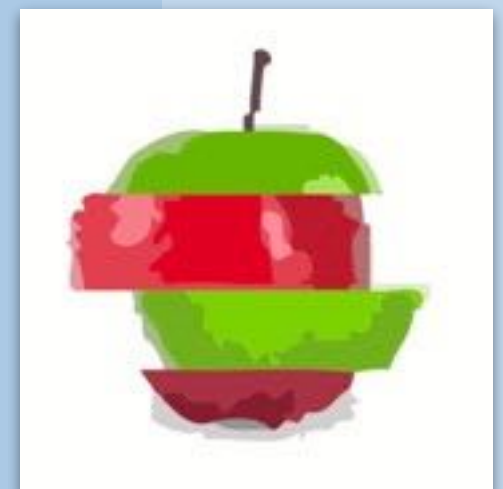


10. Logic Programming

Oscar Nierstrasz



Roadmap



- > Facts and Rules
- > Resolution and Unification
- > Searching and Backtracking
- > Recursion, Functions and Arithmetic
- > Lists and other Structures

References

- > Kenneth C. Loudon, *Programming Languages: Principles and Practice*, PWS Publishing (Boston), 1993.
- > Sterling and Shapiro, *The Art of Prolog*, MIT Press, 1986
- > Clocksin and Mellish, *Programming in Prolog*, Springer Verlag, 1981
- > Ciao Prolog, <http://ciao-lang.org>

Roadmap



- > **Facts and Rules**
- > Resolution and Unification
- > Searching and Backtracking
- > Recursion, Functions and Arithmetic
- > Lists and other Structures

Logic Programming Languages

- > ***What is a Program?***
 - > A program is a *database of facts* (axioms) together with a *set of inference rules* for proving theorems from the axioms.
 - > ***Imperative Programming:***
 - Program = Algorithms + Data
 - > ***Logic Programming:***
 - Program = Facts + Rules
 - or
 - Algorithms = Logic + Control
- *Kowalski*

Prolog was developed in Marseille in the early 1970s by Alain Colmerauer and colleagues. The design was influenced by PLANNER, an earlier language developed by Carl Hewitt in the late 60s.

One of the main goals of Prolog is to be able to write highly declarative programs that express intuitively logical queries, and to use a very powerful backtracking algorithm to answer those queries.

What is Prolog?

A Prolog program consists of *facts, rules, and questions*:
Facts are *named relations* between objects:

```
parent(charles, elizabeth).  
% elizabeth is a parent of charles  
female(elizabeth).  
% elizabeth is female
```

Rules are relations (goals) that can be *inferred* from other relations (subgoals):

```
mother(X, M) :- parent(X,M), female(M).  
% M is a mother of X  
% if M is a parent of X and M is female
```

Questions are *statements* that can be answered using facts and rules:

```
?- parent(charles, elizabeth).  
⇒ yes  
?- mother(charles, M).  
⇒ M = elizabeth <cr>  
yes
```

The basic paradigm is that there are facts, and rules to infer new facts.

A fact is expressed as a Prolog term: an identifier, possibly followed by parentheses containing a comma-separated list of further terms. Terms are pure syntax. It is up to the programmer to assign a semantics, for example:

```
parent(charles, elizabeth).
```

```
% elizabeth is a parent of charles
```

A rule consists of a head (a term) and a body (a list of terms), possibly containing upper-case variables.

```
mother(X, M) :- parent(X,M), female(M).
```

A question is simply a term (a goal), possibly containing variables.

```
parent(charles, elizabeth).
```

```
mother(charles, M).
```

A question can be answered either by a fact or a fact plus a combination of rules.

Horn Clauses

Both rules and facts are instances of Horn clauses, of the form:

$$A_0 \text{ if } A_1 \text{ and } A_2 \text{ and } \dots A_n$$

A_0 is the head of the Horn clause and “ A_1 and A_2 and ... A_n ” is the body

Facts are just Horn clauses without a body:

parent(charles, elizabeth)	if	True
----------------------------	----	------

female(elizabeth)	if	True
-------------------	----	------

mother(X, M)	if	parent(X,M)
	and	female(M)

Roadmap



- > Facts and Rules
- > **Resolution and Unification**
- > Searching and Backtracking
- > Recursion, Functions and Arithmetic
- > Lists and other Structures

Resolution and Unification

Questions (or goals) are answered by (i) *matching goals against facts or rules*, (ii) *unifying variables with terms*, and (iii) *backtracking when subgoals fail*.

If a subgoal of a Horn clause *matches the head* of another Horn clause, resolution allows us to *replace that subgoal by the body of the matching Horn clause*.

Unification lets us *bind variables to corresponding values* in the matching Horn clause:

```
mother(charles, M)
⇒ parent(charles, M) and female(M)
⇒ { M = elizabeth } True and female(elizabeth)
⇒ { M = elizabeth } True and True
```

If we pose the question:

```
mother(charles, M)
```

this matches no fact, but does match the rule:

```
mother(X, M) :- parent(X,M), female(M).
```

We replace the goal by the body, unifying the variables:

```
parent(charles,M) and female(M).
```

Now the first term matches a fact, so we simply to:

```
{M=elizabeth} True and female(elizabeth).
```

The next subgoal also matches a fact, so we terminate with:

```
{M=elizabeth} True and True
```

that is, the answer is True if M=elizabeth.

Prolog Databases

- > A Prolog database is *a file of facts and rules* to be “consulted” before asking questions:

```
female(anne).  
female(diana).  
female(elizabeth).
```

```
male(andrew).  
male(charles).  
male(edward).  
male(harry).  
male(philip).  
male(william).
```

```
parent(andrew, elizabeth).  
parent(andrew, philip).  
parent(anne, elizabeth).  
parent(anne, philip).  
parent(charles, elizabeth).  
parent(charles, philip).  
parent(edward, elizabeth).  
parent(edward, philip).  
parent(harry, charles).  
parent(harry, diana).  
parent(william, charles).  
parent(william, diana).
```

```
mother(X, M) :- parent(X,M), female(M).  
  
father(X, M) :- parent(X,M), male(M).
```

Simple queries

```
?- consult('royal').
```

```
⇒ yes
```

```
?- male(charles).
```

```
⇒ yes
```

```
?- male(anne).
```

```
⇒ no
```

```
?- male(mickey).
```

```
⇒ no
```

*Just another query
which succeeds*

NB: in ciao Prolog, use
`ensure_loaded/1`
instead of `consult/1`.

The `consult/1` query causes Prolog to try to load a file. If a query succeeds, Prolog answers “yes”. If it fails, Prolog answers “no” (`male(mickey)` fails because it cannot be answered from the database).

Prolog has a very primitive type system that just looks at the number of arguments to a term, so the “`consult`” predicate is known as “`consult/1`”, i.e., a term with one argument. A term with no argument is also called an *atom*.

An atom may be enclosed in single quotes to avoid confusion with variables (starting with upper case), or if it contains other special characters.

Queries with variables

You may accept or reject unified variables:

```
?- parent(charles, P).  
⇒ P = elizabeth <cr>  
yes
```

You may reject a binding to search for others:

```
?- male(X).  
⇒ X = andrew ;  
   X = charles <cr>  
yes
```

Use anonymous variables if you don't care:

```
?- parent(william, _).  
⇒ yes
```


If a query contains variables, Prolog will try to answer the query by proposing a possible binding for the variables. To accept the answer, enter a carriage return. To reject the answer and force Prolog to look for another answer, enter a semi-colon (“;”).

Unification

Unification is the process of instantiating variables by *pattern matching*.

1. *A constant unifies only with itself:*

```
?- charles = charles.  
⇒ yes  
?- charles = andrew.  
⇒ no
```

2. *An uninstantiated variable unifies with anything:*

```
?- parent(charles, elizabeth) = Y.  
⇒ Y = parent(charles, elizabeth) ? <cr>  
⇒ yes
```

3. *A structured term unifies with another term only if it has the same function name and number of arguments, and the arguments can be unified recursively:*

```
?- parent(charles, P) = parent(X, elizabeth).  
⇒ P = elizabeth,  
X = charles ? <cr>  
⇒ yes
```

Unification of terms is a recursive process, and loosely resembles pattern-matching in Haskell, except that variables may occur on both side of unified terms.

Constants (atoms) match only an equal constant or a variable.

$$\text{foo} = \text{foo}$$
$$\text{foo} = \text{X}$$

Variables match any kind of term.

$$\text{X} = \text{bar}$$

Structured terms must match in their name (the “functor”), and in their arity. Then, each sub-term must match recursively.

$$\text{foo}(\text{X}, \text{Y}) = \text{foo}(\text{a}, \text{b})$$

When a variable matches a term, that variable is globally replaces within the term where it appears. If a variable matches another variable, then the two variables are unified in both the left- and right-hand terms.

$$\text{foo}(\text{X}, \text{bar}) = \text{foo}(\text{V}, \text{V})$$

Evaluation Order

In principle, any of the parameters in a query may be instantiated or not

```
?- mother(X, elizabeth).
```

```
⇒ X = andrew ? ;
```

```
   X = anne ? ;
```

```
   X = charles ? ;
```

```
   X = edward ? ;
```

```
no
```

```
?- mother(X, M).
```

```
⇒ M = elizabeth,
```

```
   X = andrew ? <cr>
```

```
yes
```

As we shall see, it is not always the case that arbitrary arguments may be uninstantiated variables.

In each of the two examples, the query is evaluated in the same way: the query is matched against available facts and rules. The first to match causes the variables to be unified, and so on.

```
mother(X,M)
```

```
→ parent (X,M), female(M).
```

```
→ {X=andrew, M=elizabeth} parent(andrew, elizabeth),  
female(elizabeth)
```

```
...
```

Closed World Assumption

Prolog adopts a *closed world assumption* — whatever cannot be proved to be true, is assumed to be false.

```
?- mother(elizabeth,M) .
```

```
⇒ no
```

```
?- male(mickey) .
```

```
⇒ no
```

Roadmap



- > Facts and Rules
- > Resolution and Unification
- > **Searching and Backtracking**
- > Recursion, Functions and Arithmetic
- > Lists and other Structures

Backtracking

Prolog applies resolution in linear fashion, replacing goals left to right, and considering database clauses top-to-bottom.

```
father(X, M) :- parent(X,M), male(M).
```

```
?- debug_module(user).  
?- trace.  
?- father(charles,F).  
⇒ 1 1 Call: user:father(charles,_312) ?  
   2 2 Call: user:parent(charles,_312) ?  
   2 2 Exit: user:parent(charles,elizabeth) ?  
   3 2 Call: user:male(elizabeth) ?  
   3 2 Fail: user:male(elizabeth) ?  
   2 2 Redo: user:parent(charles,elizabeth) ?  
   2 2 Exit: user:parent(charles,philip) ?  
   3 2 Call: user:male(philip) ?  
   3 2 Exit: user:male(philip) ?  
   1 1 Exit: user:father(charles,philip) ?
```


Each Prolog implementation offers a different way to trace how it answers queries.

In ciao Prolog you must invoke `debug_module/1` and `trace/0`.

As it matches goals to facts and rules, Prolog may introduce uninstantiated variables (i.e., analogous to “fresh”, unbound names in the lambda calculus). This avoid confusions between different instantiations of the same rule.

To answer the query `father(charles,F)`, Prolog binds `X` in the rule with head `father(X,M)` to `charles` and `M` to a “fresh” name `_312`. This causes the subgoal `parent(charles,_312)` to be met successfully (*Exit*) with `parent(charles,elizabeth)`. `male(elizabeth)` fails, however, so Prolog *backtracks* to the last successfully met subgoal, and *retries*, this time succeeding with `parent(charles,philip)`. Now `male(philip)` succeeds. Since there are no more subgoals, the query succeeds with `_312` (and consequently `F`) unified with `philip`.

Comparison

The predicate = attempts to *unify* its two arguments:

```
?- X = charles.  
⇒ X = charles ?  
yes
```

The predicate == tests if the terms instantiating its arguments are *literally identical*:

```
?- charles == charles.  
⇒ yes  
?- X == charles.  
⇒ no  
?- X = charles, male(charles) == male(X).  
⇒ X = charles ?  
yes
```

```
?- X = male(charles), Y = charles, X \== male(Y).  
⇒ no
```

The predicate \== tests if its arguments are *not literally identical*:

Note that $=$ is perfectly symmetrical, so uninstantiated variables may occur on either side.

?- $X=Y$.

$Y = X$?

yes

?- $a(b,C) = a(B,c)$.

$B = b,$

$C = c$?

yes

Sharing Subgoals

Common subgoals can easily be factored out as relations:

```
sibling(X, Y)    :- mother(X, M), mother(Y, M),  
                  father(X, F), father(Y, F),  
                  X \== Y.  
  
brother(X, B)    :- sibling(X, B), male(B).  
uncle(X, U)      :- parent(X, P), brother(P, U).  
  
sister(X, S)     :- sibling(X, S), female(S).  
aunt(X, A)       :- parent(X, P), sister(P, A).
```

Disjunctions

One may define *multiple rules for the same predicate*, just as with facts:

```
isparent(C, P) :- mother(C, P).  
isparent(C, P) :- father(C, P).
```

Disjunctions (“or”) can also be expressed using the “;” operator:

```
isparent(C, P) :- mother(C, P); father(C, P).
```

Note that *same information can be represented in different forms* — we could have decided to express `mother/2` and `father/2` as facts, and `parent/2` as a rule. Ask:

Which way is it easier to *express and maintain* facts?

Which way makes it *faster to evaluate* queries?

Roadmap



- > Facts and Rules
- > Resolution and Unification
- > Searching and Backtracking
- > **Recursion, Functions and Arithmetic**
- > Lists and other Structures

Recursion

Recursive relations are defined in the obvious way:

```
ancestor(X, A) :- parent(X, A).  
ancestor(X, A) :- parent(X, P), ancestor(P, A).
```

```
?- debug_module(user).  
?- trace.  
?- ancestor(X, philip).  
⇒ + 1 1 Call: ancestor(_61,philip) ?  
   + 2 2 Call: parent(_61,philip) ?  
   + 2 2 Exit: parent(andrew,philip) ?  
   + 1 1 Exit: ancestor(andrew,philip) ?  
X = andrew ?  
yes
```

 *Will ancestor/2 always terminate?*

As with recursive functions, we must be careful to always be able to reach the base case. Note that Prolog will visit rules strictly in the order in which they appear in the program, which means that normally the *non-recursive cases should be listed first*.

By the same reasoning, in a rule with multiple subgoals, the recursive subgoal often appear last (though this is not always necessary).

Recursion ...

```
?- ancestor(harry, philip).  
⇒ + 1 1 Call: ancestor(harry,philip) ?  
   + 2 2 Call: parent(harry,philip) ?  
   + 2 2 Fail: parent(harry,philip) ?  
   + 2 2 Call: parent(harry,_316) ?  
   + 2 2 Exit: parent(harry,charles) ?  
   + 3 2 Call: ancestor(charles,philip) ?  
   + 4 3 Call: parent(charles,philip) ?  
   + 4 3 Exit: parent(charles,philip) ?  
   + 3 2 Exit: ancestor(charles,philip) ?  
   + 1 1 Exit: ancestor(harry,philip) ?  
yes
```

 What happens if you query `ancestor(harry, harry)`?

Clearly `harry` is not an ancestor of himself, but how much work will Prolog have to do before it determines this?

What about this query?

```
ancestor(harry, genghis) .
```

Evaluation Order

Evaluation of recursive queries is sensitive to the order of the rules in the database, and when the recursive call is made:

```
anc2(X, A) :- anc2(P, A), parent(X, P).  
anc2(X, A) :- parent(X, A).
```

```
?- anc2(harry, X).  
⇒ + 1 1 Call: anc2(harry,_67) ?  
   + 2 2 Call: anc2(_325,_67) ?  
   + 3 3 Call: anc2(_525,_67) ?  
   + 4 4 Call: anc2(_725,_67) ?  
   + 5 5 Call: anc2(_925,_67) ?  
   + 6 6 Call: anc2(_1125,_67) ?  
   + 7 7 Call: anc2(_1325,_67) ? abort  
{Execution aborted}
```

In `anc2 / 2`, the recursive case appears *before* the base case, leading to an infinite recursion.

What about this version?

```
anc3(X, A) :- parent(X, A).
```

```
anc3(X, A) :- anc3(P, A), parent(X, P).
```

And this one?

```
anc4(X, A) :- parent(X, P), anc4(P, A).
```

```
anc4(X, A) :- parent(X, A).
```

Do both of these also lead to infinite recursion?

Failure

Searching can be controlled by *explicit failure*:

```
printall(X) :- X, print(X), nl, fail.  
printall(_).
```

```
?- printall(brother(_,_)).  
⇒ brother(andrew,charles)  
   brother(andrew,edward)  
   brother(anne,andrew)  
   brother(anne,charles)  
   brother(anne,edward)  
   brother(charles,andrew)
```

Note how failure is used to force Prolog to try all alternatives, before it finally reaches the trivial case, which *follows* the failing case.

Try:

```
printall(ancestor(_,_) ).
```

Cuts

The cut operator (!) *commits* Prolog to a particular search path:

```
parent(C,P) :- mother(C,P), !.  
parent(C,P) :- father(C,P).
```

Cut says to Prolog:

“This is the right answer to this query. If later you are forced to backtrack, please do not consider any alternatives to this decision.”

The *cut* operator allows the programmer to control how Prolog backtracks. It essentially commits Prolog to not backtrack before the cut.

In this particular case, we are say, “*if you can establish that P is the mother of C , then please do not backtrack; it is pointless to also check if P is the father of C .*”

This cut is purely an *optimization*, since, if `mother(C, P)` succeeds, then `father(C, P)` will *necessarily fail*.

The optimization is useful in the case that `parent(C, P)` is a subgoal of a more complex rule that fails for some other reason. We can then save Prolog from pointlessly backtracking through `parent/2`.

If we remove the cut, *the semantics will not change*. This is therefore a “green cut”.

Red and Green cuts

- > A green cut does not change the semantics of the program. It just eliminates useless searching.
- > A red cut changes the semantics of your program. If you remove the cut, you will get incorrect results.

Negation as failure

Negation can be implemented by a *combination of cut and fail*:

```
not(X) :- X, !, fail.           % if X succeeds, we fail
not(_).                        % if X fails, we succeed
```

Here is the canonical example of a red cut.

Suppose we pose the query:

`not (a=a) .`

We match the first rule, and subgoal `a=a` succeeds, so we reach the cut, and then we fail. Since there is a cut, we do not backtrack, and Prolog answers “no”, which is the correct negation of `a=a`.

Conversely, if we query:

`not (a=b) .`

then `a=b` fails in the first rule, before the cut, and we proceed to the second rule, which succeeds. This yields “yes”, which is the correct negation of `a=b`.

Note that, without the cut, *all queries* have the answer “yes”. The cut is therefore *not an optimization*, and is a “red cut”.

Changing the Database

The Prolog database can be *modified dynamically* by means of `assert/1` and `retract/1`:

```
rename(X,Y) :-    retract(male(X)),
                  assert(male(Y)), rename(X,Y).
rename(X,Y) :-    retract(female(X)),
                  assert(female(Y)), rename(X,Y).
rename(X,Y) :-    retract(parent(X,P)),
                  assert(parent(Y,P)), rename(X,Y).
rename(X,Y) :-    retract(parent(C,X)),
                  assert(parent(C,Y)), rename(X,Y).
rename(_,_) .
```

Please note that `assert` and `retract` are not recommended for most normal Prolog programs. They should *only* be used in situations where you need to *dynamically modify the database* of facts.

In this example we rename a royal by retracting his or her name from the `male/1`, `female/1` and `parent/2` terms in the database, and reasserting those terms with the new name. The `rename/2` predicate is called recursively, always failing when there is nothing more to retract, and then finally reaching the base case at the end.

Note that this is an exception to the normal practice that recursive rules should list the base case first!

Changing the Database ...

```
?- male(charles); parent(charles, _).  
⇒ yes  
?- rename(charles, mickey).  
⇒ yes  
?- male(charles); parent(charles, _).  
⇒ no
```

NB: With some Prologs, such predicates must be declared to be dynamic:

```
:- dynamic male/1, female/1, parent/2.
```

Normally Prolog compiles the database to an efficient internal form that does not permit post hoc modifications. To enable such modifications, the `dynamic` declaration forgoes the compilation step.

Functions and Arithmetic

Functions are relations between expressions and values:

```
?- X is 5 + 6.  
⇒ X = 11 ?
```

This is syntactic sugar for:

```
is(X, +(5,6))
```


Mathematically a function $f : A \rightarrow B$ is just a relation over a domain A and a co-domain B , i.e., a subset $f \subset A \times B$, where

$$(a, b_1) \in f \wedge (a, b_2) \in f \Rightarrow b_1 = b_2.$$

All Prolog predicates are relations, and some of them are functions.

The predicate `is/2` is a function from its second argument to its first. The first is normally an uninstantiated variable, and the second is a Prolog term representing an arithmetic expression.

Note that we can instruct Prolog to interpret any term with two arguments as an infix operator (as is the case with `is/2`). We can also interpret terms with one argument as prefix or postfix operators.

See for example the Ciao Prolog operator documentation:

https://clip.dia.fi.upm.es/~clip/Projects/ASAP/Software/Ciao/CiaoDE/ciao_html/ciao_40.html

Defining Functions

User-defined functions are written in a *relational style*:

```
fact(0,1).  
fact(N,F) :- N > 0,  
             N1 is N - 1,  
             fact(N1,F1),  
             F is N * F1.
```

```
?- fact(10,F).
```

```
⇒ F = 3628800 ?
```

Roadmap



- > Facts and Rules
- > Resolution and Unification
- > Searching and Backtracking
- > Recursion, Functions and Arithmetic
- > **Lists and other Structures**

Lists

Lists are *pairs of elements and lists*:

<i>Formal object</i>	<i>Cons pair syntax</i>	<i>Element syntax</i>
$\cdot(a, [])$	$[a []]$	$[a]$
$\cdot(a, \cdot(b, []))$	$[a [b []]]$	$[a, b]$
$\cdot(a, \cdot(b, \cdot(c, [])))$	$[a [b [c []]]]$	$[a, b, c]$
$\cdot(a, b)$	$[a b]$	$[a b]$
$\cdot(a, \cdot(b, c))$	$[a [b c]]$	$[a, b c]$

Lists can be *deconstructed* using cons pair syntax:

?- $[a, b, c] = [a | X].$
 $\Rightarrow X = [b, c]?$

Note the similarity to Haskell lists. In both cases, lists consist either of an empty list (i.e., `[]`) or a non-empty list consisting of a head and a tail. As in Haskell, Prolog introduces a more convenient, readable syntax to construct lists.

Again, as in Haskell, in order to do something with a list, in Prolog you must *deconstruct* a list by pattern-matching.

The main differences between lists in Haskell and Prolog is that Prolog lists may contain uninstantiated variables.

The formal syntax is rarely used or needed, but you can easily verify that it is actually supported:

```
?- [a] = .(a, []).
```

```
yes
```

Pattern Matching with Lists

```
in(X, [X | _]).  
in(X, [ _ | L]) :- in(X, L).
```

```
?- in(b, [a,b,c]).  
⇒ yes
```

```
?- in(X, [a,b,c]).  
⇒ X = a ? ;  
   X = b ? ;  
   X = c ? ;  
no
```

To pattern-match a non-empty list, you should almost always use the syntax `[H | T]`. This will match `H` to head and `T` to the tail, which may be either an empty or a non-empty list. If you try to use the element-syntax, then you will only match a list with a certain specific number of elements, e.g., `[X, Y]` matches only lists with exactly two elements.

The predicate `in/2` does not need to handle the case where the second argument is an empty list, since an element `X` can only be a member of a non-empty list. However we need to handle two cases: where `X` is the head of the list, and where it might be in the tail.

Note the use of the anonymous variable `_`, when we don't care what its value is.

Pattern Matching with Lists ...

Prolog will automatically introduce new variables to represent unknown terms:

```
?- in(a, L).  
⇒ L = [ a | _A ] ? ;  
   L = [ _A , a | _B ] ? ;  
   L = [ _A , _B , a | _C ] ? ;  
   L = [ _A , _B , _C , a | _D ] ?  
yes
```


Here Prolog is just blindly applying the rules. It first supposes that `a` is the head of a list, and the tail is “whatever”. If we reject that, then it supposes `a` is the second element, and so on. In each case, “whatever” is represented by uninstantiated variables representing the other unknown elements or the tail.

Inverse relations

A carefully designed relation can be used in many directions:

```
append([ ],L,L).  
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

```
?- append([a],[b],X).
```

```
⇒ X = [a,b]
```

```
?- append(X,Y,[a,b]).
```

```
⇒ X = [] Y = [a,b] ;
```

```
   X = [a] Y = [b] ;
```

```
   X = [a,b] Y = []
```

```
yes
```

Just for fun, try the following:

```
append ( A , B , C ) .
```

Exhaustive Searching

Searching for permutations:

```
perm([ ],[ ]).  
perm([C|S1],S2) :- perm(S1,P1),  
                    append(X,Y,P1),           % split P1  
                    append(X,[C|Y],S2).
```

```
?- printall(perm([a,b,c,d],_)).  
⇒ perm([a,b,c,d],[a,b,c,d])  
   perm([a,b,c,d],[b,a,c,d])  
   perm([a,b,c,d],[b,c,a,d])  
   perm([a,b,c,d],[b,c,d,a])  
   perm([a,b,c,d],[a,c,b,d])
```

A permutation of a list $[C \mid S1]$ is $S2$, if we can insert head C anywhere into $P1$, where $P1$ is a permutation of $S1$.

We insert C into $P1$ by using append to split $P1$ into arbitrary X and Y , and then gluing together X , C and Y .

Limits of declarative programming

A declarative, but hopelessly inefficient sort program:

```
ndsort(L,S) :-                perm(L,S),
                              issorted(S).









issorted([ ]).
issorted([ _ ]).
issorted([N,M|S]) :- N <= M,
                    issorted([M|S])..
```

Of course, efficient solutions in Prolog do exist!





OK, this is a (not very funny) Prolog joke. To sort a list L , we generated an arbitrary permutation and then test if it is sorted.

This is clearly a declarative program that works, but it has worse complexity than the clumsiest insert-sort algorithm.

What you should know!

-  *What are Horn clauses?*
-  *What are resolution and unification?*
-  *How does Prolog attempt to answer a query using facts and rules?*
-  *When does Prolog assume that the answer to a query is false?*
-  *When does Prolog backtrack? How does backtracking work?*
-  *How are conjunction and disjunction represented?*
-  *What is meant by “negation as failure”?*
-  *How can you dynamically change the database?*

Can you answer these questions?

-  *How can we view functions as relations?*
-  *Is it possible to implement negation without either cut or fail?*
-  *What happens if you use a predicate with the wrong number of arguments?*
-  *What does Prolog reply when you ask `not(male(X)).` ?
*What does this mean?**



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>