Logic Programming and Deductive Databases

————————————

Chapter 6: Built-in Predicates in Prolog

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2023

http://www.informatik.uni-halle.de/~brass/lp23/

Objectives

After completing this chapter, you should be able to:

- define and explain binding patterns (modes).

- write Prolog programs using built-in predicates.

Contents

Introduction (1)

- A very pure Prolog Program contains only predicates that are defined by facts and rules.

- However, for larger, real-world applications, this is not very realistic.

- Already for simulating SQL-queries in Prolog, needs e.g. the standard arithmetic operators $+$, $-$, $*$, $/$, and the comparison operators $=$, $\neq$, $<$, $>$, $\leq$, $\geq$.

- For real programs, one needs also a mechanism for input/output etc.

Introduction (2)

- Theoretically, it might be possible to define e.g. $<$ for all numbers that occur in the program by facts.

- But it would at least be tedious to enumerate all facts $X < Y$ that might be important for a program.

- Therefore, Prolog systems and deductive database systems have certain predicates predefined by procedures in the system. These predicates are called "built-in predicates".

- E.g. for the query $3 < 5$, the system does not look up facts and rules, but calls a built-in procedure written e.g. in C.

Introduction (3)

- Since built-in predicates are defined in the system, it is illegal to write a literal with a built-in predicate in the head of a rule, e.g.

$$X \leq Z \leftarrow X \leq Y \wedge Y \leq Z. \quad \textbf{Error!}$$

- Rules contribute to the definition of the predicate in their head, and the definition of built-in predicates cannot be changed.

    Typical error message: "Attempt to modify static procedure $\leq / 2$."

- Of course, one can use built-in predicates in the body of a rule (i.e. call them).

Introduction (4)

- Built-in predicates often have restrictions on their arguments: Certain arguments must not be (unbound) variables, but must be known values.

  Whereas in Pure Prolog, predicates have no predefined input and output arguments, now certain arguments can only be input arguments.

- E.g. the query $X > 3$ is not permitted: It would immediately have infinitely many solutions.

  A typical error message is "instantiation fault in $X > 3$".

- But "$p(X) \land X > 3$" is permitted: When $X > 3$ is executed, X has already a concrete value.

  This assumes that p actually binds its argument to a value, e.g. "$p(X) \leftarrow true$" would not help here.

Binding Patterns (1)

Definition:

- A binding pattern for a predicate of arity n (i.e. a predicate with n arguments) is a string over $\{b, f\}$ of length n.

  Binding patterns are not only important for built-in predicates, but can be specified for any predicate.

- The i-th character in a binding pattern β for a predicate p defines whether i-th argument of p is an input or output argument in a call described by β. If the character is

  - b, we call the argument "bound" (input argument),

  - f, we call the argument "free" (output argument).

- In Prolog, "binding patterns" are called "modes", and "+" is written for b, "-" for f, and "?" is used as wildcard.

Binding Patterns (2)

- E.g. consider a predicate $sum(X, Y, Z)$ that is true if and only if $X + Y = Z$.

    Not every Prolog system has such a predicate, because Prolog uses `is` for evaluating arithmetic expressions, see below.

- The predicate `sum` will typically support the binding pattern `bbf`. This corresponds e.g. to the call $sum(3, 5, X)$.

- It can support also the binding patterns `bfb`, `fbb` (and `bbb`, see below).

- E.g. $sum(3, X, 8)$ binds X to $8 - 3 = 5$.

Binding Patterns (3)

- If all three binding patterns are supported, a deductive DBMS will internally have three procedures:

  - sum_bbf(X, Y, var Z): begin Z := X + Y; end

  - sum_bfb(X, var Y, Z): begin Y := Z - X; end

  - sum_fbb(var X, Y, Z): begin X := Z - Y; end

    The keyword var is from Pascal. It was used to mark output parameters (call by reference, i.e. a variable is passed instead of a value).

- The compiler then selects the right procedure depending on the arguments.

  In Prolog, it is not always possible for the compiler to know whether a variable will be bound or free, therefore, there might be a runtime test to check which case applies. Thus, there is only one procedure with three different cases.

Binding Patterns (4)

- In the example, a predicate plus a binding pattern corresponds to a classical procedure.

- However, in general, a predicate can still have multiple solutions or fail.

- Typically, built-in predicates can succeed only once.

    This is not a strict requirement, but makes the interface simpler.

    A procedure for a predicate that can fail has a boolean result value.

- Prolog systems and deductive DBMS can usually be extended by adding new built-in predicates written in C or similar languages.

Binding Patterns (5)

Definition:

- A binding pattern $\alpha_1 \ldots \alpha_n$ is more general than a binding pattern $\beta_1 \ldots \beta_n$ iff for all $i \in \{1, \ldots, n\}$:
  $\alpha_i = \mathtt{b} \implies \beta_i = \mathtt{b}$.

Example/Remark:

- The binding pattern bbf is more general than bbb.

- One can always use a procedure for a more general binding pattern.

- E.g. the compiler could transform $\mathtt{sum}(3, 5, 8)$ into $\mathtt{sum}(3, 5, \mathtt{X}) \land \mathtt{X} = 8$ (with a new variable X).

Binding Patterns (6)

Binding Patterns and Database Relations:

- Database relations have no binding restrictions, i.e. they can be evaluated for the binding pattern f . . . f.

  This is done by a full table scan. E.g. if the relation $father(X, Y)$ is stored as a heap file, even the literal $father(alan, Y)$ is evaluated like $father(X, Y) \land X = alan$: The system reads every tuple in the relation and checks whether the first attribute has the value alan.

- If there is an index on the first attribute, the system uses that index for the binding pattern bf (and bb). Each index supports a specific binding pattern.

  Since B-tree indexes support also e.g. <-conditions, indexes can more generally be seen as parameterized pre-computed queries.

Binding Patterns (7)

The meaning of "bound":

- There are two different interpretations of what a bound argument is:

    - Weakly bound: Anything except a variable.

    - Strongly bound: A ground term.

- Does a complex term with a variable somewhere inside, e.g. "$[1, X, 2]$ count as "bound"?

- In deductive databases such "terms with holes" are normally excluded (see "range restriction" in Chapter 11).

    Then there is no difference between strongly bound and weakly bound.

Binding Patterns (8)

- In Prolog, complex terms with holes are possible and sometimes useful (e.g. for meta-programming).

  Meta-programming means to process programs as data. This is especially easy in Prolog.

- But then it depends on the predicate where exactly variables might appear.

  E.g. a predicate length(L, N) that computes the length N of a list L could process $[1, X, 2]$, but not $[1, 2 \,|\, X]$.

- Thus, it is safest to assume that "bound" means "strongly bound", but that exceptions are possible.

Binding Patterns (9)

- In Prolog, the programmer usually knows the binding pattern for which a predicate is called.

  In contrast to deductive databases, where very different queries must be executed, a Prolog program typically has a "main" predicate that calls (directly or indirectly) all other predicates.

- It is common to document this by writing a comment line that lists the predicate with its arguments, where each argument is prefixed with

    - "+" for bound (input) arguments, and

    - "-" for free (output) arguments, and

    - "?" for unrestricted arguments.

Binding Patterns (10)

- E.g.:

  ```
  % length(+L, -N): N is the length of list L.
  length([], 0).
  length([_|R], N) :- length(R,M), N is M+1.
  ```

- I.e. the programmer assumes that the first argument is bound and the second argument is free.

  The program might work in other cases, but there is no guarantee. Since the binding pattern appears only in a comment, the Prolog system does not check predicate calls. In some systems, "mode" declarations (that specify binding patterns) help the compiler to optimize the program. Some systems require mode declarations for exported predicates (when modules are separately compiled).

- Ideal logic programs have no binding restrictions!

Predicate Documentation (1)

Meaning of the Predicate:

- Purpose/Function of the predicate ("synopsis").

    If the predicate name is an abbreviation, what is the full version? Use meaningful names for the arguments.

- Reasons for the truth value false ("fails").

    It is best to specify which mathematical relation is defined by the predicate.

- Behaviour on backtracking ("resatisfiable?").

    Can there be several solutions?

- Side effects.

    Input/output, changes of the data base, changes in system settings.

Predicate Documentation (2)

Reasons for Error Messages (Exceptions):

- Type-restrictions for arguments.

    E.g. it must be a number, a callable term, etc.

    ### Which arguments must be free/bound?

    Usually, arguments that must be bound are prefixed with "+", and
    arguments that should be unbound variables are marked with "-". "?"
    marks an argument without restrictions.

Further Information:

- Examples.

- Related predicates ("see also").

Contents

Term Comparison (1)

- t_1 = t_2: t_1 and t_2 are unifiable.

- t_1 == t_2: t_1 and t_2 are textually identical.

  E.g. `X = a` is true and has the side effect of binding `X` to `a`.

  However, `X == a` is false (unless `X` was already bound to `a`).

- t_1 \= t_2: t_1 and t_2 are not unifiable.

- t_1 \== t_2: t_1 and t_2 are not textually identical.

- t_1 @< t_2: t_1 is before t_2 in the standard term order.

  The standard order of terms is explained on the next slide.

- t_1 @> t_2: t_1 is after t_2 in the standard term order.

Term Comparison (2)

- t_1 `@=<` t_2: Equivalent to t_1 `@<` t_2 or t_1 `==` t_2.

- t_1 `@>=` t_2: Equivalent to t_1 `@>` t_2 or t_1 `==` t_2.

- The standard order of terms is partially system dependend (despite its name), but often one needs only any order.

  E.g. first variables (in undefined sequence), then atoms (alphabetically), then strings (alphabetically), then numbers (in the usual order), then compound terms (first by arity, then by name, then recursively by arguments from left to right).

- `compare`(o, t_1, t_2): Binds o to <, =, or >.

Term Classification (1)

- Terms have different types, e.g. integers, atoms, variables. There are various type test predicates:

  - var(t): t is an unbound (free) variable.

  - nonvar(t): t is not an unbound variable.

  - atom(t): t is an atom, e.g. abc.

  - atomic(t): t is an atom or a number.

    Depending on the system, also strings might count as atomic.

  - integer(t): t is an integer.

  - float(t): t is a floating-point number.

    Depending on the system, this might also be called real(t).

Term Classification (2)

- Type test predicates, continued:

  - number(t): t is integer or floating-point number.

  - string(t): t is a string.

    This exists only in systems that represent strings as a data type of its own, not as lists of ASCII codes.

  - compound(t): t is a compound term, e.g. f(X).

  - callable(t): t is atom or compound term.

- The result depends on the current execution state, e.g. var(X), X=2 succeeds, but X=2, var(X) fails.

  If one uses such predicates, one cannot rely on the commutativity of conjunction.

Term Manipulation

- functor(t,f,n): t is a term with functor f, arity n.

    This can be used either to extract the functor from a term (binding pattern bff) or to construct a term with the given functor and n distinct variables as arguments (binding pattern fbb).

- arg(n,t,a): a is the n-th argument of t.

- t =.. L: L is a list consisting of the functor and the arguments of t. E.g. f(a,b) =.. [f,a,b].

    This predicate is called "univ". It can be used in both directions (binding pattern bf and fb).

Contents

Conversion Atom↔String (1)

- atom_chars(Atom, List):
  List is the name of the atom Atom as a list of one
  character atoms, e.g. atom_chars(abc, [a, b, c]).

  The predicate can be called with binding pattern bf to split an atom into
  its single characters, but it can also be used with binding pattern fb to
  generate an atom with a given name (in this representation).

- atom_codes(Atom, List):
  List is the name of the atom Atom as a list of ASCII
  codes, e.g. atom_codes(abc, [97, 98, 99]).

  This, too, supports the binding patterns bf and fb. Lists of ASCII codes
  are the classical representation of strings in Prolog.

Conversion Atom↔String (2)

- name(Atomic, List):
  List is the external representation of the atomic value
  (atom or number) Atomic as a list of ASCII codes, e.g.
  name(abc, [97, 98, 99]).

    This is very similar to atom_codes, but works not only on atoms, but also
    on numbers. Thus, if the List happens to be a list of ASCII codes of
    digits, one does not get an atom, but an integer. This predicate is probably
    older than atom_codes, but did not make it into the ISO Standard.
    ECLiPSe has name, but not atom_codes or atom_chars.

- atom_string(Atom, String):
  Bidirectional conversion between atoms and the new
  string datatype, e.g. atom_string(abc, "abc").

String Functions (1)

- SWI-Prolog has strings as new datatype.

    As explained above, classical Prolog systems represent strings as list of character codes. ECLiPSe Prolog also has a string data type, but the functions have different names.

- Conversion functions:

    - string_to_atom(s, a): Conversion between string and atom.

        Despite its name, both directions are supported (binding patterns bf and fb).

    - string_to_list(s, L): Conversion between string and list of ASCII codes.

String Functions (2)

- Other string functions:

  - string_length(s, n): Computes the number of characters in s.

  - string_concat(s_1, s_2, s_3): String concatenation.

    Supports binding patterns ffb, bbf. There is also atom_concat.

  - sub_string(s_1, n_1, n_2, n_3, s_2): s_2 is substring of s_1.

    The substring starts at position n_1 (i.e. there are n_1 characters before the match), it has length n_2, and there are n_3 characters (in s_1) after the match. There is also sub_atom.

Contents

Arithmetic Predicates (1)

- Arithmetic expressions can be evaluted with the built-in predicate `is`, e.g. `X is Y+1`.

- `is` is defined as infix operator (`xfx`) of priority 700.

- The arithmetic expression can contain `+`, `-` (unary and binary), `*`, `/`, `div` (integer division), `mod` (modulo, remainder of `div`), `/\` (bit-and), `\/` (bit-or), `<<` (left shift), `>>` (right shift), `\` (bit complement).

  Possibly also functions such as `sin`, `cos`, etc. can be used.

- The right argument must be variable-free, i.e. `is` supports only the binding patterns `fb` and `bb`.

Arithmetic Predicates (2)

- Arithmetic comparison operators first evaluate expressions on both sides before they do the comparison. E.g. $X + 1 < Y * 2$ is possible.

- Both arguments must be variable-free (binding pattern bb). Of course, bound variables are no problem.

- Arithmetic comparison operators are:
  =:= $(=)$, =\= $(\neq)$, < $(<)$, > $(>)$, =< $(\leq)$, >= $(\geq)$.

  The equality test is written =:=, because = is already the unification (which does not evaluate arithmetic expressions). In the same way, inequality is written =\=, because \= means "does not unify with". Note that $\leq$ is written =<, because the Prolog designers wanted to save the arrow <= for other purposes.

Exercises

- Define a predicate to compute the Fibonacci numbers:
$$f(n) := \begin{cases} 1 & n = 0,\ n = 1 \\ f(n-1) + f(n-2) & n \geq 2. \end{cases}$$

- Define $\text{sum}(X, Y, Z)$ that holds iff $X + Y = Z$ and can handle the binding patterns bbf, bfb, fbb, bbb.

- Define a predicate $\text{makeground}(t)$ that binds all variables that appear in t to x.

Contents

Constructed Goals

- Proof goals can be dynamically constructed, i.e. can be computed at runtime.

    In purely compiled languages, that is not possible.

- call(A): Executes the literal A.

- In many Prolog systems, one can write simply X instead of call(X).

    But it might be clearer to use call.

All Solutions (1)

- findall(X, A, L): L is the list of all X such that A is true.

- E.g. given the facts $p(a)$ and $p(b)$,

  findall(X, p(X), L)

  returns L = [a, b].

- It is not required that the first argument is a variable, it could also be e.g. f(X,Y) if one is interested in bindings for both variables.

  I.e. in general, the result list contains the instantiation of the first argument whenever a solution to the second argument was found.

All Solutions (2)

- bagof(X, A, L): (similar to findall).

    The different lies in the treatment of variables that occur in A, but do not occur in X. findall treats them as existentially quantified, i.e. it does not bind them, and findall can succeed only once. In contrast, bagof binds such variables to a value and collects then only solutions with this value. Upon backtracking, one can also get other solutions. For example, suppose that p is defined by the facts p(a,1), p(a,2), p(b,3). Then findall(X, p(Y,X), L) would bind L=[1,2,3]. However, bagof(X, p(Y,X), L) would succeed two times: One for Y=a and L=[1,2], and once for Y=b and L=[3].

- setof(X, A, L): As bagof, but the result list is ordered and does not contain duplicates.

Contents

Dynamic Database (1)

- Prolog systems permit that the definition of certain predicates is modified at runtime.

- E.g. if a database relation is represented as a set of facts, one can insert and delete facts.

- Such changes persist even when Prolog backtracks to find another solution.

    Input/output and modifications of the dynamic database are the only changes that are not undone upon backtracking.

Dynamic Database (2)

- Since modern Prolog systems normally compile predicates, one must explicitly declare predicates that can be modified at runtime:

$$:- \text{dynamic}(p/n).$$

- assert(F): The clause F is inserted into the dynamic database.

  Normally, F will be a fact, but it is also possible to assert rules.

  Some Prolog systems guarantee that the new clause is appended at the end of the predicate definition, but officially, there is no guarantee about the order unless one uses asserta (insert at the beginning) or assertz (insert at the end).

Dynamic Database (3)

- retract(F): Remove a clause from the database.

- retractall(A): All rules for which the head unifies with A are removed from the database.

  In ECLiPSe it is retract_all. The call retractall(A) succeeds also when there are no facts/rules that match A.

- abolish(p, n): Remove the definition of p/n.

  Then the predicate is no longer defined at all. A call to the predicate would give an error.

- listing: Lists the dynamic database.

Dynamic Database (4)

Exercise:

- Define a predicate `next(N)`, that generates unique numbers, i.e. the first call returns 1, the second call returns 2, and so on.

    Independent of whether backtracking happens in between or not.

- Define a predicate `all_solutions` that works like `findall`.

    Of course, you should not use `findall` or `bagof`, but the dynamic database. For simplicity, you can assume that the goal does not call recursively `all_solutions`. You need the predicate `fail` that is logically false (triggers backtracking).

Contents

Input/Output (1)

Input/Output of Terms:

- write(t): Print term t (using operators).

    E.g. write(1+1) and write(+(1,1)) both print 1+1. The predicate write does not know how the term was originally written, it only gets the internal data structure as input. Normally also variable names are lost when the term is represented internally, therefore output variable names might appear strange (e.g. write(X) might print _G219, where 219 is probably a memory address).

- display(t): Print term t in standard syntax.

    E.g. display(1+1) prints +(1, 1).

- writeq(t): Print t, put atoms in '...' (if necessary).

    This guarantees that the term can be read again with read, see below.

Input/Output (2)

Input/Output of Terms, continued:

- write_canonical(*t*): Print *t* in standard syntax, and put atoms in '...' (if necessary).

  This is even safer than writeq for reading the term again, because the current operator declarations are not needed. New Prologs have a predicate write_term(*t*, *O*) that prints *t* with options *O*. Then write, display, etc. are abbreviations for write_term with certain options.

- nl: Print a line break.

- read(*X*): Read a term, bind *X* to the result.

  The input term must be terminated with "..⟨Newline⟩". At the end of the file, most Prolog systems return *X* = end_of_file. Together with operator declarations, read is already a quite powerful parser.

Input/Output (3)

Input/Output of Characters:

- put_code(C): Print character with ASCII-code C.

  In older Prolog versions (compatible to DEC-10 Prolog), this is simply called put. The newer put_code is contained in the ISO Standard.

- get_code(C): Read next character, unify C with its ASCII-code.

  At the end of file, C is set to −1. In older Prolog versions, this is called get0. The predicate get first skipped spaces, and then unified C with the next non-space character.

- peek_code(C): Unify C with ASCII-code of next input character without actually reading it.

Input/Output (4)

	0	1	2	3	4	5	6	7	8	9
0	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	␣	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{			}	~	DEL		

Input/Output (5)

Input/Output of Characters, continued:

- It is also possible to work with one-character atoms instead of ASCII-codes:

    - put_char(C): Print atom as character.

        E.g. to print a space one writes `put_char(' ')`.

    - get_char(C): Read next character, unify C with corresponding atom.

        E.g. if the user enters "a", `get_code(C)` returns C=97, whereas `get_char(C)` returns C=a. At the end of file, C is set to `end_of_file`.

    - peek_char(C): Unify C next input character (as atom) without actually reading it.

Input/Output (6)

Input/Output of Characters, continued:

- The above `get_*`-predicates normally wait for an entire line of input from the keyboard.

- In contrast to `read`, it is not necessary to finish the input with ".", Enter/Return suffices.

- Every Prolog system has a way to read characters without buffering, but that is system dependent.

    E.g. in SWI Prolog, use `get_single_char`. In GNU Prolog, use `get_key`.

Input/Output (7)

File Input/Output:

- In Prolog, open files are called "streams".

- Streams are actually a bit more general. Usually, the following is also supported (depending on the system):

    - keyboard input and screen output,

        These streams are called user_input and user_output.

    - pipes (for inter-process communication),

    - sockets (for network communication),

    - I/O from atom names and ASCII-code lists/strings.

Input/Output (8)

File Input/Output, continued:

- All of the above I/O predicates have also a version with an additional argument for a stream.

- E.g. write(S, t) prints term t to stream S.

- open(F, M, S): Opens file F in mode M (read, write, append, possibly update), and returns stream S.

  The file name F should be an atom. At least SWI-Prolog also supports pipe(C) with a command C. There is also open(F,M,S,O) that has in addition a list O of options, e.g. [type(binary)]. One can also use the option [alias(A)] to declare atom A as stream name which can be used in calls to write etc. (instead of the stream object S itself).

Input/Output (9)

File Input/Output, continued:

- Files open in binary mode must be read or written with
  put_byte/get_byte/peek_byte instead of the
  put_code/get_code/peek_code predicates.

  > The difference is that the Prolog system might do operating system
  > dependent translations for text files, e.g. map CR/LF to LF under Windows,
  > whereas binary files are read verbatim. At least GNU Prolog produces a
  > runtime error (exception) if one uses get_byte on a text file or vice versa.

- close(S): Close stream S.

  > There is also close(S, O) with options O. If an output file is not closed,
  > the buffer might not be written, and data is lost.

Input/Output (10)

File Input/Output, continued:

- flush_output/flush_output(S):
  Flush pending (buffered) output.

- at_end_of_stream/at_end_of_stream(S):
  Succeeds after the last character was read.

- set_input(S): Set the default input stream to S.

- set_output(S): Set the default output stream to S.

- get_input(S): Set S to the current input stream.

- get_output(S): Set S to the current output stream.

Input/Output (11)

Exercises:

- DEC-10 Prolog had a predicate tab(N) that printed N spaces. Please define it.

  Since tab is still contained in some Prologs, it might be necssary to use a different name, e.g. nspaces.

- Define a predicate calc that prints a prompt, reads an arithmetic expression (without variables), evaluates it, prints the result, and so on until the user enters "quit".

  You can assume that the user ends each input line with ".". Furthermore, you do not have to handle syntax errors.

Contents

Control (1)

- A, B: A and B (conjunction).

- A; B: A or B (disjunction).

    Conjunction binds stronger than disjunction ("`;`" has priority 1100, "`,`" has priority 1000). One can use parentheses if necessary. Disjunction is not strictly needed, one can use several rules instead.

- true: True (always succeeds).

- fail: False (always fails).

    Obviously, this can only be interesting with previous side effects (or the cut). Examples are shown in the next chapter.

- repeat: Always succeeds, also on backtracking.

    This can be defined as   repeat.   repeat :- repeat.

Control (2)

- !: Ignore all previous alternatives in this predicate activation (cut, see next Chapter).

  This means that no further rules for the same predicate will be tried, and no further solutions for all body literals to the left of the cut.

- A -> B_1 ; B_2: If A, then B_1, else B_2.

  This really means $(A$ -> $B_1)$; B_2. The arrow "->" has priority 1050, disjunction ";" has priority 1100.

- A -> B: If A, then B, else fail.

  This is equivalent to A, !, B.

- once(A): Compute only first solution for A.

Control (3)

- $\backslash+$ *A*: *A* is not provable (fails).

    This is called negation as failure. It is not the logical negation, because
    Prolog permits only to write down positive knowledge. Negation as failure
    behaves non-monotonically, whereas classical predicate logic is monotonic:
    If one adds formulas, one can at least prove everything that was provable
    earlier. Some Prologs also understand `not` *A*.

Prolog Environment (1)

- halt: Leave the Prolog system.

- abort: Stop the current Prolog program.

    Control returns to the top-level Prolog prompt. This predicate is not
    contained in the ISO standard.

- help(p/n): Show online manual for predicate p of arity n
  (not in all Prolog systems).

- shell(C, E): Execute the operating system
  command C, unify E with the exit status.

    This is not contained in the ISO standard. If the predicate is missing, look
    for system/1, unix/1, and shell/1. There might be many more
    predicates to give Prolog an operating system interface.

Prolog Environment (2)

- statistics: Display statistics, such as used CPU time, used size of various memory areas, etc.

  This predicate is not contained in the ISO Prolog standard. There might also be statistics/2 to query specific statistics.

- trace: Switch debugger on (in creep mode).

  Creep mode means step by step execution.

- spy(p/n): Set a breakpoint on predicate p of arity n.

- debug: Switch debugger on (in leap mode).

  Leap mode means that execution stops only at breakpoints.

- notrace/nodebug: Switch debugger off.