Lists in Prolog

How?

```
[trace]  ?- mylength([a,b,c],3).
   Call: (8) mylength([a, b, c], 3) ?
   Call: (9) mylength([b, c], _L193) ?
   Call: (10) mylength([c], _L212) ?
   Call: (11) mylength([], _L231) ?
   Exit: (11) mylength([], 0) ?
^  Call: (11) _L212 is 0+1 ?
^  Exit: (11) 1 is 0+1 ?
   Exit: (10) mylength([c], 1) ?
^  Call: (10) _L193 is 1+1 ?
^  Exit: (10) 2 is 1+1 ?
   Exit: (9) mylength([b, c], 2) ?
^  Call: (9) 3 is 2+1 ?
^  Exit: (9) 3 is 2+1 ?
   Exit: (8) mylength([a, b, c], 3) ?
true
```

Lists in Prolog

```
% mylength(?L,?N) iff N is the length of list L
mylength([],0).
mylength([_|T],N):-mylength(T,NT), N is NT+1.

?- mylength([a,b,c], L).
L = 3.

?- mylength([X,Y], L).
L = 2.

?- mylength(X, 3).
X = [_G226, _G229, _G232] ; <-- infinite run
```

Lists in Prolog

```
[trace]  ?- mylength(X, 1).
...
X = [_G373] ;
   Redo: (9) mylength(_G374, _L196) ?
   Call: (10) mylength(_G377, _L215) ?
   Exit: (10) mylength([], 0) ?
^  Call: (10) _L196 is 0+1 ?
^  Exit: (10) 1 is 0+1 ?
   Exit: (9) mylength([_G376], 1) ?
^  Call: (9) 1 is 1+1 ?
^  Fail: (9) 1 is 1+1 ?
```

Lists in Prolog

```
  Redo: (10) mylength(_G377, _L215) ?
   Call: (11) mylength(_G380, _L227) ?
   Exit: (11) mylength([], 0) ?
^  Call: (11) _L215 is 0+1 ?
^  Exit: (11) 1 is 0+1 ?
   Exit: (10) mylength([_G379], 1) ?
^  Call: (10) _L196 is 1+1 ?
^  Exit: (10) 2 is 1+1 ?
   Exit: (9) mylength([_G376, _G379], 2) ?
^  Call: (9) 1 is 2+1 ?
^  Fail: (9) 1 is 2+1 ?
   Redo: (11) mylength(_G380, _L227) ?
...
```

We'll learn how to fix this later.

Logic Programming vs. Prolog

What happens if we change the order of rules:

```
% mylength(?L,?N) iff N is the length of list L
mylength([_|T],N):-mylength(T,NT), N is NT+1.
mylength([],0).

?- mylength([a,b,c], 3).
true.

?- mylength([a,b,c], N).
N = 3.

?- mylength(X, 2).
ERROR: Out of local stack
```

Not very “declarative”.

Negation as Failure

No equivalent of logical negation in Prolog:

- Prolog can only assert that something is true.
- Prolog **cannot** assert that something is false.
- Prolog can assert that the given facts and rules do not allow something to be proven true.

Negation as Failure

Assuming that something unprovable is false is called **negation as failure**.
(Based on a **closed world assumption**.)
The goal \+(G) (or `not G`) succeeds whenever the goal G fails.

```
?- member(b, [a,b,c]).
true

?- \+ member(b, [a,b,c]).
false.

?- not(member(b, [a,b,c])).
false.

?- not(member(b,[a,c])).
true.
```

Negation as Failure

Example: Disjoint Sets

```
overlap(S1,S2) :- member(X,S1),member(X,S2).

disjoint(S1,S2) :- \+(overlap(S1,S2)).

?- overlap([a,b,c],[c,d,e]).
true
?- overlap([a,b,c],[d,e,f]).
false
?- disjoint([a,b,c],[c,d,e]).
false
?- disjoint([a,b,c],[d,e,f]).
true
?- disjoint([a,b,c],X).
false   %<---------Not what we wanted
```

Negation as Failure

```
overlap(S1,S2) :- member(X,S1),member(X,S2).
disjoint(S1,S2) :- \+(overlap(S1,S2)).

[trace]  ?- disjoint([a,b,c],X).
   Call: (7) disjoint([a, b, c], _G293) ? creep
   Call: (8) overlap([a, b, c], _G293) ? creep
   Call: (9) lists:member(_L230, [a, b, c]) ? creep
   Exit: (9) lists:member(a, [a, b, c]) ? creep
   Call: (9) lists:member(a, _G293) ? creep
   Exit: (9) lists:member(a, [a|_G352]) ? creep
   Exit: (8) overlap([a, b, c], [a|_G352]) ? creep
   Fail: (7) disjoint([a, b, c], _G293) ? creep
false
```

Negation as Failure

Proper use of Negation as Failure

`not(G)` works properly only in the following cases:

1. When `G` is fully instantiated at the time prolog processes the goal `not(G)`.

   (In this case, `not(G)` is interpreted to mean "goal `G` does not succeed".)

2. When all variables in `G` are unique to `G`, i.e., they don't appear elsewhere in the same clause.

   (In this case, `not(G(X))` is interpreted to mean "There is no value of `X` that will make `G(X)` succeed".)

Negation as Failure

```
woman(jane).
woman(marilyn).
famous(marilyn).
loves(john,X) :- woman(X), famous(X).
hates(john,X) :- \+ loves(john,X).
```

There are infinitely many women that John hates, not just Jane:

```
?- hates(john,jane).
true
?- hates(john,susan).
true
?- hates(john,betty).
true
...
```

Negation as Failure

```
woman(jane).
woman(marilyn).
famous(marilyn).
loves(john,X) :- woman(X), famous(X).
hates(john,X) :- \+ loves(john,X).
```

Plus John hates many things:

```
?- hates(john,pizza).
?- hates(john,john).
```

We say that the rule `hates` is not **safe**. Solution:

```
hates(john,X) :- woman(X), \+ loves(john,X).
```

`woman(X)` is called a **guard** — it protects from making unwanted inferences.

Execution of Prolog Programs

- **Unification**: variable bindings.
- **Backward Chaining/
  Top-Down Reasoning/
  Goal-Directed Reasoning**:
  Reduces a goal to one or more subgoals.
- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

Cut

The goal "!", pronounced "cut" always succeeds immediately.
It has an important side effect: Once it is satisfied, it disallows either:

- backtracking back over the cut, or
- backtracking and applying a different clause of the same predicate to satisfy the present goal.

You can think of satisfying cut as making a commitment both

- to the variable bindings we've made during the application of this rule, and
- to this particular rule itself.

The cut goal trims the derivation tree of all other choices on the way back up to and including the point in the derivation tree where the cut was introduced into the sequence of goals.

Cut

Cut can be used to improve the efficiency of search by reducing the search space.
For example, when two predicates are mutually exclusive:

```
q(X) :- even(X), a(X).
q(X) :- odd(X), b(X).
```

With cut

```
q(X) :- even(X), !, a(X).
q(X) :- odd(X), b(X).
```

Cut

Cut can remove unwanted answers. Consider the Family Database:

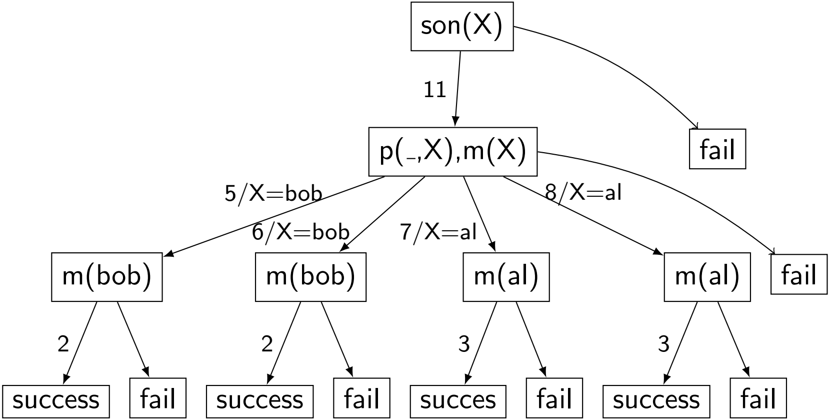
```
1. male(charlie). 2. male(bob).
3. male(albert). 4. female(eve).

5.  parent(charlie,bob).
6.  parent(eve,bob).
7.  parent(charlie,albert).
8.  parent(eve,albert).

% son(?X) iff X is a son
11. son(X):-parent(_,X),male(X).

?- son(X).
X = bob ;
X = bob ;
X = albert ;
X = albert.
```

Cut


son(X)
11
fail
p(_,X),m(X)
5/X=bob
6/X=bob
7/X=al
8/X=al
m(bob)
m(bob)
m(al)
m(al)
fail
2
2
3
3
success
fail
success
fail
succes
fail
success
fail

Cut

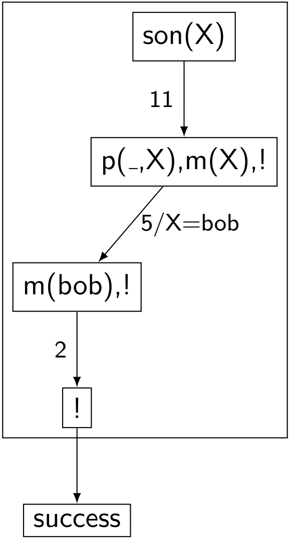
We want to rewrite the rule `son` so that it does not generate duplicate answers.

```
son(X):-parent(_,X),male(X),! .
```

```
?- son(X).
X = bob.
```

What about `albert`?

Cut



Cut

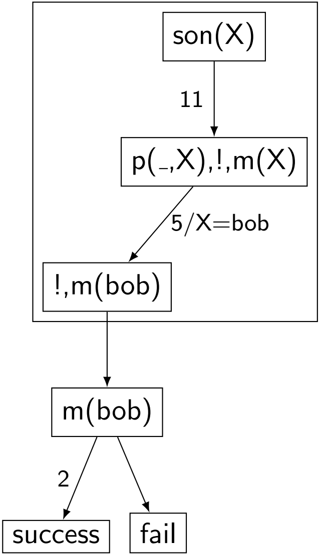
Try again:

```
son(X):-parent(_,X),!, male(X).
```

```
?- son(X).
X = bob.
```

Aghrr..

Cut



Cut

Think:

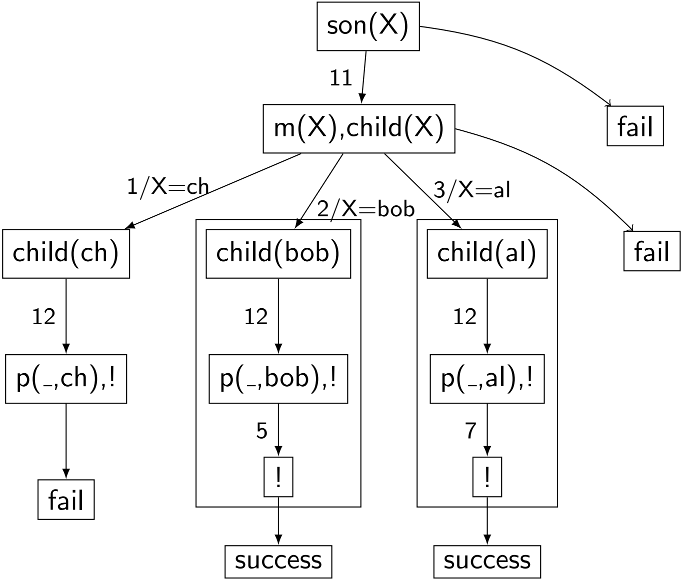
- Any male is a potentially good answer, so we want to try all of them: can't put "cut" after "male" in the same rule.
- If a male has 2 parents, we only want to list him once as the answer: want to put "cut" after "parent".

Result:

```
son(X):-male(X), child(X).
child(X):-parent(_,X),!.

?- son(X).
X = bob ;
X = albert.
```

Cut


son(X)
11
fail
m(X),child(X)
1/X=ch
2/X=bob
3/X=al
fail
child(ch)
child(bob)
child(al)
12
12
12
p(_,ch),!
p(_,bob),!
p(_,al),!
5
7
fail
!
!
success
success

Cut

What about `sibling`?

```
sibling(X,Y):-parent(P,X),parent(P,Y).
```

```
?- sibling(bob, X).
X = bob ;
X = albert ;
X = bob ;
X = albert ;
false.
```

Cut

Think:

- Any two people in the database are potentially good answers, so we want to try all of them: can't put "cut" in a rule after `X` and/or `Y` is instantiated.
- If 2 people share 2 parents, we only want to list them once as the answer: want to put "cut" after 2 "parent" rules.

Result:

```
sibling(X,Y):-person(X),person(Y),commonparent(X,Y).
person(X):-male(X).
person(X):-female(X).
commonparent(X,Y):-parent(P,X),parent(P,Y),!.
```

Cut

```
sibling(X,Y):-person(X),person(Y),commonparent(X,Y).
person(X):-male(X).
person(X):-female(X).
commonparent(X,Y):-parent(P,X),parent(P,Y),!.

?- sibling(bob, X).
X = bob ;
X = albert ;
false.
```

Cut

Finally, we don't want X to be a sibling of X.

```
sibling(X,Y):-\+(X=Y),person(X),person(Y),
              commonparent(X,Y).
person(X):-male(X).
person(X):-female(X).
commonparent(X,Y):-parent(P,X),parent(P,Y),!.
```

```
?- sibling(bob,X).
false
```

What went wrong?

Cut

Solution:

```
sibling(X,Y):-person(X),person(Y),\+(X=Y),
              commonparent(X,Y).
person(X):-male(X).
person(X):-female(X).
commonparent(X,Y):-parent(P,X),parent(P,Y),!.

?- sibling(bob,X).
X = albert ;
false.
```

Execution of Prolog Programs

- **Unification**: variable bindings.

- **Backward Chaining/**
  Top-Down Reasoning/
  Goal-Directed Reasoning:
  Reduces a goal to one or more subgoals.

- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

Reasoning

- **Bottom-up** (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

  e.g., Suppose the fact B and the rule $A \leftarrow B$ are given. Then infer that A is true.

- **Top-down** (or backward) reasoning: starting from the query, apply the rules in reverse, attempting only those lines of inference that are relevant to the query.

  e.g., Suppose the query is A, and the rule $A \leftarrow B$ is given. Then to prove A, try to prove B.

Reasoning

Backtracking plus reduction gives Prolog the built-in ability to perform **top-down search**. This naturally models program execution in imperative languages (main program calls subprograms, which call sub-subprograms, etc.).

Some languages (e.g. Coral programming language) do **bottom-up** search. Bottom-up search is also often used in natural language processing.

Reasoning

Bottom-up search has:

- very early access to the axioms of inference, which
- often results in greater speed (because variables are bound early, which creates opportunities for failure). But
- it is not goal-oriented — many useless facts may be derived along the way.

Top-down search is:

- very goal-oriented, but
- it often has problems with termination and efficiency, as
- it may explore many lines of reasoning that fail

The two methods are logically equivalent. There are many hybrid search strategies, too. The best combination depends on the empirical domain being modelled.

Reasoning

Examples:

```
a:-b;c.
b:-d;e.
c:-f;g.
g.
?-a.
```

But:

```
c:-bN.
b1:-a1.
...
bN:-aN.
a1.
...
aN.
?-c.
```

Nondeterministic Programming

Nondeterminism is powerful for defining and implementing algorithms.

Intuitively, a nondeterministic machine can choose its next operation correctly when faced with several alternatives.

Nondeterminism can be simulated/approximated by Prolog's sequential search and backtracking. Nondeterminism cannot truly be achieved.

Towers of Hanoi

Setup: 3 pegs ("left", "centre", "right"). In the initial state one peg (let's say the "left" peg) has N rings on it, stacked from largest to smallest.

Task: Move N disks from the left peg to the right peg using the centre peg as an auxiliary holding peg. At no time can a larger disk be placed upon a smaller disk.

Towers of Hanoi

```
% move(+N,?X,?Y,?Z) iff it is possible to move N disks
% from peg X to peg Y using peg Z as an auxiliary
% holding peg.
%  As a side effect, print out the sequence of moves.
move(1,X,Y,_) :-
        write('Move top disk from '),
        write(X),
        write(' to '),
        write(Y),
        nl.
move(N,X,Y,Z) :-
        N>1,
        M is N-1,
        move(M,X,Z,Y),
        move(1,X,Y,_),
        move(M,Z,Y,X).
```

Nondeterministic Programming

Can you think of other problems / games where you can specify the rules of the game in Prolog and Prolog will solve it for you?