```
End = 2
```

In Edinburgh Prolog the second argument is missing. It is fixed to be `'$VAR'`.

free_variables(+*Term*, -*List***)**
Unify *List* with a list of variables, each sharing with a unique variable of *Term*. For example:

```
?- free_variables(a(X, b(Y, X), Z), L).

L = [G367, G366, G371]
X = G367
Y = G366
Z = G371
```

copy_term(+*In*, -*Out***)**
Make a copy of term *In* and unify the result with *Out*. Ground parts of *In* are shared by *Out*. Provided *In* and *Out* have no sharing variables before this call they will have no sharing variables afterwards. `copy_term/2` is semantically equivalent to:

```
copy_term(In, Out) :-
        recorda(copy_key, In, Ref),
        recorded(copy_key, Out, Ref),
        erase(Ref).
```

3.16 Analysing and Constructing Atoms

name(?*Atom*, ?*String***)**
String is a list of ASCII values describing *Atom*. Each of the arguments may be a variable, but not both. When *String* is bound to an ASCII value list describing an integer and *Atom* is a variable *Atom* will be unified with the integer value described by *String* (e.g. '`name(N, "300"), 400 is N + 100`' succeeds).

int_to_atom(+*Int*, +*Base*, -*Atom***)**
Convert *Int* to an ascii representation using base *Base* and unify the result with *Atom*. If $Base \neq 10$ the base will be prepended to *Atom*. $Base = 0$ will try to interpret *Int* as an ASCII value and return `0'c`. Otherwise $2 \leq Base \leq 36$. Some examples are given below.

int_to_atom(45, 2, A)	$\longrightarrow$	$A = 2'101101$
int_to_atom(97, 0, A)	$\longrightarrow$	$A = 0'a$
int_to_atom(56, 10, A)	$\longrightarrow$	$A = 56$

int_to_atom(+*Int*, -*Atom***)**
Equivalent to `int_to_atom(Int, 10, Atom)`.

term_to_atom(?*Term*, ?*Atom***)**
Succeeds if *Atom* describes a term that unifies with *Term*. When *Atom* is instantiated *Atom* is converted and then unified with *Term*. Otherwise *Term* is "written" on *Atom* using `write/1`.

```
read_history(h, '!h', [trace], '%w ?- ', Goal, Bindings)
```

history_depth(-*Int***)**
Dynamic predicate, normally not defined. The user can define this predicate to set the history depth. It should unify the argument with a positive integer. When not defined 15 is used as the default.

prompt(-*Old*, +*New***)**
Set prompt associated with `read/1` and its derivates. *Old* is first unified with the current prompt. On success the prompt will be set to *New* if this is an atom. Otherwise an error message is displayed. A prompt is printed if one of the read predicates is called and the cursor is at the left margin. It is also printed whenever a newline is given and the term has not been terminated. Prompts are only printed when the current input stream is *user*.

3.15 Analysing and Constructing Terms

functor(?*Term*, ?*Functor*, ?*Arity***)**
Succeeds if *Term* is a term with functor *Functor* and arity *Arity*. If *Term* is a variable it is unified with a new term holding only variables. `functor/3` silently fails on instantiation faults[8]

arg(+*Arg*, +*Term*, ?*Value***)**
Term should be instantiated to a term, *Arg* to an integer between 1 and the arity of *Term*. *Value* is unified with the *Arg*-th argument of *Term*.

?*Term* **=..** ?*List*
List is a list which head is the functor of *Term* and the remaining arguments are the arguments of the term. Each of the arguments may be a variable, but not both. This predicate is called 'Univ'. Examples:

```
?- foo(hello, X) =.. List.

List = [foo, hello, X]

?- Term =.. [baz, foo(1)]

Term = baz(foo(1))
```

numbervars(+*Term*, +*Functor*, +*Start*, -*End***)**
Unify the free variables of *Term* with a term constructed from the atom *Functor* with one argument. The argument is the number of the variable. Counting starts at *Start*. *End* is unified with the number that should be given to the next variable. Example:

```
?- numbervars(foo(A, B, A), this_is_a_variable, 0, End).

A = this_is_a_variable(0)
B = this_is_a_variable(1)
```

[8] In version 1.2 instantiation fauls let to error messages. The new version can be used to do type testing without the need to catch illegal instantiations first.

write(*+Stream, +Term*)
Write *Term* to *Stream*.

writeq(*+Term*)
Write *Term* to the current output, using brackets and operators where appropriate. Atoms that need quotes are quoted. Terms written with this predicate can be read back with `read/1` provided the currently active operator declarations are identical.

writeq(*+Stream, +Term*)
Write *Term* to *Stream*, inserting quotes.

print(*+Term*)
Prints *Term* on the current output stream similar to `write/1`, but for each (sub)term of *Term* first the dynamic predicate `portray/1` is called. If this predicate succeeds *print* assumes the (sub)term has been written. This allows for user defined term writing.

print(*+Stream, +Term*)
Print *Term* to *Stream*.

portray(*+Term*)
A dynamic predicate, which can be defined by the user to change the behaviour of `print/1` on (sub)terms. For each subterm encountered that is not a variable `print/1` first calls `portray/1` using the term as argument. For lists only the list as a whole is given to `portray/1`. If portray succeeds `print/1` assumes the term has been written.

read(*-Term*)
Read the next Prolog term from the current input stream and unify it with *Term*. On a syntax error `read/1` displays an error message, attempts to skip the erroneous term and fails. On reaching end-of-file *Term* is unified with the atom `end_of_file`.

read(*+Stream, -Term*)
Read *Term* from *Stream*.

read_clause(*-Term*)
Equivalent to `read/1`, but warns the user for variables only occurring once in a term (singleton variables) which do not start with an underscore if `style_check(singleton)` is active (default). Used to read Prolog source files (see `consult/1`).

read_clause(*+Stream, -Term*)
Read a clause from *Stream*.

read_variables(*-Term, -Bindings*)
Similar to `read/1`, but *Bindings* is unified with a list of '*Name* = *Var*' tuples, thus providing access to the actual variable names.

read_variables(*+Stream, -Term, -Bindings*)
Read, returning term and bindings from *Stream*.

read_history(*+Show, +Help, +Special, +Prompt, -Term, -Bindings*)
Similar to `read_variables/2`, but allows for history substitutions. `history_read/6` is used by the top level to read the user's actions. *Show* is the command the user should type to show the saved events. *Help* is the command to get an overview of the capabilities. *Special* is a list of commands that are not saved in the history. *Prompt* is the first prompt given. Continuation prompts for more lines are determined by `prompt/2`. A `%w` in the prompt is substituted by the event number. See section 2.4 for available substitutions.

SWI-Prolog calls `history_read/6` as follows:

flush
Flush pending output on current output stream. `flush/0` is automatically generated by `read/1` and derivates if the current input stream is *user* and the cursor is not at the left margin.

flush_output(*+Stream*)
Flush output on the specified stream. The stream must be open for writing.

ttyflush
Flush pending output on stream *user*. See also `flush/0`.

get0(*-Char*)
Read the current input stream and unify the next character with *Char*. *Char* is unified with -1 on end of file.

get0(*+Stream, -Char*)
Read the next character from *Stream*.

get(*-Char*)
Read the current input stream and unify the next non-blank character with *Char*. *Char* is unified with -1 on end of file.

get(*+Stream, -Char*)
Read the next non-blank character from *Stream*.

get_single_char(*-Char*)
Get a single character from input stream 'user' (regardless of the current input stream). Unlike `get0/1` this predicate does not wait for a return. The character is not echoed to the user's terminal. This predicate is meant for keyboard menu selection etc.. If SWI-Prolog was started with the `-tty` flag this predicate reads an entire line of input and returns the first non-blank character on this line, or the ASCII code of the newline (10) if the entire line consisted of blank characters.

3.14 Term Reading and Writing

display(*+Term*)
Write *Term* on the current output stream using standard parenthesised prefix notation (i.e. ignoring operator declarations). Display is normally used to examine the internal representation for terms holding operators.

display(*+Stream, +Term*)
Display *Term* on *Stream*.

displayq(*+Term*)
Write *Term* on the current output stream using standard parenthesised prefix notation (i.e. ignoring operator declarations). Atoms that need quotes are quoted. Terms written with this predicate can always be read back, regardless of current operator declarations.

displayq(*+Stream, +Term*)
Display *Term* on *Stream*. Equivalent to Quintus `write_canonical/2`.

write(*+Term*)
Write *Term* to the current output, using brackets and operators where appropriate.

```
?- open('/dev/ttyp4', read, P4),
   wait_for_input([user, P4], Inputs, 0).
```

character_count(*+Stream, -Count*)
Unify *Count* with the current character index. For input streams this is the number of characters read since the open, for output streams this is the number of characters written. Counting starts at 0.

line_count(*+Stream, -Count*)
Unify *Count* with the number of lines read or written. Counting starts at 1.

line_position(*+Stream, -Count*)
Unify *Count* with the position on the current line. Note that this assumes the position is 0 after the open. Tabs are assumed to be defined on each 8-th character and backspaces are assumed to reduce the count by one, provided it is positive.

fileerrors(*-Old, +New*)
Define error behaviour on errors when opening a file for reading or writing. Valid values are the atoms `on` (default) and `off`. First *Old* is unified with the current value. Then the new value is set to *New*.[7]

tty_fold(*-OldColumn, +NewColumn*)
Fold Prolog output to stream *user* on column *NewColumn*. If *Column* is 0 or less no folding is performed (default). *OldColumn* is first unified with the current folding column. To be used on terminals that do not support line folding.

3.13 Primitive Character Input and Output

nl
Write a newline character to the current output stream. On Unix systems `nl/0` is equivalent to `put(10)`.

nl(*+Stream*)
Write a newline to *Stream*.

put(*+Char*)
Write *Char* to the current output stream, *Char* is either an integer-expression evaluating to an ASCII value ($0 \leq Char \leq 255$) or an atom of one character.

put(*+Stream, +Char*)
Write *Char* to *Stream*.

tab(*+Amount*)
Writes *Amount* spaces on the current output stream. *Amount* should be an expression that evaluates to a positive integer (see section 3.19).

tab(*+Stream, +Amount*)
Writes *Amount* spaces to *Stream*.

[7]Note that Edinburgh Prolog defines `fileerrors/0` and `nofileerrors/0`. As this does not allow you to switch back to the old mode I think this definition is better.

open_null_stream(_?Stream_**)**
On Unix systems, this is equivalent to `open('/dev/null', write, Stream)`. Characters written to this stream are lost, but the stream information (see `character_count/2`, etc.) is maintained.

close(_+Stream_**)**
Close the specified stream. If _Stream_ is not open an error message is displayed. If the closed stream is the current input or output stream the terminal is made the current input or output.

current_stream(_?File, ?Mode, ?Stream_**)**
Is true if a stream with file specification _File_, mode _Mode_ and stream identifier _Stream_ is open. The reserved streams _user_ and _user_error_ are not generated by this predicate. If a stream has been opened with mode `append` this predicate will generate mode _write_.

stream_position(_+Stream, -Old, +New_**)**
Unify the position parameters of _Stream_ with _Old_ and set them to _New_. A position is represented by the following term:

```
'$stream_position'(CharNo, LineNo, LinePos).
```

It is only possible to change the position parameters if the stream is connected to a disk file.

3.11.3 Switching Between Implicit and Explicit I/O

The predicates below can be used for switching between the implicit- and the explicit stream based I/O predicates.

set_input(_+Stream_**)**
Set the current input stream to become _Stream_. Thus, open(file, read, Stream), set_input(Stream) is equivalent to see(file).

set_output(_+Stream_**)**
Set the current output stream to become _Stream_.

current_input(_-Stream_**)**
Get the current input stream. Useful to get access to the status predicates associated with streams.

current_output(_-Stream_**)**
Get the current output stream.

3.12 Status of Input and Output Streams

wait_for_input(_+ListOfStreams, -ReadyList, +TimeOut_**)**
Wait for input on one of the streams in _ListOfStreams_ and return a list of streams on which input is available in _ReadyList_. `wait_for_input/3` waits for at most _TimeOut_ seconds. _Timeout_ may be specified as a floating point number to specify fractions of a second. If _Timeout_ equals 0, `wait_for_input/3` waits indefinetely. This predicate can be used to implement timeout while reading and to handle input from multiple sources. The following example will wait for input from the user and an explicitely opened second terminal. On return, _Inputs_ may hold `user` or _P4_ or both.

```
getwd(Wd) :-
        seeing(Old), see(pipe(pwd)),
        collect_wd(String),
        seen, see(Old),
        name(Wd, String).

collect_wd([C|R]) :-
        get0(C), C \== -1, !,
        collect_wd(R).
collect_wd([]).
```

Figure 3.1: Get the working directory

`tell/1` or `append/1` and has not been closed since, writing will be resumed. Otherwise the file is created or –when existing– truncated. See also `append/1`.

append(+*File***)**
Similar to `tell/1`, but positions the file pointer at the end of *File* rather than truncating an existing file. The pipe construct is not accepted by this predicate.

seeing(-*SrcDest***)**
Unify the name of the current input stream with *SrcDest*.

telling(-*SrcDest***)**
Unify the name of the current output stream with *SrcDest*.

seen
Close the current input stream. The new input stream becomes *user*.

told
Close the current output stream. The new output stream becomes *user*.

3.11.2 Explicit Input and Output Streams

The predicates below are part of the Quintus compatible stream-based I/O package. In this package streams are explicitely created using the predicate `open/3`. The resulting stream identifier is then passed as a parameter to the reading and writing predicates to specify the source or destination of the data.

open(+*SrcDest*, +*Mode*, ?*Stream***)**
SrcDest is either an atom, specifying a Unix file, or a term 'pipe(*Command*)', just like `see/1` and `tell/1`. *Mode* is one of `read`, `write` or `append`. *Stream* is either a variable, in which case it is bound to a small integer identifying the stream, or an atom, in which case this atom will be the stream indentifier. In the latter case the atom cannot be an already existing stream identifier. Examples:

```
?- open(data, read, Stream).          % Open 'data' for reading.
?- open(pipe(lpr), write, printer).   % 'printer' is a stream to 'lpr'.
```

clause(*?Head, ?Body, ?Reference*)
Equivalent to `clause/2`, but unifies *Reference* with a unique reference to the clause (see also `assert/2`, `erase/1`). If *Reference* is instantiated to a reference the clause's head and body will be unified with *Head* and *Body*.

3.11 Input and Output

SWI-Prolog provides two different packages for input and output. One confirms to the Edinburgh standard. This package has a notion of 'current-input' and 'current-output'. The reading and writing predicates implicitely refer to these streams. In the second package, streams are opened explicitely and the resulting handle is used as an argument to the reading and writing predicate to specify the source or destination. Both packages are fully integrated; the user may switch freely between them.

3.11.1 Input and Output Using Implicit Source and Destination

The package for implicit input and output destination is upwards compatible to DEC-10 and C-Prolog. The reading and writing predicates refer to resp. the current input- and output stream. Initially these streams are connected to the terminal. The current output stream is changed using `tell/1` or `append/1`. The current input stream is changed using `see/1`. The stream's current value can be obtained using `telling/1` for output- and `seeing/1` for input streams. The table below shows the valid stream specifications. The reserved names `user_input`, `user_output` and `user_error` are for neat integration with the explicit streams.

`user`	This reserved name refers to the terminal
`user_input`	Input from the terminal
`user_output`	Output to the terminal
`stderr or user_error`	Unix error stream (output only)
Atom	Name of a Unix file
`pipe(`*Atom*`)`	Name of a Unix command

Source and destination are either a file, one of the reserved words above, or a term 'pipe(*Command*)'. In the predicate descriptions below we will call the source/destination argument '*SrcDest*'. Below are some examples of source/destination specifications.

```
?- see(data).          % Start reading from file 'data'.
?- tell(stderr).       % Start writing on the error stream.
?- tell(pipe(lpr)).    % Start writing to the printer.
```

Another example of using the `pipe/1` construct is shown on in figure 3.1. Note that the `pipe/1` construct is not part of Prolog's standard I/O reportoire.

see(*+SrcDest*)
Make *SrcDest* the current input stream. If *SrcDest* was already opened for reading with `see/1` and has not been closed since, reading will be resumed. Otherwise *SrcDest* will be opened and the file pointer is positioned at the start of the file.

tell(*+SrcDest*)
Make *SrcDest* the current output stream. If *SrcDest* was already opened for writing with

predicate_property(*?Head, ?Property***)**

Succeeds if *Head* refers to a predicate that has property *Property*. Can be used to test whether a predicate has a certain property, obtain all properties known for *Head*, find all predicates having *property* or even obtaining all information available about the current program. *Property* is one of:

interpreted

Is true if the predicate is defined in Prolog. We return true on this because, although the code is actually compiled, it is completely transparent, just like interpreted code.

built_in

Is true if the predicate is locked as a built-in predicate. This implies it cannot be redefined in it's definition module and it can normally not be seen in the tracer.

foreign

Is true if the predicate is defined in the C language.

dynamic

Is true if the predicate is declared dynamic using the `dynamic/1` declaration.

multifile

Is true if the predicate is declared multifile using the `multifile/1` declaration.

undefined

Is true if a procedure definition block for the predicate exists, but there are no clauses in it and it is not declared dynamic. This is true if the predicate occurs in the body of a loaded predicate, an attempt to call it has been made via one of the meta-call predicates or the predicate had a definition in the past. See the library package *check* for example usage.

transparent

Is true if the predicate is declared transparent using the `module_transparent/1` declaration.

exported

Is true if the predicate is in the public list of the context module.

imported_from(*Module***)**

Is true if the predicate is imported into the context module from module *Module*.

indexed(*Head***)**

Predicate is indexed (see `index/1`) according to *Head*. *Head* is a term whose name and arity are identical to the predicate. The arguments are unified with '1' for indexed arguments, '0' otherwise.

dwim_predicate(*+Term, -Dwim***)**

'Do What I Mean' ('dwim') support predicate. *Term* is a term, which name and arity are used as a predicate specification. *Dwim* is instantiated with the most general term built from *Name* and the arity of a defined predicate that matches the predicate specified by *Term* in the 'Do What I Mean' sence. See `dwim_match/2` for 'Do What I Mean' string matching. Internal system predicates are not generated, unless style_check(+dollar) is active. Backtracking provides all alternative matches.

clause(*?Head, ?Body***)**

Succeeds when *Head* can be unified with a clause head and *Body* with the corresponding clause body. Gives alternative clauses on backtracking. For facts *Body* is unified with the atom *true*. Normally `clause/2` is used to find clause definitions for a predicate, but it can also be used to find clause heads for some body template.

index(*+Head*)

Index the clauses of the predicate with the same name and arity as *Head* on the specified arguments. *Head* is a term of which all arguments are either '1' (denoting 'index this argument') or '0' (denoting 'do not index this argument'). Indexing has no implications for the semantics of a predicate, only on its performance. If indexing is enabled on a predicate a special purpose algorithm is used to select candidate clauses based on the actual arguments of the goal. This algorithm checks whether indexed arguments might unify in the clause head. Only atoms, integers and functors (e.g. name and arity of a term) are considered. Indexing is very useful for predicates with many clauses representing facts.

Due to the representation technique used at most 4 arguments can be indexed. All indexed arguments should be in the first 32 arguments of the predicate. If more than 4 arguments are specified for indexing only the first 4 will be accepted. Arguments above 32 are ignored for indexing.

By default all predicates with arity ≥ 1 are indexed on their first argument. It is possible to redefine indexing on predicates that already have clauses attached to them. This will initiate a scan through the predicate's clause list to update the index summary information stored with each clause.

If –for example– one wants to represents sub-types using a fact list 'sub_type(Sub, Super)' that should be used both to determine sub- and super types one should declare `sub_type/2` as follows:

```
:- index(sub_type(1, 1)).

sub_type(horse, animal).
...
...
```

3.10 Examining the Program

current_atom(*-Atom*)

Successively unifies *Atom* with all atoms known to the system. Note that `current_atom/1` always succeeds if *Atom* is intantiated to an atom.

current_functor(*?Name, ?Arity*)

Successively unifies *Name* with the name and *Arity* with the arity of functors known to the system.

current_flag(*-FlagKey*)

Successively unifies *FlagKey* with all keys used for flags (see `flag/3`).

current_key(*-Key*)

Successively unifies *Key* with all keys used for records (see `recorda/3`, etc.).

current_predicate(*?Name, ?Head*)

Successively unifies *Name* with the name of predicates currently defined and *Head* with the most general term built from *Name* and the arity of the predicate. This predicate succeeds for all predicates defined in the specified module, imported to it, or in one of the modules from which the predicate will be imported if it is called.

erase(*+Reference*)
Erase a record or clause from the database. *Reference* is an integer returned by `recorda/3` or `recorded/3`, `clause/3`, `assert/2`, `asserta/2` or `assertz/2`. Other integers might conflict with the internal consistency of the system. Erase can only be called once on a record or clause. A second call also might conflict with the internal consistency of the system.[6]

flag(*+Key, -Old, +New*)
Key is an atom, integer or term. Unify *Old* with the old value associated with *Key*. If the key is used for the first time *Old* is unified with the integer 0. Then store the value of *New*, which should be an integer, atom or arithmetic integer expression, under *Key*. `flag/3` is a very fast mechanism for storing simple facts in the database. Example:

```
:- module_transparent succeeds_n_times/2.

succeeds_n_times(Goal, Times) :-
        flag(succeeds_n_times, _, 0),
        Goal,
        flag(succeeds_n_times, N, N+1),
        fail ; flag(succeeds_n_times, Times, Times).
```

3.9 Declaring Properties of Predicates

This section describes directives which manipulate attributes of predicate definitions. The functors `dynamic/1`, `multifile/1` and `discontiguous/1` are operators of priority 1150 (see `op/3`), which implies the list of predicates they involve can just be a comma separated list:

```
:- dynamic
        foo/0,
        baz/2.
```

On SWI-Prolog all these directives are just predicates. This implies they can also be called by a program. Do not rely on this feature if you want to maintain portability to other Prolog implementations.

dynamic *+Functor/+Arity, ...*
Informs the interpreter that the definition of the predicate(s) may change during execution (using `assert/1` and/or `retract/1`). Currently `dynamic/1` only stops the interpreter from complaining about undefined predicates (see `unknown/2`). Future releases might prohibit `assert/1` and `retract/1` for not-dynamic declared procedures.

multifile *+Functor/+Arity, ...*
Informs the system that the specified predicate(s) may be defined over more than one file. This stops `consult/1` from redefining a predicate when a new definition is found.

discontiguous *+Functor/+Arity, ...*
Informs the system that the clauses of the specified predicate(s) might not be together in the source file. See also `style_check/1`.

[6] BUG: The system should have a special type for pointers, thus avoiding the Prolog user having to worry about consistency matters. Currently some simple heuristics are used to determine whether a reference is valid.

is considerably faster than the mechanisms described above, but can only be used to store simple status information like counters, etc.

abolish(+*Functor*, +*Arity***)**
Removes all clauses of a predicate with functor *Functor* and arity *Arity* from the database. Unlike version 1.2, all predicate attributes (dynamic, multifile, index, etc.) are reset to their defaults. Abolishing an imported predicate only removes the import link; the predicate will keep its old definition in its definition module. For 'cleanup' of the dynamic database, one should use `retractall/1` rather than `abolish/2`.

retract(+*Term***)**
When *Term* is an atom or a term it is unified with the first unifying fact or clause in the database. The fact or clause is removed from the database.

retractall(+*Term***)**
All facts or clauses in the database that unify with *Term* are removed.

assert(+*Term***)**
Assert a fact or clause in the database. *Term* is asserted as the last fact or clause of the corresponding predicate.

asserta(+*Term***)**
Equivalent to `assert/1`, but *Term* is asserted as first clause or fact of the predicate.

assertz(+*Term***)**
Equivalent to `assert/1`.

assert(+*Term*, -*Reference***)**
Equivalent to `assert/1`, but *Reference* is unified with a unique reference to the asserted clause. This key can later be used with `clause/3` or `erase/1`.

asserta(+*Term*, -*Reference***)**
Equivalent to `assert/2`, but *Term* is asserted as first clause or fact of the predicate.

assertz(+*Term*, -*Reference***)**
Equivalent to `assert/2`.

recorda(+*Key*, +*Term*, -*Reference***)**
Assert *Term* in the recorded database under key *Key*. *Key* is an integer, atom or term. *Reference* is unified with a unique reference to the record (see `erase/1`).

recorda(+*Key*, +*Term***)**
Equivalent to `recorda(Key, Value, _)`.

recordz(+*Key*, +*Term*, -*Reference***)**
Equivalent to `recorda/3`, but puts the *Term* at the tail of the terms recorded under *Key*.

recordz(+*Key*, +*Term***)**
Equivalent to `recordz(Key, Value, _)`.

recorded(+*Key*, -*Value*, -*Reference***)**
Unify *Value* with the first term recorded under *Key* which does unify. *Reference* is unified with the memory location of the record.

recorded(+*Key*, -*Value***)**
Equivalent to `recorded(Key, Value, _)`.

call(*+Goal*)
Invoke *Goal* as a goal. Note that clauses may have variables as subclauses, which is identical to `call/1`, except when the argument is bound to the cut. See !/0.

apply(*+Term, +List*)
Append the members of *List* to the arguments of *Term* and call the resulting term. For example: '`apply(plus(1), [2, X])`' will call '`plus(1, 2, X)`'. `Apply/2` is incorporated in the virtual machine of SWI-Prolog. This implies that the overhead can be compared to the overhead of `call/1`.

not *+Goal*
Succeeds when *Goal* cannot be proven. Retained for compatibility only. New code should use `\+/1`.

once(*+Goal*)
Defined as:

```
once(Goal) :-
        Goal, !.
```

`Once/1` can in many cases be replaced with `->/2`. The only difference is how the cut behaves (see !/0). The following two clauses are identical:

```
1) a :- once((b, c)), d.
2) a :- b, c -> d.
```

ignore(*+Goal*)
Calls *Goal* as `once/1`, but succeeds, regardless of whether *Goal* succeeded or not. Defined as:

```
ignore(Goal) :-
        Goal, !.
ignore(_).
```

3.8 Database

SWI-Prolog offers three different database mechanisms. The first one is the common assert/retract mechanism for manipulating the clause database. As facts and clauses asserted using `assert/1` or one of it's derivates become part of the program these predicates compile the term given to them. `Retract/1` and `retractall/1` have to unify a term and therefore have to decompile the program. For these reasons the assert/retract mechanism is expensive. On the other hand, once compiled, queries to the database are faster than querying the recorded database discussed below. See also `dynamic/1`.

The second way of storing arbitrary terms in the database is using the "recorded database". In this database terms are associated with a *key*. A key can be an atom, integer or term. In the last case only the functor and arity determine the key. Each key has a chain of terms associated with it. New terms can be added either at the head or at the tail of this chain. This mechanism is considerably faster than the assert/retract mechanism as terms are not compiled, but just copied into the heap.

The third mechanism is a special purpose one. It associates an integer or atom with a key, which is an atom, integer or term. Each key can only have one atom or integer associated with it. It again

!

Cut. Discard choice points of parent frame and frames created after the parent frame. Note that the control structures `;/2`, `|/2` `->/2` and `\+/1` are normally handled by the compiler and do not create a frame, which implies the cut operates through these predicates. Some examples are given below. Note the difference between `t3/1` and `t4/1`. Also note the effect of `call/1` in `t5/0`. As the argument of `call/1` is evaluated by predicates rather than the compiler the cut has no effect.[5]

```
t1 :- (a, !, fail ; b).          % cuts a/0 and t1/0
t2 :- (a -> b, !  ; c).          % cuts b/0 and t2/0
t3(G) :- a, G, fail.             % if 'G = !' cuts a/0 and t1/1
t4(G) :- a, call(G), fail.       % if 'G = !' cut has no effect
t5 :- call((a, !, fail ; b)).    % Cut has no effect
t6 :- \+ (a, !, fail ; b).       % cuts a/0 and t6/0
```

+Goal1 , *+Goal2*

Conjunction. Succeeds if both 'Goal1' and 'Goal2' can be proved. It is defined as (this definition does not lead to a loop as the second comma is handled by the compiler):

```
Goal1, Goal2 :- Goal1, Goal2.
```

+Goal1 ; *+Goal2*

The 'or' predicate is defined as:

```
Goal1 ; _Goal2 :- Goal1.
_Goal1 ; Goal2 :- Goal2.
```

+Goal1 | *+Goal2*

Equivalent to `;/2`. Retained for compatibility only. New code should use `;/2`.

+Condition -> *+Action*

If-then and If-Then-Else. Implemented as:

```
If -> Then; _Else :- If, !, Then.
If -> _Then; Else :- !, Else.
If -> Then :- If, !, Then.
```

\+ *+Goal*

Succeeds if 'Goal' cannot be proven (mnemnonic: + refers to *provable* and the backslash is normally used to indicate negation).

3.7 Meta-Call Predicates

Meta call predicates are used to call terms constructed at run time. The basic meta-call mechanism offered by SWI-Prolog is to use variables as a subclause (which should of course be bound to a valid goal at runtime). A meta-call is slower than a normal call as it involves actually searching the database at runtime for the predicate, while for normal calls this search is done at compile time.

[5] Version 1.2 did not compile `;/2`, etc.. To make the cut work a special predicate attribute called 'cut_parent' was introduced. This implied the cut had effect in all the examples. The current implementation is much neater and considerably faster.

+Term1 `=` *+Term2*
: Unify *Term1* with *Term2*. Succeeds if the unification succeeds.

+Term1 `\=` *+Term2*
: Equivalent to '`\+ Term1 = Term2`'.

+Term1 `=@=` *+Term2*
: Succeeds if *Term1* is 'structurally equal' to *Term2*. Structural equivalence is weaker than equivalence (`==`/2), but stronger than unification (`=`/2). Two terms are structurally equal if their tree representation is identical and they have the same 'pattern' of variables. Examples:

`a =@= A`	false
`A =@= B`	true
`x(A,A) =@= x(B,C)`	false
`x(A,A) =@= x(B,B)`	true
`x(A,B) =@= x(C,D)`	true

+Term1 `\=@=` *+Term2*
: Equivalent to '`\+ Term1 =@= Term2`'.

+Term1 `@<` *+Term2*
: Succeeds if *Term1* is before *Term2* in the standard order of terms.

+Term1 `@=<` *+Term2*
: Succeeds if both terms are equal (`==`) or *Term1* is before *Term2* in the standard order of terms.

+Term1 `@>` *+Term2*
: Succeeds if *Term1* is after *Term2* in the standard order of terms.

+Term1 `@>=` *+Term2*
: Succeeds if both terms are equal (`==`) or *Term1* is after *Term2* in the standard order of terms.

3.6 Control Predicates

The predicates of this section implement control structures. Normally these constructs are translated into virtual machine instructions by the compiler. It is still necessary to implement these constructs as true predicates to support meta-calls, as demonstrated in the example below. The predicate finds all currently defined atoms of 1 character long. Note that the cut has no effect when called via one of these predicates (see !/0).

```
one_character_atoms(As) :-
        findall(A, (current_atom(A), atom_length(A, 1)), As).
```

fail
: Always fail.

true
: Always succeed.

repeat
: Always succeed, provide an infinite number of choice points.

3.4 Verify Type of a Term

var(*+Term*)
Succeeds if *Term* currently is a free variable.

nonvar(*+Term*)
Succeeds if *Term* currently is not a free variable.

integer(*+Term*)
Succeeds if *Term* is bound to an integer.

float(*+Term*)
Succeeds if *Term* is bound to a floating point number.

number(*+Term*)
Succeeds if *Term* is bound to an integer or a floating point number.

atom(*+Term*)
Succeeds if *Term* is bound to an atom.

string(*+Term*)
Succeeds if *Term* is bound to a string.

atomic(*+Term*)
Succeeds if *Term* is bound to an atom, string, integer or floating point number.

ground(*+Term*)
Succeeds if *Term* holds no free variables.

3.5 Comparison and Unification or Terms

Standard Order of Terms

Comparison and unification of arbitrary terms. Terms are ordered in the so called "standard order". This order is defined as follows:

1. *Variables* $<$ *Atoms* $<$ *Strings*[3] $<$ *Numbers* $<$ *Terms*
2. *Old Variable* $<$ *New Variable*[4]
3. *Atoms* are compared alphabetically.
4. *Strings* are compared alphabetically.
5. *Numbers* are compared by value. Integers and floats are treated identically.
6. *Terms* are first checked on their functor (alphabetically), then on their arity and finally recursively on their arguments, left most argument first.

+Term1 `==` *+Term2*
Succeeds if *Term1* is equivalent to *Term2*. A variable is only identical to a sharing variable.

+Term1 `\==` *+Term2*
Equivalent to '`\+ Term1 == Term2`'.

[3] Strings might be considered atoms in future versions. See also section 3.17

[4] In fact the variables are compared on their (dereferenced) addresses. Variables living on the global stack are always $<$ than variables on the local stack. Programs should not rely on the order in which variables are sorted.

```
?- preprocessor(Old, '/lib/cpp -C -P %f'), consult(...).

Old = none
```

3.3 Listing Predicates and Editor Interface

SWI-Prolog offers an interface to the Unix *vi* editor (vi(1)), Richard O'Keefe's *top* editor [O'Keefe, 1985] and the GNU-EMACS invocations `emacs` and `emacsclient`. Which editor is used is determined by the Unix environment variable `EDITOR`, which should hold the full pathname of the editor. If this variable is not defined, vi(1) is used.

After the user quits the editor `make/0` is invoked to reload all modified source files using `consult/1`. If the editor can be quit such that an exit status non-equal to 0 is returned `make/0` will not be invoked. *top* can do this by typing control-C, *vi* cannot do this.

A predicate specification is either a term with the same functor and arity as the predicate wanted, a term of the form `Functor/Arity` or a single atom. In the latter case the database is searched for a predicate of this name and arbitrary arity (see `current_predicate/2`). When more than one such predicate exists the system will prompt for confirmation on each of the matched predicates. Predicates specifications are given to the 'Do What I Mean' system (see `dwim_predicate/2`) if the requested predicate does not exist.

ed(+*Pred***)**
: Invoke the user's preferred editor on the source file of *Pred*, providing a search specification which searches for the predicate at the start of a line.

ed
: Invoke `ed/1` on the predicate last edited using `ed/1`. Asks the user to confirm before starting the editor.

edit(+*File***)**
: Invoke the user's preferred editor on *File*. *File* is a file specification as for `consult/1` (but not a list). Note that the file should exist.

edit
: Invoke `edit/1` on the file last edited using `edit/1`. Asks the user to confirm before starting the editor.

listing(+*Pred***)**
: List specified predicates (when an atom is given all predicates with this name will be listed). The listing is produced on the basis of the internal representation, thus loosing user's layout and variable name information. See also `portray_clause/1`.

listing
: List all predicates of the database using `listing/1`.

portray_clause(+*Clause***)**
: Pretty print a clause as good as we can. A clause should be specified as a term '`Head :- Body`' (put brackets around it to avoid operator precedence problems). Facts are represented as '`Head :- true`'.

File may also be `library(Name)`, in which case the libraries are searched for a file with the specified name. See also `library_directory/1`. `consult/1` may be abbreviated by just typing a number of file names in a list. Examples:

`?- consult(load).`	% consult 'load' or 'load.pl'
`?- [library(quintus)].`	% load Quintus compatibility library

ensure_loaded(*+File***)**
Equivalent to `consult/1`, but the file is consulted only if this was not done before. This is the recommended way to load files from other files.

make
Consult all source files that have been changed since they were consulted. It checks *all* loaded source files: files loaded into a compiled state using `pl -c ...` and files loaded using consult or one of its derivates. `make/0` is normally invoked by the `edit/[0,1]` and `ed/[0,1]` predicates. `make/0` can be combined with the compiler to speed up the development of large packages. In this case compile the package using

```
sun% pl -g make -o my_program -c file ...
```

If 'my_program' is started it will first reconsult all source files that have changed since the compilation.

library_directory(*-Atom***)**
Dynamic predicate used to specify library directories. Default `.`, `./lib`, `~/lib/prolog` and the system's library (in this order) are defined. The user may add library directories using `assert/1` or remove system defaults using `retract/1`.

source_file(*-File***)**
Succeeds if *File* was loaded using `consult/1` or `ensure_loaded/1`. *File* refers to the full path name of the file (see `expand_file_name/2`). `Source_file/1` backtracks over all loaded source files.

source_file(*?Pred, ?File***)**
Is true if the predicate specified by *Pred* was loaded from file *File*, where *File* is an absolute path name (see `expand_file_name/2`). Can be used with any instantiation pattern, but the database only maintains the source file for each predicate. Predicates declared *multifile* (see `multifile/1`) cannot be found this way.

term_expansion(*+Term1, -Term2***)**
Dynamic predicate, normally not defined. When defined by the user all terms read during consulting that are given to this predicate. If the predicate succeeds Prolog will assert *Term2* in the database rather then the read term (*Term1*). *Term2* may be a term of a the form '?- *Goal*' or ':- *Goal*'. *Goal* is then treated as a directive. *Term2* may also be a list, in which case all terms of the list are stored in the database or called (for directives).

compiling
Succeeds if the system is compiling source files with the `-c` option into an intermediate code file. Can be used to perform code optimisations in `expand_term/2` under this condition.

preprocessor(*-Old, +New***)**
Read the input file via a Unix process that acts as preprocessor. A preprocessor is specified as an atom. The first occurrence of the string '`%f`' is replaced by the name of the file to be loaded. The resulting atom is called as a Unix command and the standard output of this command is loaded. To use the Unix C preprocessor one should define:

Chapter 3

Built-In Predicates

3.1 Notation of Predicate Descriptions

We have tried to keep the predicate descriptions clear and concise. First the predicate name is printed in bold face, followed by the arguments in italics. Arguments are preceded by a '+', '–' or '?' sign. '+' indicates the argument is input to the predicate, '–' denotes output and '?' denotes 'either input or output'.[1] Constructs like '`op/3`' refer to the predicate 'op' with arity '3'.

3.2 Consulting Prolog Source files

SWI-Prolog source files normally have a suffix '`.pl`'. Specifying the suffix is optional. All predicates that handle source files first check whether a file with suffix '`.pl`' exists. If not the plain file name is checked for existence. Library files are specified by embedding the file name using the functor `library/1`. Thus '`foo`' refers to '`foo.pl`' or '`foo`' in the current directory, '`library(foo)`' refers to '`foo.pl`' or '`foo`' in one of the library directories specified by the dynamic predicate `library_directory/1`.

SWI-Prolog recognises grammar rules as defined in [Clocksin & Melish, 1981]. The user may define additional compilation of the source file by defining the dynamic predicate `term_expansion/2`. Transformations by this predicate overrule the systems grammar rule transformations. It is not allowed to use `assert/1`, `retract/1` or any other database predicate in `term_expansion/2` other than for local computational purposes.[2]

Directives may be placed anywhere in a source file, invoking any predicate. They are executed when encountered. If the directive fails, a warning is printed. Directives are specified by :-/1 or ?-/1. There is no difference between the two.

SWI-Prolog does not have a separate `reconsult/1` predicate. Reconsulting is implied automatically by the fact that a file is consulted which is already loaded.

consult(*+File*)

Read *File* as a Prolog source file. *File* may be a list of files, in which case all members are consulted in turn. *File* may start with the csh(1) special sequences `~`, `~<user>` and `$<var>`.

[1] These marks do NOT suggest instanstiation (e.g. var(+Var)).

[2] It does work for consult, but makes it impossible to compile programs into a stand alone executable (see section 2.6)

Option	Default[a]	Area name	Description
-L	200K (2M)	**local stack**	The local stack is used to store the execution environments of procedure invocations. The space for an environment is reclaimed when it fails, exits without leaving choice points, the alternatives are cut of with the `!` predicate or no choice points have been created since the invocation and the last subclause is started (tail recursion optimisation).
-G	100K (4M)	**global stack**	The global stack is used to store terms created during Prolog's execution. Terms on this stack will be reclaimed by backtracking to a point before the term was created or by garbage collection (provided the term is no longer referenced).
-T	50K (4M)	**trail stack**	The trail stack is used to store assignments during execution. Entries on this stack remain alive until backtracking before the point of creation or the garbage collector determines they are nor needed any longer.
-A	5K (1M)	**argument stack**	The argument stack is used to store one of the intermediate code interpreter's registers. The amount of space needed on this stack is determined entirely by the depth in which terms are nested in the clauses that constitute the program. Overflow is most likely when using long strings in a clause.

[a]Defaults may depend on local installation. The value between brackets is the default limit for machines that allow for dynamic stack allocation.

Table 2.3: Memory areas

2.12.3 Reserved Names

The boot compiler (see `-b` option) does not support the module system (yet). As large parts of the system are written in Prolog itself we need some way to avoid name clashes with the user's predicates, database keys, etc. Like Edinburgh C-Prolog [Pereira, 1986] all predicates, database keys, etc. that should be hidden from the user start with a dollar (`$`) sign (see `style_check/2`).

The compiler uses the special functor `$VAR$/1` while analysing the clause to compile. Using this functor in a program causes unpredictable behaviour of the compiler and resulting program.