used[12].

profiler(*-Old, +New*)
Query or change the status of the profiler. The status is one of `off`, `plain` or `cummulative`. `plain` implies the time used by childs of a predicate is not added to the time of the predicate. For status `cummulative` the time of childs is added (except for recursive calls). Cummulative profiling implies the stack is scanned upto the top on each time slice to find all active predicates. This implies the overhead grows with the number of active frames on the stack. Cummulative profiling starts debuggin mode to disable tail recursion optimisation, which would otherwise remove the necessary parent environments. Switching status from `plain` to `cummulative` resets the profiler. Switching to and from status `off` does not reset the collected statistics, thus allowing to suspend profiling for certain parts of the program.

reset_profiler
Swiches the profiler to `off` and clears all collected statistics.

profile_count(*+Head, -Calls, -Promilage*)
Obtain profile statistics of the predicate specified by *Head*. *Head* is an atom for predicates with arity 0 or a term with the same name and arity as the predicate required (see `current_predicate/2`). *Calls* is unified with the number of 'calls' and 'redos' while the profiler was active. *Promilage* is unified with the relative number of counts the predicate was active (`cummulative`) or on top of the stack (`plain`). *Promilage* is an integer between 0 and 1000.

3.37 Memory Management

Note: `limit_stack/2` and `trim_stacks/0` have no effect on machines that do not offer dynamic stack expansion. On these machines these predicates simply succeed to improve portability.

garbage_collect
Invoke the global- and trail stack garbage collector. Normally the garbage collector is invoked automatically if necessary. Explicit invokation might be useful to reduce the need for garbage collections in time critical segments of the code. After the garbage collection `trim_stacks/0` is invoked to release the collected memory resources.

limit_stack(*+Key, +Kbytes*)
Limit one of the stack areas to the specified value. *Key* is one of `local`, `global` or `trail`. The limit is an integer, expressing the desired stack limit in K bytes. If the desired limit is smaller than the currently used value, the limit is set to the nearest legal value above the currently used value. If the desired value is larger than the maximum, the maximum is taken. Finally, if the desired value is either 0 or the atom `unlimited` the limit is set to its maximum. The maximum and initial limit is determined by the command line options `-L`, `-G` and `-T`.

trim_stacks
Release stack memory resources that are not in use at this moment, returning them to the operating system. Trim stack is a relatively cheap call. It can be used to release memory resources in a backtracking loop, where the iterations require typically seconds of execution time and very different, potentionally large, amounts of stack space. Such a loop should be written as follows:

[12] `show_profile/1` is defined in Prolog and takes a considerable amount of memory.

cputime	(User) cpu time since Prolog was started in seconds
inferences	Total number of passes via the call and redo ports since Prolog was started.
heapused	Bytes heap in use.
local	Allocated size of the local stack in bytes.
localused	Number of bytes in use on the local stack.
locallimit	Size to which the local stack is allowed to grow
global	Allocated size of the global stack in bytes.
globalused	Number of bytes in use on the global stack.
globallimit	Size to which the global stack is allowed to grow
trail	Allocated size of the trail stack in bytes.
trailused	Number of bytes in use on the trail stack.
traillimit	Size to which the trail stack is allowed to grow
atoms	Total number of defined atoms.
functors	Total number of defined name/arity pairs.
predicates	Total number of predicate definitions.
modules	Total number of module definitions.
externals	Total number of *external references* in all clauses.
codes	Total amount of byte codes in all clauses.

Table 3.2: Keys for `statistics/2`

3.36 Finding Performance Bottlenecks

SWI-Prolog offers a statistical program profiler similar to Unix prof(1) for C and some other languages. A profiler is used as an aid to find performance pigs in programs. It provides information on the time spent in the various Prolog predicates.

The profiler is based on the assumption that if we monitor the functions on the execution stack on time intervals not correlated to the program's execution the number of times we find a procedure on the environment stack is a measure of the time spent in this procedure. It is implemented by calling a procedure each time slice Prolog is active. This procedure scans the local stack and either just counts the procedure on top of this stack (`plain` profiling) or all procedures on the stack (`cummulative` profiling). To get accurate results each procedure one is interested in should have a reasonable number of counts. Typically a minute runtime will suffice to get a rough overview of the most expensive procedures.

profile(*+Goal, +Style, +Number*)
: Execute *Goal* just like `time/1`. Collect profiling statistics according to style (see `profiler/2`) and show the top *Number* procedures on the current output stream (see `show_profile/1`). The results are kept in the database until `reset_profiler/0` or `profile/3` is called and can be displayed again with `show_profile/1`. `profile/3` is the normal way to invoke the profiler. The predicates below are low-level predicates that can be used for special cases.

show_profile(*+Number*)
: Show the collected results of the profiler. Stops the profiler first to avoid interference from `show_profile/1`. It shows the top *Number* predicates according the percentage cpu-time

unknown(*-Old, +New*)
Unify *Old* with the current value of the unknown system flag. On success *New* will be used to specify the new value. *New* should be instantiated to either `fail` or `trace` and determines the interpreter's action when an undefined predicate which is not declared dynamic is encountered (see `dynamic/1`). `fail` implies the predicate just fails silently. `trace` implies the tracer is started. Default is `trace`. The unknown flag is local to each module and `unknown/2` is module transparent. Using it as a directive in a module file will only change the unknown flag for that module. Using the :/2 construct the behaviour on trapping an undefined predicate can be changed for any module. Note that if the unknown flag for a module equals `fail` the system will not call `exception/3` and will **not** try to resolve the predicate via the dynamic library system. The system will still try to import the predicate from the public module.

style_check(*+Spec*)
Set style checking options. *Spec* is either `+<option>`, `-<option>`, `?<option>` or a list of such options. `+<option>` sets a style checking option, `-<option>` clears it and `?<option>` succeeds or fails according to the current setting. `consult/1` and derivates resets the style checking options to their value before loading the file. If –for example– a file containing long atoms should be loaded the user can start the file with:

```
:- style_check(-atom).
```

Currently available options are:

Name	Default	Description
`singleton`	on	`read_clause/1` (used by `consult/1`) warns on variables only appearing once in a term (clause) which have a name not starting with an underscore.
`atom`	on	`read/1` and derivates will produce an error message on quoted atoms or strings longer than 5 lines.
`dollar`	off	Accept dollar as a lower case character, thus avoiding the need for quoting atoms with dollar signs. System maintenance use only.
`discontiguous`	on	Warn if the clauses for a predicate are not together in the same source file.
`string`	off	Read and derivates transform `"..."` into a prolog string instead of a list of ASCII characters.

3.35 Obtaining Runtime Statistics

statistics(*+Key, -Value*)
Unify system statistics determined by *Key* with *Value*. The possible keys are given in the table 3.2.

statistics
Display a table of system statistics on the current output stream.

time(*+Goal*)
Execute *Goal* just like `once/1` (i.e. leaving no choice points), but print used time, number of logical inferences and the average number of *lips* (logical inferences per second). Note that SWI-Prolog counts the actual executed number of inferences rather than the number of passes through the call- and redo ports of the theoretical 4-port model.

protocola(+*File*)
Equivalent to `protocol/1`, but does not truncate the *File* if it exists.

noprotocol
Stop making a protocol of the user interaction. Pending output is flushed on the file.

protocolling(-*File*)
Succeeds if a protocol was started with `protocol/1` or `protocola/1` and unifies *File* with the current protocol output file.

3.34 Debugging and Tracing Programs

trace
Start the tracer. `trace/0` itself cannot be seen in the tracer. Note that the Prolog toplevel treats `trace/0` special; it means 'trace the next goal'.

tracing
Succeeds when the tracer is currently switched on. `tracing/0` itself can not be seen in the tracer.

notrace
Stop the tracer. `notrace/0` itself cannot be seen in the tracer.

debug
Start debugger (stop at spy points).

nodebug
Stop debugger (do not trace, nor stop at spy points).

debugging
Print debug status and spy points on current output stream.

spy(+*Pred*)
Put a spy point on all predicates meeting the predicate specification *Pred*. See section 3.3.

nospy(+*Pred*)
Remove spy point from all predicates meeting the predicate specification *Pred*.

nospyall
Remove all spy points from the entire program.

leash(?*Ports*)
Set/query leashing (ports which allow for user interaction). *Ports* is one of `+Name`, `-Name`, `?Name` or a list of these. `+Name` enables leashing on that port, `-Name` disables it and `?Name` succeeds or fails according to the current setting. Recognised ports are: `call`, `redo`, `exit`, `fail` and `unify`. The special shorthand `all` refers to all ports, `full` refers to all ports except for the unify port (default). `half` refers to the `call`, `redo` and `fail` port.

visible(+*Ports*)
Set the ports shown by the debugger. See `leash/1` for a description of the port specification. Default is `full`.

absolute_file_name(*+File, -Absolute*)
Expand Unix file specification into an absolute path. User home directory expansion (`~` and `~user`) and variable expansion is done. The absolute path is canonised: references to '.' and '..' are deleted. SWI-Prolog uses absolute file names to register source files independant of the current working directory.

expand_file_name(*+WildChart, -List*)
Unify *List* with a sorted list of files or directories matching *WildChart*. The normal Unix wildchart constructs '`?`', '`*`', '`[...]`' and '`{...}`' are recognised. The interpretation of '`{...}`' is interpreted slightly different from the C shell (csh(1)). The comma separated argument can be arbitrary patterns, including '`{...}`' patterns. The empty pattern is legal as well: '`{.pl,}`' matches either '`.pl`' or the empty string.

chdir(*+Path*)
Change working directory to *Path*.

3.32 User Toplevel Manipulation

break
Recursively start a new Prolog top level. This Prolog top level has it's own stacks, but shares the heap with all break environments and the top level. Debugging is switched off on entering a break and restored on leaving one. The break environment is terminated by typing the system's end-of-file character (control-D). If the `-t toplevel` command line option is given this goal is started instead of entering the default interactive top level (`prolog/0`).

abort
Abort the Prolog execution and start a new top level. If the `-t toplevel` command line options is given this goal is started instead of entering the default interactive top level. Break environments are aborted as well. All open files except for the terminal related files are closed. The input- and output stream again refers to *user*.[10]

halt
Terminate Prolog execution. Open files are closed and if the command line option `-tty` is not active the terminal status (see Unix stty(1)) is restored.

prolog
This goal starts the default interactive top level. `prolog/0` is terminated (succeeds) by typing control-D.

3.33 Creating a Protocol of the Unser Interaction

SWI-Prolog offers the possibility to log the interaction with the user on a file.[11] All Prolog interaction, including warnings and tracer output, are written on the protocol file.

protocol(*+File*)
Start protocolling on file *File*. If there is already a protocol file open then close it first. If *File* exists it is truncated.

[10] BUG: Erased clauses which could not actually be removed from the database, because they are active in the interpreter, will never be garbage collected after an abort.
[11] A similar facility was added to Edinburgh C-Prolog by Wouter Jansweijer.

setenv(*+Name, +Value*)
Set Unix environment variable. *Name* and *Value* should be instantiated to atoms or integers. The environment variable will be passed to `shell/[0-2]` and can be requested using `getenv/2`.

unsetenv(*+Name*)
Remove Unix environment variable from the environment.

get_time(*-Time*)
Return the number of seconds that elapsed since the epoch of Unix, 1 January 1970, 0 hours. *Time* is a floating point number. Its granularity is system dependant. On SUN, this is 1/60 of a second.

convert_time(*+Time, -Year, -Month, -Day, -Hour, -Minute, -Second, -MilliSeconds*)
Convert a time stamp, provided by `get_time/1`, `file_time/2`, etc. *Year* is unified with the year, *Month* with the month number (January is 1), *Day* with the day of the month (starting with 1), *Hour* with the hour of the day (0–23), *Minute* with the minute (0–59). *Second* with the second (0–59) and *MilliSecond* with the milli seconds (0–999). Note that the latter might not be acurate or might always be 0, depending on the timing capabilities of the system.

3.31 Unix File System Interaction

access_file(*+File, +Mode*)
Succeeds when *File* exists and can be accessed by this prolog process under mode *Mode*. *Mode* is one of the atoms `read`, `write` or `execute`. *File* may also be the name of a directory. Fails silently otherwise.

exists_file(*+File*)
Succeeds when *File* exists. This does not imply the user has read and/or write permission for the file.

same_file(*+File1, +File2*)
Succeeds if both filenames refer to the same physical file. That is, if *File1* and *File2* are the same string or both names exist and point to the same file (due to hard or symbolic links and/or relative vs. absolute paths).

exists_directory(*+Directory*)
Succeeds if *Directory* exists. This does not imply the user has read, search and or write permission for the directory.

delete_file(*+File*)
Unlink *File* from the Unix file system.

rename_file(*+File1, +File2*)
Rename *File1* into *File2*. Currently files cannot be moved across devices.

size_file(*+File, -Size*)
Unify *Size* with the size of *File* in characters.

time_file(*+File, -Time*)
Unify the last modification time of *File* with *Time*. *Time* is a floating point number expressing the seconds ellapsed since Jan 1, 1970.

```
:- format_predicate(n, dos_newline(_Arg)).

dos_newline(Arg) :-
        between(1, Ar, _), put(13), put(10), fail ; true.
```

3.29 Terminal Control

The following predicates form a simple access mechanism to the Unix termcap library to provide terminal independant I/O for screen terminals. The library package `tty` builds on top of these predicates.

tty_get_capability(*+Name, +Type, -Result***)**
Get the capability named *Name* from the termcap library. See termcap(5) for the capability names. *Type* specifies the type of the expected result, and is one of `string`, `number` or `bool`. String results are returned as an atom, number result as an integer and bool results as the atom `on` or `off`. If an option cannot be found this predicate fails silently. The results are only computed once. Succesive queries on the same capability are fast.

tty_goto(*+X, +Y***)**
Goto position *(X, Y)* on the screen. Note that the predicates `line_count/2` and `line_position/2` will not have a well defined behaviour while using this predicate.

tty_put(*+Atom, +Lines***)**
Put an atom via the termcap library function tputs(). This function decodes padding information in the strings returned by `tty_get_capability/3` and should be used to output these strings. *Lines* is the number of lines affected by the operation, or 1 if not applicable (as in almost all cases).

set_tty(*-OldStream, +NewStream***)**
Set the output stream, used by `tty_put/2` and `tty_goto/2` to a specific stream. Default is user_output.

3.30 Unix Interaction

shell(*+Command, -Status***)**
Execute *Command* on the operating system. *Command* is given to the bourne shell (/bin/sh). *Status* is unified with the exit status of the command.

shell(*+Command***)**
Equivalent to '`shell(Command, 0)`'.

shell
Start an interactive Unix shell. Default is `/bin/sh`, the environment variable `SHELL` overrides this default.

getenv(*+Name, -Value***)**
Get Unix environment variable (see csh(1) and sh(1)). Fails if the variable does not exist.

| Set a tabstop on the current position. If an argument is supplied set a tabstop on the position of that argument. This will cause all `~t`'s to be distributed between the previous and this tabstop.

+ Set a tabstop relative to the current position. Further the same as `~|`.

w Give the next argument to `write/1`.

Example:

```
simple_statistics :-
    <obtain statistics>         % left to the user
    format('~tStatistics~t~72|~n~n'),
    format('Runtime: ~`.t ~2f~34|  Inferences: ~`.t ~D~72|~n',
                                          [RunT, Inf]),
    ....
```

Will output

```
                              Statistics

Runtime: .................. 3.45  Inferences: .......... 60,345
```

sformat(-*String, +Format, +Arguments*)
Equivalent to `format/3`, but "writes" the result on *String* instead of the current output stream. Example:

```
?- sformat(S, '~w~t~15|~w', ['Hello', 'World']).

S = "Hello          World"
```

sformat(-*String, +Format*)
Equivalent to '`sformat(String, Format, []).`'

3.28.3 Programming Format

format_predicate(+*Char, +Head*)
If a sequence `~c` (tilde, followed by some character) is found, the format derivates will first check whether the user has defined a predicate to handle the format. If not, the built in formatting rules described above are used. *Char* is either an ascii value, or a one character atom, specifying the letter to be (re)defined. *Head* is a term, whose name and arity are used to determine the predicate to call for the redefined formatting character. The first argument to the predicate is the numeric argument of the format command, or the atom `default` if no argument is specified. The remaining arguments are filled from the argument list. The example below redefines `~n` to produce *Arg* times return followed by linefeed (so a (Grr.) DOS machine is happy with the output).

required by the format specification. If only one argument is required and this is not a list of ASCII values the argument need not be put in a list. Otherwise the arguments are put in a list.

Special sequences start with the tilde (`~`), followed by an optional numeric argument, followed by a character describing the action to be undertaken. A numeric argument is either a sequence of digits, representing a positive decimal number, a sequence `'<character>`, representing the ASCII value of the character (only useful for `~t`) or a asterisk (*), in when the numeric argument is taken from the next argument of the argument list, which should be a positive integer. Actions are:

~ Output the tilde itself.

a Output the next argument, which should be an atom. This option is equivalent to **w**. Compatibility reasons only.

c Output the next argument as an ASCII value. This argument should be an integer in the range [0, ..., 255] (including 0 and 255).

d Output next argument as a decimal number. It should be an integer. If a numeric argument is specified a dot is inserted *argument* positions from the right (useful for doing fixed point arithmetic with integers, such as handling amounts of money).

D Same as **d**, but makes large values easier to read by inserting a comma every three digits left to the dot or right.

e Output next argument as a floating point number in exponentional notation. The numeric argument specifies the precission. Default is 6 digits. Exact representation depends on the C library function printf(). This function is invoked with the format `%.<percission>e`.

E Equivalent to **e**, but outputs a capital E to indicate the exponent.

f Floating point in non-exponentional notation. See C library function printf().

g Floating point in **e** or **f** notation, whichever is shorter.

G Floating point in **E** or **f** notation, whichever is shorter.

i Ignore next argument of the argument list. Produces no output.

k Give the next argument to `displayq/1` (canonical write).

n Output a newline character.

N Only output a newline if the last character output on this stream was not a newline. Not properly implemented yet.

p Give the next argument to `print/1`.

q Give the next argument to `writeq/1`.

r Print integer in radix the numeric argument notation. Thus `~16r` prints its argument hexadecimal. The argument should be in the range [2, ... 36]. Lower case letters are used for digits above 9.

R Same as **r**, but uses upper case letters for digits above 9.

s Output a string of ASCII characters from the next argument.

t All remaining space between 2 tabstops is distributed equaly over `~t` statements between the tabstops. This space is padded with spaces by default. If an argument is supplied this is taken to be the ASCII value of the character used for padding. This can be used to do left or right alignment, centering, distributing, etc. See also `~|` and `~+` to set tab stops. A tabstop is assumed at the start of each line.

\n	<NL> is output
\l	<LF> is output
\r	<CR> is output
\t	<TAB> is output
\\	The character '\' is output
\%	The character '%' is output
\nnn	where nnn is an integer (1-3 digits) the character with ASCII code nnn is output (NB : nnn is read as DECIMAL)

Escape sequences to include arguments from *Arguments*. Each time a % escape sequence is found in *Format* the next argument from *Arguments* is formatted according to the specification.

%t	print/1 the next item (mnemonic: term)
%w	write/1 the next item
%q	writeq/1 the next item
%d	display/1 the next item
%p	print/1 the next item (identical to %t)
%n	put the next item as a character (i.e. it is an ASCII value)
%r	write the next item N times where N is the second item (an integer)
%s	write the next item as a String (so it must be a list of characters)
%f	perform a ttyflush/0 (no items used)
%Nc	write the next item Centered in N columns.
%Nl	write the next item Left justified in N columns.
%Nr	write the next item Right justified in N columns. N is a decimal number with at least one digit. The item must be an atom, integer, float or string.

swritef(-*String, +Format, +Arguments*)
Equivalent to writef/3, but "writes" the result on *String* instead of the current output stream. Example:

```
?- swritef(S, '%15L%w', ['Hello', 'World']).

S = "Hello          World"
```

swritef(-*String, +Format*)
Equivalent to swritef(String, Format, []).

3.28.2 Format

format(+*Format*)
Defined as 'format(Format) :- format(Format, []).'

format(+*Format, +Arguments*)
Format is an atom, list of ASCII values, or a Prolog string. *Arguments* provides the arguments

checklist(*+Pred, +List*)
Pred is applied successively on each element of *List* until the end of the list or *Pred* fails. In the latter case the `checklist/2` fails.

maplist(*+Pred, ?List1, ?List2*)
Apply *Pred* on all successive pairs of elements from *List1* and *List2*. Fails if *Pred* can not be applied to a pair. See the example above.

sublist(*+Pred, +List1, ?List2*)
Unify *List2* with a list of all elements of *List1* to which *Pred* applies.

3.27 Forall

forall(*+Cond, +Action*)
For all alternative bindings of *Cond Action* can be proven. The example verifies that all arithmic statements in the list *L* are correct. It does not say which is wrong if one proves wrong.

```
?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 * 2]),
                  Result =:= Formula).
```

3.28 Formatted Write

The current version of SWI-Prolog provides two formatted write predicates. The first is `writef/[1,2]`, which is compatible with Edinburgh C-Prolog. The second is `format/[1,2]`, which is compatible with Quintus Prolog. We hope the Prolog community will once define a standard formatted write predicate. If you want performance use `format/[1,2]` as this predicate is defined in C. Otherwise compatibility reasons might tell you which predicate to use.

3.28.1 Writef

write_ln(*+Term*)
Equivalent to `write(Term), nl.`

writef(*+Atom*)
Equivalent to `writef(Atom, []).`

writef(*+Format, +Arguments*)
Formatted write. *Format* is an atom whose characters will be printed. *Format* may contain certain special character sequences which specify certain formatting and substitution actions. *Arguments* then provides all the terms required to be output.

Escape sequences to generate a single special character:

bagof(*+Var, +Goal, -Bag***)**

Unify *Bag* with the alternatives of *Var*, if *Goal* has free variables besides the one sharing with *Var* bagof will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Var*. The construct `Var^Goal` tells bagof not to bind *Var* in *Goal*. `Bagof/3` fails if *Goal* has no solutions.

The example below illustrates `bagof/3` and the `^` operator. The variable bindings are printed together on one line to save paper.

```
2 ?- listing(foo).

foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).

Yes
3 ?- bagof(C, foo(A, B, C), Cs).

A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g] ;

No
4 ?- bagof(C, A^foo(A, B, C), Cs).

A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g] ;

No
5 ?-
```

setof(*+Var, +Goal, -Set***)**

Equivalent to `bagof/3`, but sorts the result using `sort/2` to get a sorted list of alternatives without duplicates.

3.26 Invoking Predicates on all Members of a List

All the predicates in this section call a predicate on all members of a list or until the predicate called fails. The predicate is called via `apply/2`, which implies common arguments can be put in front of the arguments obtained from the list(s). For example:

```
?- maplist(plus(1), [0, 1, 2], X).

X = [1, 2, 3]
```

we will phrase this as "*Predicate* is applied on ..."

list_to_set(*+List, -Set***)**
Succeeds if *Set* holds the same elements as *List* in the same order, but has no duplicates. See also `sort/1`.

intersection(*+Set1, +Set2, -Set3***)**
Succeeds if *Set3* unifies with the intersection of *Set1* and *Set2*. *Set1* and *Set2* are lists without duplicates. They need not be ordered.

subtract(*+Set, +Delete, -Result***)**
Delete all elements of set 'Delete' from 'Set' and unify the resulting set with 'Result'.

union(*+Set1, +Set2, -Set3***)**
Succeeds if *Set3* unifies with the union of *Set1* and *Set2*. *Set1* and *Set2* are lists without duplicates. They need not be ordered.

subset(*+Subset, +Set***)**
Succeeds if all elements of *Subset* are elements of *Set* as well.

merge_set(*+Set1, +Set2, -Set3***)**
Set1 and *Set2* are lists without duplicates, sorted to the standard order of terms. *Set3* is unified with an ordered list without duplicates holding the union of the elements of *Set1* and *Set2*.

3.24 Sorting Lists

sort(*+List, -Sorted***)**
Succeeds if *Sorted* can be unified with a list holding the elements of *List*, sorted to the standard order of terms (see section 3.5). Duplicates are removed.

msort(*+List, -Sorted***)**
Equivalent to `sort/2`, but does not remove duplicates.

keysort(*+List, -Sorted***)**
List is a list of `Key-Value` pairs (e.g. terms of the functor '`-`' with arity 2). `keysort/2` sorts *List* like `msort/2`, but only compares the keys. Can be used to sort terms not on standard order, but on any criterion that can be expressed on a multi-dimensional scale. Sorting on more than one criterion can be done using terms as keys, putting the first criterion as argument 1, the second as argument 2, etc.

predsort(*+Pred, +List, -Sorted***)**
Sorts similar to `msort/2`, but determines the order of two terms by applying *Pred* to pairs of elements from *List* (see `apply/2`). The predicate should succeed if the first element should be before the second.

3.25 Finding all Solutions to a Goal

findall(*+Var, +Goal, -Bag***)**
Creates a list of the instantiations *Var* gets successively on backtracking over *Goal* and unifies the result with *Bag*. Succeeds with an empty list if *Goal* has no solutions. `findall/3` is equivalent to `bagof/3` with all free variables bound with the existence operator (`^`), except that `bagof/3` fails when goal has no solutions.

append(*?List1, ?List2, ?List3*)
Succeeds when *List3* unifies with the concatenation of *List1* and *List2*. The predicate can be used with any instantiation pattern (even three variables).

member(*?Elem, ?List*)
Succeeds when *Elem* can be unified with one of the members of *List*. The predicate can be used with any instantiation pattern.

memberchk(*?Elem, +List*)
Equivalent to `member/2`, but leaves no choice point.

delete(*+List1, ?Elem, ?List2*)
Delete all members of *List1* that simultaneously unify with *Elem* and unify the result with *List2*.

select(*?List1, ?Elem, ?List2*)
Select an element of *List1* that unifies with *Elem*. *List2* is unified with the list remaining from *List1* after deleting the selected element. Normally used with the instantiation pattern *+List1, -Elem, -List2*, but can also be used to insert an element in a list using *-List1, +Elem, +List2*.

nth0(*?Index, ?List, ?Elem*)
Succeeds when the *Index*-th element of *List* unifies with *Elem*. Counting starts at 0.

nth1(*?Index, ?List, ?Elem*)
Succeeds when the *Index*-th element of *List* unifies with *Elem*. Counting starts at 1.

last(*?Elem, ?List*)
Succeeds if *Elem* unifies with the last element of *List*.

reverse(*+List1, -List2*)
Reverse the order of the elements in *List1* and unify the result with the elements of *List2*.

flatten(*+List1, -List2*)
Transform *List1*, possibly holding lists as elements into a 'flat' list by replacing each list with its elements (recursively). Unify the resulting flat list with *List2*. Example:

```
?- flatten([a, [b, [c, d], e]], X).

X = [a, b, c, d, e]
```

length(*?List, ?Int*)
Succeeds if *Int* represents the number of elements of list *List*. Can be used to create a list holding only variables.

merge(*+List1, +List2, -List3*)
List1 and *List2* are lists, sorted to the standard order of terms (see section 3.5). *List3* will be unified with an ordered list holding the both the elements of *List1* and *List2*. Duplicates are **not** removed.

3.23 Set Manipulation

is_set(*+Set*)
Succeeds if set is a proper list (see `proper_list/1`) without duplicates.

cputime
Evaluates to a floating point number expressing the cpu time (in seconds) used by Prolog up till now. See also `statistics/2` and `time/1`.

3.21 Adding Arithmetic Functions

Prolog predicates can be given the role of arithmetic function. The last argument is used to return the result, the arguments before the last are the inputs. Arithmetic functions are added using the predicate `arithmetic_function/1`, which takes the head as its argument. Arithmetic functions are module sensitive, that is they are only visible from the module in which the function is defined and delared. Global arithmetic functions should be defined and registered from module `user`. Global definitions can be overruled locally in modules. The builtin functions described above can be redefined as well.

arithmetic_function(*+Head***)**
Register a Prolog predicate as an arithmetic function (see `is/2`, `>/2`, etc.). The Prolog predicate should have one more argument than specified by *Head*, which it either a term *Name/Arity*, an atom or a complex term. This last argument is an unbound variable at call time and should be instantiated to an integer or floating point number. The other arguments are the parameters. This predicate is module sensitive and will declare the arithmetic function only for the context module, unless declared from module `user`. Example:

```
1 ?- [user].
:- arithmetic_function(mean/2).

mean(A, B, C) :-
        C is (A+B)/2.
user compiled, 0.07 sec, 440 bytes.

Yes
2 ?- A is mean(4, 5).

A = 4.500000
```

current_arithmetic_function(*?Head***)**
Succesively unifies all arithmetic functions that are visible from the context module with *Head*.

3.22 List Manipulation

is_list(*+Term***)**
Succeeds if *Term* is bound to the empty list (`[]`) or a term with functor '`.`' and arity 2.

proper_list(*+Term***)**
Equivalent to `is_list/1`, but also requires the tail of the list to be a list (recursively). Examples:

```
is_list([x|A])          % true
proper_list([x|A])      % false
```

+IntExpr << *+IntExpr*
Bitwise shift *IntExpr1* by *IntExpr2* bits to the left.

+IntExpr \/ *+IntExpr*
Bitwise 'or' *IntExpr1* and *IntExpr2*.

+IntExpr /\ *+IntExpr*
Bitwise 'and' *IntExpr1* and *IntExpr2*.

+IntExpr **xor** *+IntExpr*
Bitwise 'exclusive or' *IntExpr1* and *IntExpr2*.

\ *+IntExpr*
Bitwise negation.

sqrt(*+Expr*)
Result = $\sqrt{Expr}$

sin(*+Expr*)
Result = sin *Expr*. *Expr* is the angle in radials.

cos(*+Expr*)
Result = cos *Expr*. *Expr* is the angle in radials.

tan(*+Expr*)
Result = tan *Expr*. *Expr* is the angle in radials.

asin(*+Expr*)
Result = arcsin *Expr*. *Result* is the angle in radials.

acos(*+Expr*)
Result = arccos *Expr*. *Result* is the angle in radials.

atan(*+Expr*)
Result = arctan *Expr*. *Result* is the angle in radials.

atan(*+YExpr, +XExpr*)
Result = arctan $\frac{YExpr}{\mathrm{XExpr}}$. *Result* is the angle in radials. The return value is in the range $-\pi ... \pi$. Used to convert between rectangular and polar coordinate system.

log(*+Expr*)
Result = ln *Expr*

log10(*+Expr*)
Result = lg *Expr*

exp(*+Expr*)
Result = e^{Expr}

+Expr1 ^ *+Expr2*
Result = $Expr1^{Expr2}$

pi
Evaluates to the mathematical constant π (3.141593...).

e
Evaluates to the mathematical constant e (2.718282...).

In case integer addition, subtraction and multiplication would lead to an integer overflow the operands are automatically converted to floating point numbers. The floating point functions (`sin/1`, `exp/1`, etc.) form a direct interface to the corresponding C library functions used to compile SWI-Prolog. Please refer to the C library documentation for details on percision, error handling, etc.

`-` *+Expr*
Result $= -Expr$

+Expr1 `+` *+Expr2*
Result $= Expr1 + Expr2$

+Expr1 `-` *+Expr2*
Result $= Expr1 - Expr2$

+Expr1 `*` *+Expr2*
Result $= Expr1 \times Expr2$

+Expr1 `/` *+Expr2*
Result $= \frac{Expr1}{\mathrm{Expr2}}$

+IntExpr1 **mod** *+IntExpr2*
Result $= Expr1\ mod\ Expr2$ (remainder of division).

+IntExpr1 `//` *+IntExpr2*
Result $= Expr1\ div\ Expr2$ (integer division).

abs(*+Expr*)
Evaluate *Expr* and return the absolute value of it.

max(*+Expr1, +Expr2*)
Evaluates to the largest of both *Expr1* and *Expr2*.

min(*+Expr1, +Expr2*)
Evaluates to the smallest of both *Expr1* and *Expr2*.

.(*+Int, []*)
A list of one element evaluates to the element. This implies `"a"` evaluates to the ASCII value of the letter `a` (97). This option is available for compatibility only. It will not work if 'style_check(+string)' is active as `"a"` will then be tranformed into a string object. The recommended way to specify the ASCII value of the letter 'a' is `0'a`.

random(*+Int*)
Evaluates to a random integer i for which $0 \leq i < Int$. The seed of this random generator is determined by the system clock when SWI-Prolog was started.

integer(*+Expr*)
Evaluates *Expr* and rounds the result to the nearest integer.

floor(*+Expr*)
Evaluates *Expr* and returns the largest integer smaller or equal to the result of the evaluation.

ceil(*+Expr*)
Evaluates *Expr* and returns the smallest integer larger or equal to the result of the evaluation.

+IntExpr `>>` *+IntExpr*
Bitwise shift *IntExpr1* by *IntExpr2* bits to the right. Note that integers are only 27 bits.

The general arithmic predicates are optionaly compiled now (see `please/3` and the `-O` command line option). Compiled arithmetic reduces global stack requirements and improves performance. Unfortunately compiled arithmetic cannot be traced, which is why it is optional.

The general arithmetic predicates all handle *expressions*. An expression is either a simple number or a *function*. The arguments of a function are expressions. The functions are described in section 3.20.

between(*+Low, +High, ?Value*)
: *Low* and *High* are integers, $High \geq Low$. If *Value* is an integer, $Low \leq Value \leq High$. When *Value* is a variable it is successively bound to all integers between *Low* and *High*.

succ(*?Int1, ?Int2*)
: Succeeds if $Int2 = Int1 + 1$. At least one of the arguments must be instantiated to an integer.

plus(*?Int1, ?Int2, ?Int3*)
: Succeeds if $Int3 = Int1 + Int2$. At least two of the three arguments must be instantiated to integers.

+Expr1 `>` *+Expr2*
: Succeeds when expression *Expr1* evaluates to a larger number than *Expr2*.

+Expr1 `<` *+Expr2*
: Succeeds when expression *Expr1* evaluates to a smaller number than *Expr2*.

+Expr1 `=<` *+Expr2*
: Succeeds when expression *Expr1* evaluates to a smaller or equal number to *Expr2*.

+Expr1 `>=` *+Expr2*
: Succeeds when expression *Expr1* evaluates to a larger or equal number to *Expr2*.

+Expr1 `=\=` *+Expr2*
: Succeeds when expression *Expr1* evaluates to a number non-equal to *Expr2*.

+Expr1 `=:=` *+Expr2*
: Succeeds when expression *Expr1* evaluates to a number equal to *Expr2*.

-Number **is** *+Expr*
: Succeeds when *Number* has successfully been unified with the number *Expr* evaluates to.

3.20 Arithmetic Functions

Arithmetic functions are terms which are evaluated by the arithmetic predicates described above. SWI-Prolog tries to hide the difference between integer arithmetic and floating point arithmetic from the Prolog user. Arithmetic is done as integer arithmetic as long as possible and converted to floating point arithmetic whenever one of the arguments or the combination of them requires it. If a function returns a floating point value which is whole it is automatically transformed into an integer. There are three types of arguments to functions:

Expr	Arbitrary expression, returning either a floating point value or an integer.
IntExpr	Arbitrary expression that should evaluate into an integer.
Int	An integer.

3.18 Operators

op(*+Precedence, +Type, +Name***)**

Declare *Name* to be an operator of type *Type* with precedence *Precedence*. *Name* can also be a list of names, in which case all elements of the list are declared to be identical operators. *Precedence* is an integer between 0 and 1200. Precedence 0 removes the declaration. *Type* is one of: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy` or `fx`. The '`f`' indicates the position of the functor, while `x` and `y` indicate the position of the arguments. '`y`' should be interpreted as "on this position a term with precedence lower or equal to the precedence of the functor should occur". For '`x`' the precedence of the argument must be strictly lower. The precedence of a term is 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator. A term enclosed in brackets (`(...)`) has precedence 0.

The predefined operators are shown in table 3.1. Note that all operators can be redefined by the user.

1200	xfx	`-->`, `:-`
1200	fx	`:-`, `?-`
1150	fx	`dynamic`, `multifile`, `module_transparent`, `discontiguous`
1100	xfy	`;`, `|`
1050	xfy	`->`
1000	xfy	`,`
954	xfy	`\\`
900	fy	`\+`, `not`
700	xfx	`<`, `=`, `=..`, `=@=`, `=:=`, `=<`, `==`, `=\=`, `>`, `>=`, `@<`, `@=<`, `@>`, `@>=`, `\=`, `\==`, `is`
600	xfy	`:`
500	yfx	`+`, `-`, `/\`, `\/`, `xor`
500	fx	`+`, `-`, `?`, `\`
400	yfx	`*`, `/`, `//`, `<<`, `>>`
300	xfx	`mod`
200	xfy	`^`

Table 3.1: System operators

current_op(*?Precedence, ?Type, ?Name***)**

Succeeds when *Name* is currently defined as an operator of type *Type* with precedence *Precedence*. See also `op/3`.

3.19 Arithmetic

Arithmetic can be divided into some special purpose integer predicates and a series of general predicates for floating point and integer arithmetic as appropriate. The integer predicates are as "logical" as possible. Their usage is recommended whenever applicable, resulting in faster and more "logical" programs.

atom_to_term(*+Atom, -Term, -Bindings*)
Use *Atom* as input to `read_variables/2` and return the read term in *Term* and the variable bindings in *Bindings*. *Bindings* is a list of *Name = Var* couples, thus providing access to the actual variable names. See also `read_variables/2`.

concat(*?Atom1, ?Atom2, ?Atom3*)
Atom3 forms the concatenation of *Atom1* and *Atom2*. At least two of the arguments must be instantiated to atoms, intergers or floating point numbers.

concat_atom(*+List, -Atom*)
List is a list of atoms, integers or floating point numbers. Succeeds if *Atom* can be unified with the concatenated elements of *List*. If *List* has exactly 2 elements it is equivalent to `concat/3`, allowing for variables in the list.

atom_length(*+Atom, -Length*)
Succeeds if *Atom* is an atom of *Length* characters long. This predicate also works for integers and floats, expressing the number of characters output when given to `write/1`.

3.17 Representing Text in Strings

SWI-Prolog supports the data type *string*. Strings are a time and space efficient mechanism to handle text in Prolog. Atoms are under some circumstances not suitable because garbage collection on them is next to impossible (Although it is possible: BIM_prolog does it). Representing text as a list of ASCII values is, from the logical point of view, the cleanest solution. It however has two drawbacks: 1) they cannot be distinguished from a list of (small) integers; and 2) they consume (in SWI-Prolog) 12 bytes for each character stored.

Within strings each character only requires 1 byte storage. Strings live on the global stack and their storage is thus reclaimed on backtracking. Garbage collection can easily deal with strings.

The ISO standard proposes `"..."` is transformed into a string object by `read/1` and derivates. This poses problems as in the old convention `"..."` is transformed into a list of ascii characters. For this reason the style check option '*string*' is available (see `style_check/2`).

The set of predicates associated with strings is incomplete and tentative. Names and definitions might change in the future to confirm to the emerging standard.

string_to_atom(*?String, ?Atom*)
Logical conversion between a string and an atom. At least one of the two arguments must be instantiated. *Atom* can also be an integer or floating point number.

string_to_list(*?String, ?List*)
Logical conversion between a string and a list of ASCII characters. At least one of the two arguments must be instantiated.

string_length(*+String, -Length*)
Unify *Length* with the number of characters in *String*. This predicate is functonally equivalent to `atom_length/2` and also accepts atoms, integers and floats as its first argument.

substring(*+String, +Start, +Length, -Sub*)
Create a substring of *String* that starts at character *Start* (1 base) and has *Length* characters. Unify this substring with *Sub*.[9]

[9] Future versions probably will provide a more logical variant of this predicate.