mean_absolute_deviation/2`, `sample_median_absolute_deviation/2`.

13.1.12 sample_arithmetic_mean/2

Synopsis

```
sample_arithmetic_mean(+L, -M)
```

Description

This predicates calculates the arithmetic mean `M` of a list of numbers `L`.

Examples

```
| ?- sample_arithmetic_mean([1,2,3,4,5,6,7,8,9,10], M).
M = 5.5 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.13 `sample_coefficient_of_variation/2`

Synopsis

```
sample_coefficient_of_variation(+L, -Result)
```

Description

`Result` is the sample coefficient of variation of the list of numbers `L`.

Examples

```
| ?- sample_coefficient_of_variation([1,1,1], COV).
COV = 0.0 ?

% yes
| ?- sample_coefficient_of_variation([1,2,3], COV).
COV = 0.5 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.14 sample_geometric_mean/2

Synopsis

```
sample_geometric_mean(+L, -M)
```

Description

This predicates calculates the geometric mean `M` of a list of numbers `L`.

Examples

```
| ?- sample_geometric_mean([2,8], M).
M = 4.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.15 sample_harmonic_mean/2

Synopsis

```
sample_harmonic_mean(+L, -M)
```

Description

This predicates calculates the harmonic mean `M` of a list of numbers `L`.

Examples

```
| ?- sample_harmonic_mean([1,2,4], M).
M = 1.714286 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.16 sample_interquartile_range/2

Synopsis

```
sample_interquartile_range(+L, -Result)
```

Description

`Result` is the sample interquartile range of the list of numbers `L`. This predicate behaves as if it were defined as:

```
sample_interquartile_range(L, Result) :-
        sample_quantile(L, 0.75, SeventyFive),
        sample_quantile(L, 0.25, TwentyFive),
        Result is (SeventyFive - TwentyFive).
```

Examples

```
| ?- sample_interquartile_range([1,2,3,4,5,6,7,8], R).
R = 4.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_semi_interquartile_range/2`.

13.1.17 sample_mean_absolute_deviation/2

Synopsis

```
sample_mean_absolute_deviation(+L, -Result)
```

Description

`Result` is the arithmetic mean of the absolute deviations of the members of the list of numbers `L` from the arithmetic mean of `L`. This predicate behaves as if it were defined as:

```
sample_mean_absolute_deviation(L, Result) :-
        sample_arithmetic_mean(L, Mean),
        sample_absolute_deviation(L, Mean, Result).
```

Examples

```
| ?- sample_mean_absolute_deviation([1,2,3], R).
R = 0.666667 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_absolute_deviation/3`.

13.1.18 sample_median/2

Synopsis

```
sample_median(+L, -Result)
```

Description

`Result` is the median of the list of numbers `L`. This predicate behaves as if it were defined as follows:

```
sample_median(L, Result) :-
        sample_quantile(L, 0.5, Result).
```

Examples

```
| ?- sample_median([1,2,3,4,5,6,7,8,9,10], Result).
Result = 5.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as all members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

sample_quantile/3.

13.1.19 sample_median_absolute_deviation/2

Synopsis

```
sample_median_absolute_deviation(+L, -Result)
```

Description

`Result` is the arithmetic mean of the absolute deviations of the members of the list of numbers `L` from the median of `L`. This predicate behaves as if it were defined as:

```
sample_mean_absolute_deviation(L, Result) :-
        sample_median(L, Median),
        sample_absolute_deviation(L, Median, Result).
```

Examples

```
| ?- sample_median_absolute_deviation([1,2,3], R).
R = 0.666667 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by eval/2.

See also

sample_absolute_deviation/3.

13.1.20 sample_quantile/3

Synopsis

```
sample_quantile(+L, +Q, -Result)
```

Description

`Result` is the $(1/\texttt{Q})^{\text{th}}$ quantile of the list of numbers `L`. This predicate behaves as if it were defined as follows:

```
sample_quantile(L, Q, Result) :-
        sample_quantile(L, Q, 0, 0, 1, 0, Result).
```

Examples

```
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5, Result).
Result = 5.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` Either the argument `L` or the argument `Q` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the argument `Q` and all members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_quantile/7`.

13.1.21 sample_quantile/7

Synopsis

```
sample_quantile(+L, +Q, +P1, +P2, +P3, +P4, -Result)
```

Description

`Result` is the $(1/\texttt{Q})^{\text{th}}$ quantile of the list of numbers `L`. The arguments `P1`, `P2`, `P3`, and, `P4` are parameters which control the computation of `Result`. The specification of that computation is:

$$S = sort(L)$$

$$x = P1 + (length(S) + P2) \cdot Q$$

$$Result = S_{\lfloor x \rfloor} + (S_{\lceil x \rceil} - S_{\lfloor x \rfloor})(P3 + P4 \cdot fractional\_part(x))$$

Popular values for the parameters are shown in the following table:

P1	P2	P3	P4	Name
0	0	1	0	Inverse CDF
0	0	0	1	Linear interpolation (California Dept. of Publ. Works)
1/2	0	0	0	Closest element
1/2	0	0	1	Linear interpolation (Hazen's, Hydrologist)
0	1	0	1	Mean estimate (Weibull)
1	-1	0	1	Mode estimate
3/8	1/4	0	1	Normal distribution estimate

Examples

```
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5,
                     0, 0, 1, 0,
                     InverseCDF).
InverseCDF = 5.0 ?

% yes
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5,
                     0, 0 ,0 , 1,
                     LinearInterpolationCalifornia).
LinearInterpolationCalifornia = 5.0 ?

% yes
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5,
                     0.5, 0, 0, 0,
                     ClosestElement).
ClosestElement = 5.0 ?

% yes
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5,
                     0.5, 0 , 0, 1,
                     LinearInterpolationHydrologist).
LinearInterpolationHydrologist = 5.5 ?

% yes
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5,
                     0, 1, 0, 1,
                     Wiebull).
Wiebull = 5.5 ?

% yes
```

```
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5,
                     1, -1, 0, 1,
                     ModeEstimate).
ModeEstimate = 5.5 ?

% yes
| ?- sample_quantile([1,2,3,4,5,6,7,8,9,10], 0.5,
                     3/8, 1/4, 0, 1,
                     NormalDistributionEstimate).
NormalDistributionEstimate = 5.5 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` At least one of the arguments `L`, `Q`, `P1`, `P2`, `P3`, or `P4` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as almost all of the arguments are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.22 `sample_semi_interquartile_range/2`

Synopsis

```
sample_semi_interquartile_range(+L, -Result)
```

Description

`Result` is the sample semi-interquartile range of the list of numbers `L`. This predicate behaves as if it were defined as:

```
sample_semi_interquartile_range(L, Result) :-
        sample_interquartile_range(L, InterQuartileRange),
        Result is InterQuartileRange / 2.
```

Examples

```
| ?- sample_semi_interquartile_range([1,2,3,4,5,6,7,8], R).
R = 2.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_interquartile_range/2`.

13.1.23 sample_standard_deviation/2

Synopsis

```
sample_standard_deviation(+L, -Result)
```

Description

`Result` is the sample standard deviation of the list of numbers `L`.

Examples

```
| ?- sample_standard_deviation([1,1,1], D).
D = 0.0 ?

% yes
| ?- sample_standard_deviation([1,2,3], D).
D = 1.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_standard_deviation/3`.

13.1.24 sample_standard_deviation/3

Synopsis

```
sample_standard_deviation(+L, +M, -Result)
```

Description

`Result` is the sample standard deviation of the list of numbers `L` which has an arithmetic mean of `M`. This predicate is provided for the cases where the mean of a sample is already known. In other cases, `sample_standard_deviation/2` should be used.

Examples

```
| ?- sample_standard_deviation([1,1,1], 1, D).
D = 0.0 ?

% yes
| ?- sample_standard_deviation([1,2,3], 2, D).
D = 1.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the argument `M` and the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_standard_deviation/2`.

13.1.25 sample_variance/2

Synopsis

```
sample_variance(+L, -Result)
```

Description

`Result` is the sample variance of the list of numbers `L`.

Examples

```
| ?- sample_variance([1,1,1], V).
V = 0.0 ?

| ?- sample_variance([1,2,3], V).
V = 1.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_variance/3`.

13.1.26 sample_variance/3

Synopsis

```
sample_variance(+L, +M, -Result)
```

Description

`Result` is the sample variance of the list of numbers `L` which has an arithmetic mean of `M`. This predicate is provided for the cases where the mean of a sample is already known. In other cases, `sample_variance/2` should be used.

Examples

```
| ?- sample_variance([1,1,1], 1, V).
V = 0.0 ?

% yes
| ?- sample_variance([1,2,3], 2, V).
V = 1.0 ?

% yes
```

Errors

`domain_error(non_empty_list, [])` The argument `L` was the empty list `[]`.

`instantiation_error` The argument `L` was an uninstantiated variable.

`type_error(list, L)` The argument `L` was not a list.

Also, as the members of the argument `L` are evaluated, errors may be thrown by `eval/2`.

See also

`sample_variance/2`.

13.1.27 `students_t_cdf/3`

Synopsis

```
students_t_cdf(+X, +N, -Answer)
```

Description

`Answer` is the probability that `Y=<X` where `Y` is a random variable distributed according to the Student's t distribution with `N` degrees of freedom.

Examples

```
| ?- students_t_cdf(-2, 1, Answer).
Answer = 0.147584 ?

% yes
```

Errors

`domain_error(not_less_than_one, N)` The argument `N` was less than `1`.

`instantiation_error` Either the argument `X` or the argument `N` was an uninstantiated variable.

`type_error(integer, N)` The argument `N` did not evaluate to an integer.

Also, as the arguments `X` and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.28 `students_t_pdf/3`

Synopsis

```
students_t_pdf(+X, +N, -Answer)
```

Description

`Answer` is the probability density at `Y=:=X` where `Y` is a random variable distributed according to the Student's t distribution with `N` degrees of freedom.

Examples

```
| ?- students_t_pdf(0, 1, Answer).
Answer = 0.318310 ?

% yes
```

Errors

`domain_error(not_less_than_one, N)` The argument `N` was less than `1`.

`instantiation_error` One the arguments `X` or `N` was an uninstantiated variable.

`type_error(integer, N)` The argument `N` did not evaluate to an integer.

Also, as the arguments `X` and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.29 `students_t_quantile/3`

Synopsis

```
students_t_quantile(+X, +N, -Answer)
```

Description

`Answer` is the `X` quantile (`0<=X, X<=1`) of the Student's t distribution with `N` degrees of freedom.

Examples

```
| ?- students_t_quantile(0.9, 9, Answer).
Answer = 1.383027 ?

% yes
```

Errors

`domain_error(not_less_than_one, N)` The argument `N` was less than `1`.

`domain_error(not_less_than_zero, X)` The argument `X` was less than `0`.

`domain_error(not_greater_than_one, X)` The argument `X` was greater than `1`.

`instantiation_error` Either the argument `X` or the argument `N` was an uninstantiated variable.

`type_error(integer, N)` The argument `N` did not evaluate to an integer.

Also, as the arguments `X` and `N` are evaluated, errors may be thrown by `eval/2`.

See also

None.

13.1.30 unpaired_t_test/5

Synopsis

```
unpaired_t_test(+A, +B, +Alpha, -Low, -High)
```

Description

The interval (`Low`, `High`) is the (`1-Alpha`)-percent confidence interval of the difference of the means of the list of numbers `A` and `B`. If `Low =< 0 <= High` then we cannot say with (`1-Alpha`)-percent confidence that the two samples are different.

Examples

Suppose we have measured the response times for a benchmark program on two different. Call these response times `T1` and `T2`. We can use `unpaired_t_test/5` to calculate the 95% confidence interval for the mean difference:

```
| ?- T1=[186, 181, 176, 149, 184, 190,
         158, 139, 175, 148, 152, 111,
         141, 153, 190, 157, 131, 149,
         135, 132],
|    T2=[129, 132, 102, 106, 94, 102,
         87, 99, 170, 113, 135, 142,
         86, 143, 152, 146, 144],
|    unpaired_t_test(T1, T2, 0.05, Low, High).
T1 = [186,181,176,149,184,190,158,139,
      175,148,152,111,141,153,190,157,
      131,149,135,132]
T2 = [129,132,102,106,94,102,87,99,170,
      113,135,142,86,143,152,146,144]
Low = 18.142755
High = 50.616069 ?
% yes
```

The result tells us with 95% confidence that the mean response time for the second program is between 18.143 and 50.616 time units faster than the mean response time for the first program.

Errors

`domain_error(non_empty_list, [])` Either the argument `A` or the argument `B` was the empty list `[]`.

`domain_error(not_less_than_zero, Alpha)` The argument `Alpha` was less than `0`.

`domain_error(not_greater_than_one, Alpha)` The argument `Alpha` was greater than `1`.

`instantiation_error` At least one the arguments `A`, `B` or `Alpha` was an uninstantiated variable.

`type_error(list, L)` Either the argument `A` or the argument `B` was not a list.

Also, as the arguments `A`, `B` and `Alpha` are evaluated, errors may be thrown by `eval/2`.

See also

None.

Chapter 14

Debug

Most Prolog debuggers are variations of the DECsystem-10 implementation [BBP+81] which traces Prolog execution by displaying messages at certain execution events corresponding to the ports of the Byrd Box model [Byr80]. The debugger described here is just another variation of this classic design, but instead of providing an inferior rewrite of the usual Byrd Box description, we'll try to reconstruct a simplified version from first principles. Hopefully this way the user will gain a useful insight into why the debugger is the way it is.

14.1 A Simplified Tracer

Imagine we were interested in examining the execution of the goal `a` given the following clauses:

```
a :- b, c ; d.
b :- true.
d :- true.
```

We'll assume that `c` is not defined which means that as a goal it will fail. However, from this simple program we can see that our original goal `a` should succeed since the sub-goal `d` succeeds. Let's actually demonstrate that this is true by giving our goal to the most basic of Prolog interpreters — `demo/1`.

```
demo(true) :- !.
demo((G1, G2)) :- !, debug(G1), debug(G2).
demo((G1; G2)) :- !, debug(G1); debug(G2).
demo(G) :- clause(G, C), demo(C).
```

When we feed our goal `a` as an argument to `demo/1` we see that it succeeds, so all is well. Whilst this basic interpreter is useful from a computational point of view, it gives us no useful information when debugging. What would help us is a trace of the interesting events which occur during execution.

An obvious step towards this goal could be an interpreter which traced any attempt to call a goal. We achieve this by augmenting `demo/1` with some output predicates and we call the result `trace1/1`.

```
trace1(true) :- !.
trace1((G1, G2)) :- !, trace1(G1), trace1(G2).
trace1((G1; G2)) :- !, (trace1(G1); trace1(G2)).
trace1(G) :- write(call), tab(1), write(G), nl,
             clause(G, C), trace1(C).
```

When we feed the goal `a` to `trace1/1` we get the following output

```
| ?- trace1(a).
call a
call b
call c
call d
% yes
```

This trace lets us see the Prolog interpreter in action but since we do not see what goals succeed and what goals fail, the trace can be difficult to use for all but the smallest programs. Another simple augmentation, this time of `trace1/1`, will give us a trace upon a successful exit.

```
trace2(true) :- !.
trace2((G1, G2)) :- !, trace2(G1), trace2(G2).
trace2((G1; G2)) :- !, (trace2(G1); trace2(G2)).
trace2(G) :- write(call), tab(1), write(G), nl,
             clause(G, C), trace2(C),
             write(exit), tab(1), write(G), nl.
```

When we feed the goal `a` to `trace2/1` we get the following output

```
| ?- trace2(a).
call a
call b
exit b
call c
call d
exit d
exit a
% yes
```

From a lack of an exit trace for `c`, we see that the goal `c` doesn't exit and this is due to `clause(c,C)` failing. A small change to `trace2/1` would allow us to trace such an event. Another interesting event linked to failure is the recalling of a goal due to the failure of subsequent goals. In the running

example the sub-goal `b` is called twice, the second calling being forced by the failure of goal `c`. This recalling will lead to failure as `clause(b,C)` succeeds only once. Again, a small change to `trace2/1` would allow us to trace recalling events. Let us call the failure of the current goal "fail" and the recalling of a goal due to backtracking "redo". We change `trace2/1` to give us `trace3/1` which will trace the events: call, exit, redo, and fail.

```
trace3(true) :- !.
trace3((G1, G2)) :- !, trace3(G1), trace3(G2).
trace3((G1; G2)) :- !, (trace3(G1); trace3(G2)).
trace3(G) :- pre(G), clause(G, C), trace3(C), post(G).

pre(G) :- write(call), tab(1), write(G), nl.
pre(G) :- write(fail), tab(1), write(G), nl, fail.

post(G) :- write(exit), tab(1), write(G), nl.
post(G) :- write(redo), tab(1), write(G), nl, fail.
```

When we feed the goal `a` to `trace3/1` we get the following output

```
| ?- trace3(a).
call a
call b
exit b
call c
fail c
redo b
fail b
call d
exit d
exit a
% yes
```

Here we can easily see how `c` fails which backtracks to `b` which in turn fails.

14.2 The Debugger

The provided debugger is essentially `trace/3` extended with the following features:

- When a goal exits prematurely due to an exception being thrown, we trace an "exception" event. In total we can trace call, exit, redo, fail, and exception events.
- Trace output is extended to include, amongst other things, a global invocation count and a call depth count.

- Trace events can be *leashed*. When execution reaches a leashed event a trace output is generated as usual but instead of continuing straight on to the next event, execution is paused and the user is prompted for a command which the debugger will execute before it continues. This is similar to the "step" function of other programming language debuggers.

- Tracing can be turned off and at a later stage automatically turned on when control reaches predicates which have *spypoints* placed on them. The debugger will then pause and await a command at every event connected to this predicate regardless of whether the event is leashed or not. A spypoint is similar to the "breakpoint" function of other programming language debuggers.

In order to accomplish all this, a degree of resource usage is required beyond that of the default interpreter, e.g., local stack usage is increased greatly. This should be borne in mind so as to avoid resource exhaustion when dealing with deeply nested calls or recursion.

14.2.1 Starting the debugger

The debugger is not a fundamental part of the system as the system can function quite happily without it. To bring the debugger into play you will need to load the file `debug.fasl`. One way of doing this is with the call: `ensure_loaded(runtime(debug))`. Once loaded the debugger's behaviour is configured with the Prolog flags `debug` and `trace`:

`debug` When set to `off` then the debugger is disengaged and — apart from the debugger taking up space in the clause store and the code store — the the operation of the system is no different from the case where the debugger has not been loaded. When the flag is set to `on` then the debugger is engaged and all necessary debug information is recorded during execution which leads to an increase in resource usage. It is possible to set the value of `debug` to `on` when the debugger is not loaded. This should be avoided as it offers no functional gain and Prolog code is executed more slowly.

`trace` The value of this flag only has an effect when the debugger is engaged. When the debugger is engaged and the flag `trace` is set to `off`, then no trace output is generated until a spypoint is reached. When the debugger is engaged and the flag `trace` is set to `on`, then all events are traced and should the traced event be leashed, then the execution is stopped until a debug command is given.

14.2.2 Trace output

Each trace output has six fields. A typical example is the following:

```
** (0) 0 call:  (dynamic) fact(3,_195168)?
```

The six fields are to be interpreted as follows:

`**`
: This trace relates to a spypoint. Another possibility is two spaces instead of two asterisks which indicate that this is not a spypoint.

`(0)`
: This is the global invocation count. It is incremented on each sub-goal call the debugger makes. Static predicates do not increment the counter when calling sub-goals. This counter is set to zero before the top-level loop tries to satisfy a query.

`0`
: This is the depth count and should be interpreted as a count of the number of dynamic parent goals the current goal has. This is set to zero upon two events: when the global invocation count is set to zero, and, every time a static predicate passes control to a dynamic predicate.

`call:`
: This is the trace event. It is either `call:`, `exit:`, `redo:`, `fail:`, or `exception:`

`(dynamic)`
: This is an indication of the representation of the current goal's predicate. It is either `(dynamic)` indicating interpreted code, or `(static)` indicating compiled code.

`fact(3,_195168)`
: This is the current goal. This field is written using `write_term/2` along with the value of the flag `debug_write_term_options`.

The question mark, at the end of the output, indicates that the system has stopped execution and is waiting for the user to input a debugging command. As a consequence, we know that either the goal predicate has a spypoint — as it is in this example — or this is a leashed event and we are tracing. No question mark is shown when the debugger continues to execute the traced event without asking for a debugging command.

As an example of tracing, suppose we had defined `fact/2` as follows:

```
fact(N,F) :- N =< 1, !, F=1.
fact(N,F) :- M is N-1, fact(M, E), F is N*E.
```

If we wanted to trace the execution of `fact(2,X)` with no user interaction we could perform the following sequence of actions:

1. Make sure no events are leashed to remove any need for input from the user.

2. Turn on debugging and tracing. We'll assume the debugger is actually loaded.

3. Give the query as we normally would do.

And we do it just like this:

```
| ?- leash(off).
% yes

| ?- trace.
% yes

| ?- fact(2,X).
   (0) 0 call:  (dynamic) fact(2,_195168)
   (1) 1 call:  (static) 2 =< 1
   (1) 1 fail:  (static) 2 =< 1
   (2) 1 call:  (static) _316624 is 2-1
   (2) 1 exit:  (static) 1 is 2-1
   (3) 1 call:  (dynamic) fact(1,_316784)
   (4) 2 call:  (static) 1 =< 1
   (4) 2 exit:  (static) 1 =< 1
   (5) 2 call:  (static) _316784 = 1
   (5) 2 exit:  (static) 1 = 1
   (3) 1 exit:  (dynamic) fact(1,1)
   (6) 1 call:  (static) _195168 is 2*1
   (6) 1 exit:  (static) 2 is 2*1
   (0) 0 exit:  (dynamic) fact(2,2)
X = 2
% yes
```

14.2.3 Debugging Commands

The following commands can be given by the user when the debugger stops at a either a spypoint or a leashed event:

a

Abort. The predicate `abort/0` is called. This sets the flags `debug` and `trace` to `off` and the debugger is exited.

c

Creep. The debugger continues execution until the next trace event. At each event a trace message is written and if the event is leashed or the goal has a spypoint, then execution will stop and the user will be asked for a new command. Creeping is the equivalent to single-stepping found in debuggers for other programming languages.

carriage return

Creep. See above.

d

Display goal. The current goal is passed to `display/1`.

f

Fail. Instead of continuing with normal execution, the predicate `fail/0` is called. This only makes sense at call events.

g

Ancestor goals. A list of the ancestor goals of the current goal are shown starting with the most recent ancestor then continuing back in time. Each goal is written using `write_term/2` along with the value of the flag `debug_write_term_options`. This list may not be complete because the debugger cannot trace ancestors inside static predicates. This means that the list which starts with the current goal's parent — should it exist — continues up to and *not* including the first ancestor which is a static goal.

l

Leap. The flag `trace` is set to `off` and execution continues. Trace messages will not be shown again until either the flag `trace` is set to `on`, or, execution progresses to a spypoint. Use this command to leap from spypoint to spypoint without any trace messages in-between.

n

No debug. The predicate `nodebug/0` is called and execution continues. The effect of this is to turn of debugging.

p

Print goal. The current goal is passed to `print/1`. This should be avoided if `print/1` calls dynamic predicates when portraying as this could end up in a recursive call to the debugger.

s

Skip goal. The flag `trace` is set to `off` for the duration of the current goal invocation. This command only has an effect at call events. Upon any other event, a skip is equivalent to a creep.

e

Exception. A prompt is displayed and a Prolog term is read. The read term is then passed to `throw/1`.

h

Help. The list of commands is displayed.

?

Help. See above.

w

Write goal. The current goal is passed to `writeq/1`.

@

Command. A prompt is displayed and a Prolog term is read; the flags `debug` and `trace` are both set to `off`; the term which was read is passed to `call/1`; finally, the flags `debug` and `trace` are both set to `on`. Note that if you gave the command `nodebug` then it would have no effect as debugging and tracing would be turned on upon return. If you want to turn off debugging then use the "no debug" command described above.

=

Debugging. The predicate `debugging/0` is called.

+

Spy. The current goal's predicate indicator is passed to `add_spypoint/1`.

−

Nospy. The current goal's predicate indicator is passed to `remove_spypoint/1`.

14.3 Debug Predicates

14.3.1 `add_spypoint/1`

Synopsis

```
add_spypoint(+PI)
```

Description

Places a spypoint on the predicate identified by the predicate indicator `PI`. It is possible to have a spypoint placed on a predicate before it is defined. The predicates that are marked for spying can be queried with the predicate `debugging/0`.

Examples

```
| ?- add_spypoint(fact/2).
% yes
| ?- debugging.
Debugging is off
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
```

Errors

`instantiation_error` The argument `PI` was not ground.

`type_error(predicate_indicator, PI)` The argument `PI` was not a valid predicate indicator.

See also

`debugging/0`, `spy/1`.

14.3.2 debug/0

Synopsis

`debug`

Description

Sets the flag `debug` to the value `on` and the flag `trace` to the value `off`.

Examples

```
| ?- debug.
% yes
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
```

Errors

None.

See also

`nodebug/0`, `notrace/0`, `trace/0`.

14.3.3 debugging/0

Synopsis

```
debugging
```

Description

Display the status of the debugging flags and a list of the spypoints which have been placed on predicates.

Examples

```
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
```

Errors

None.

See also

debug/0, spy/1, trace/0.

14.3.4 leash/1

Synopsis

```
leash(+Mode)
```

Description

Configures the debugger to stop upon certain trace events, prompt for a debugging command from the user, then execute the given command. These trace events correspond to the the debugger entering and leaving a predicate, namely on calling, successful exit, redo upon backtracking, complete failure, and upon the throwing of an exception. An event for which we configure the debugger to stop is said to be leashed.

If `Mode` is the empty list then no events are leashed. Otherwise, if `Mode` is a list of symbols where each symbol is one of `call`, `exit`, `redo`, `fail`, or `exception`, then the corresponding event is leashed. A more user-friendly mnemonic interface is offered where `Mode` can be one of atoms `all`, `tight`, `half`, or `off`. These modes are defined as follows:

all Equivalent to `leash([call, exit, redo, fail, exception])`.

tight Equivalent to `leash([call, redo, fail])`.

half Equivalent to `leash([call, redo])`.

off Equivalent to `leash([])`.

Examples

```
| ?- leash(all).
% yes
```

Errors

`domain_error(leash_mode, M)` Either the argument `Mode` was a list but contained an element `M` that was not a trace event symbol, or, the argument `Mode` was the symbol `M` which is not a valid port list mnemonic symbol.

`instantiation_error` The argument `Mode` was not instantiated.

See also

None.

14.3.5 `nodebug/0`

Synopsis

`nodebug`

Description

Sets both of the flags `debug` and `trace` to the value `off`.

Examples

```
| ?- nodebug.
% yes
| ?- debugging.
Debugging is off
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
```

Errors

None.

See also

debug/0, notrace/0, trace/0.

14.3.6 nospy/1

Synopsis

```
nospy(+PI)
```

Description

Equivalent to remove_spypoint(PI). This predicate is defined as a prefix operator so the argument does not need parentheses.

Examples

```
| ?- debugging.
Debugging is off
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
| ?- nospy fact/2.
% yes
| ?- debugging.
Debugging is off
Tracing is off
Spying on the following list of predicates []
% yes
```

Errors

See remove_spypoint/1.

See also

debugging/0, remove_spypoint/1.

14.3.7 nospyall/0

Synopsis

```
nospyall
```

Description

Removes any spypoints on any predicates, existing or not.

Examples

```
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
| ?- nospyall.
% yes
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates []
% yes
```

Errors

None.

See also

debugging/0, remove_spypoint/1, nospy/1.

14.3.8 notrace/0

Synopsis

```
notrace
```

Description

Equivalent to nodebug/0.

Examples

See nodebug/0.

Errors

None.

See also

nodebug/0.

14.3.9 remove_spypoint/1

Synopsis

```
remove_spypoint(+PI)
```

Description

Removes the spypoint from the predicate identified by the predicate indicator `PI`. It is possible to remove a nonexistent spypoint. It is even possible to remove the spypoint of a nonexistent predicate. The predicates that have spypoints can be queried with the predicate `debugging/0`.

Examples

```
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
| ?- remove_spypoint(fact/2).
% yes
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates []
% yes
```

Errors

`instantiation_error` The argument `PI` was not ground.

`type_error(predicate_indicator, PI)` The argument `PI` was not a valid predicate indicator.

See also

`debugging/0`, `nospy/1`.

14.3.10 spy/1

Synopsis

```
spy(+PI)
```

Description

Calls add_spypoint/1 with the predicate indicator PI then sets the flag debug to the value on. This predicate is defined as a prefix operator so the argument does not need parentheses.

Examples

```
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates []
% yes
| ?- spy fact/2.
% yes
| ?- spy fact/2.
% yes
| ?- debugging.
Debugging is on
Tracing is off
Spying on the following list of predicates [fact/2]
% yes
```

Errors

See add_spypoint/1.

See also

add_spypoint/1, debugging/0, nospy/1.

14.3.11 trace/0

Synopsis

```
trace
```

Description

Sets both of the flags debug and trace to the value on.

Examples

```
| ?- trace, true.
   (0) 0 call: (static) true ?

   (0) 0 exit: (static) true ?
```

```
% yes
```

Errors

None.

See also

debug/0, nodebug/0, notrace/0.

14.4 Debug Flags

14.4.1 debug

Description

A switch which engages or disengages the debugger.

Default Value

`off`

Possible Values

`on` If the debugger is loaded then it is engaged and it will start operation.

`off` Disengage the debugger.

14.4.2 debug_write_term_options

Description

A list of options used to configure the output of the current goal in trace output.

Default Value

`[quoted(true), numbervars(true)]`

Possible Values

Any possible legal option list that can be passed to write_term/3. You might want to avoid the option `portray(true)` as this could lead to the debugger calling dynamic predicates which could lead to a recursive call to the debugger.

14.4.3 `trace`

Description

A switch to control the debugger's trace output generation.

Default Value

`off`

Possible Values

`on` Turn on debugger trace outputs.

`off` Turn off debugger trace outputs.

Chapter 15

Compiling

The compiler is used to speed up the execution of Prolog procedures. Compiled (static) procedures cannot be analysed with the debugger, nor can existing compiled procedures be augmented by consulting or replaced by reconsulting. The user who wishes to compile has two options:

- Dynamic procedures in the clause store can be individually compiled. In this case they are removed from the clause store and placed into the code section. They then cease to be dynamic and become static instead. This method of compilation is useful during program development when the user is not entirely confident that a procedure is working properly. The user can follow a reconsult/test/fix cycle with dynamic versions of the procedure until the test is passed then the procedure can be compiled for greater efficiency.
- Entire Prolog source files can be compiled. The user supplies the input file name and the name of the file where the compiled result is to be written. This output file contains Prolog terms which install compiled procedures into the code section. These files of compiled procedures are typically known as `fasl` files. See absolute_file_name/3 for information on how the file extension `fasl` can be used by predicates.

15.1 Predicates

15.1.1 `compile_file/2`

Synopsis

```
compile_file(+SourceFile, +DestinationFile)
```

Description

Compiles the predicates in the file `SourceFile` and writes the compiled predicates to the file `DestinationFile`. The output file contains Prolog terms

and can be loaded with consult/1, reconsult/1, or ensure_loaded/1. Any compiled predicates loaded this way are static. Any predicate which is defined as dynamic with the dynamic compiler directive is not compiled but copied to the output file instead. This allows the programmer to mix static and dynamic predicates.

Examples

Imagine the contents of the file test.pl were the following:

```
:- dynamic foo/1.
foo(X) :- member(X, [1,2,3]).

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

The transcript below shows a simple compilation. Note how the file test.fasl is automatically consulted by the query [test].

```
| ?- compile_file('test.pl', 'test.fasl').
% Compiling
% test.pl
% Finished
% Writing :
% test.fasl
% Done.
% yes
| ?- [test].
% yes
| ?- foo(1).
% yes
| ?- bar(2).
% yes
| ?- procedure_property(foo/1, Prop).
Prop = dynamic ?

% yes
| ?- procedure_property(bar/1, Prop).
Prop = static ?

% yes
```

Errors

The compiler calls open/3 and read_term/3 both of which may throw errors.

See also

None.

15.1.2 `compile_procedure/1`

Synopsis

```
compile_procedure(+PI)
```

Description

Compiles the dynamic procedure indicated by the predicate indicator `PI`.

Examples

```
| ?- assert((foo(X) :- member(X, [1,2,3]))).
X = _548432 ?

% yes
| ?- foo(2).
% yes
| ?- procedure_property(foo/1, Prop).
Prop = dynamic ?

% yes
| ?- compile_procedure(foo/1).
% yes
| ?- foo(2).
% yes
| ?- procedure_property(foo/1, Prop).
Prop = static ?

% yes
```

Errors

`instantiation_error` The argument `PI` was not instantiated.

`type_error(dynamic_procedure, PI)` The argument `PI` did not indicate a dynamic procedure.

`type_error(predicate_indicator, PI)` The argument `PI` was not a predicate indicator.

See also

None.

Bibliography

[BBP+81] D. L. Bowen, L. Byrd, L. M. Pereira, F. C. N. Pereira, and D. H. D. Warren. PROLOG on the DECSystem-10 user's manual. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1981. 339

[Byr80] Lawrence Byrd. Understanding control flow of prolog programs. In *Proc. Workshop on Logic Programming*, 1980. 339

[DM93] Pierre Deransart and Jan Maluszynski. *A grammatical view of logic programming.* MIT Press, 1993. 217

[MTH+83] Yuji Matsumoto, Hozumi Tanaka, Hideki Hirakawa, Hideo Miyoshi, and Hideki Yasukawa. Bup: A bottom-up parser embedded in prolog. *New Generation Comput.*, 1(2):145–158, 1983. 237

[O'K90] Richard A. O'Keefe. *The craft of Prolog.* Logic programming. Cambridge, Mass. MIT Press, 1990. 224

Index