Logic Programming

Resit 2021

Answers by
David De Potter

Question 1: The first hurdle

(a) The two atoms are different, so they don't unify.

(b) An atom can't be unified with a complex term.

(c) A complex term of arity 1 can't be unified with a complex term of arity 2, even if their functors are the same.

(d) The complex terms have the same functor and arity, but the atoms don't match, so unification fails.

(e) The complex terms don't have the same functor, so they can't be unified.

(f) The instantiations for the variable Ann are incompatible. We can't instantiate Ann to 0, if the variable was already instantiated to 1.

(g) The same reasoning as in f. The instantiations for the variable Ann are incompatible.

(h) The term on the right-hand side is an atom as it is enclosed by single quotes. An atom can't be unified with a complex term.

(i) A list can't be unified with a complex term.

(j) A list can't be unified with an atom.

(k) A list of two items can't be unfied with a list of one.

(l) The atoms bee and bea are different and so unification fails.

(m) There's no way to instantiate the variable BEA such that [BEA] can be unified with the atom bea.

(n) The atom bea can't be unified with the list [bea].

(o) The atoms bea and bee don't match, so unification fails.

(p) Both sides feature a list of length three, but the one on the left has cee as its third item, whereas the other one has [cee]. A list can't be unified with an atom.

(q) In order to unify both sides we have to start by instantiating Ann to [ann]. But this means that the list on the left-hand side has length four, and can never be unified with the list on the other side which has length two. So unification fails.

(r) A complex term can't be unified with a list. The use of the functor *sort* is only meant to cause confusion.

(s) A complex term can't be unified with an atom. Unification would actually succeed if we turned this into: `member(X,[ape,bee,cee]),X = bee.` Prolog's answer would then be: `X=bee`, indicating that unification is possible iff the variable X is instantiated to the atom bee.

(t) Both sides have a complex term with arity 2, and we can instantiate X to d, but a list of length three can never be unified with one of length four.

Question 2: Cutting it fine

The predicate pos/2 simply collects all the positive numbers from the input list. Zeros and negative numbers are ignored. For example, if we were to perform the query `?- pos([0,1,-3,5,-1,4],Out)`, then we'd get the answer: `Out=[1,5,4]`.

cls 1: This recursive clause verifies if the current head of the input list is positive and if so, this item is unified with the head of the output list.

cls 2: The second recursive clause verifies if the current head of the input list is non-positive, i.e. zero or negative, and if so, then the number is ignored by leaving the output list unaltered.

cls 3: This is the base clause. Recursion stops when the input list can be unified with the empty list and the last piece of the output list is set to the empty list. A Prolog list is built using the list constructor '.'/2. Here the output list is constructed through the recursive calls and once the base clause is reached, the list is completed by setting the last piece to the empty list.

Adding cuts at different positions.

cut pos 1: This is a red cut. It's a bad position because it makes the program produce wrong answers. Now the predicate will only succeed if the input list contains positive numbers only. This is because all other branches are continually discarded, even before it's clear if X is positive or not, for as long as the input list is non-empty. If at any point a non-positive number is found, then the predicate simply fails.

cut pos 2: This is a green cut. It doesn't change the meaning of the predicate and enhances efficiency by discarding the open branches as soon as it's confirmed that $X > 0$. So, this is a good position for the cut.

cut pos 3: This is a bad position for the cut because it's completely redundant and actually decreases the program's efficiency by making the predicate non-tail-recursive. Now there will be a long and completely useless goal stack of cuts to go through (goals which all succeed, because the cut always does) once the base clause of the predicate pos/2 is reached and recursion has stopped.

cut pos 4: This is a green cut. It's a good position for the cut because it doesn't change the meaning of the predicate (it still does what it was intended to do) and it enhances efficiency. The latter is accomplished by discarding the open branch for the last option of the predicate once it's clear that the current head of the list is non-positive and the current state could be matched to an input list that's not empty: therefore there's no need to leave the third option open.

Question 3: Count noses

```
% count/2 reduces a list by adding frequencies indicating how
% many times this item was repeated consecutively in the input list

count([H|T],C) :- count(T,[H,1],C).

count([],[H,C],[H:C]).

count([H|T],[H,C1],C) :- C2 is C1+1, count(T,[H,C2],C).

count([H|T],[H1,C1],[H1:C1|T2]) :- H \= H1, count(T,[H,1],T2).
```

Alternatively, we can use a cut and remove the condition from the last rule, but then the order of the clauses can't be changed anymore.

```
count([H|T],C) :- count(T,[H,1],C).

count([],[H,C],[H:C]).

count([H|T],[H,C1],C) :- !, C2 is C1+1, count(T,[H,C2],C).  % red cut

count([H|T],[H1,C1],[H1:C1|T2]) :- count(T,[H,1],T2).
```

Question 4: The young ones

```
% The given database

age(ann, 20). age(joe, 44).
age(bob, 40). age(min, 27).
age(cai, 30). age(ned, 27).
age(deb, 42). age(pat, 36).
age(edo, 24). age(tod, 56).
```

(a) older/2

```
% true iff X is older than Y

older(X,Y) :- age(X,A1), age(Y,A2), A1 > A2.
```

(b) hundred/1

```
% true iff the argument unifies with a list of at
% least two persons whose ages add up to 100

hundred(G) :-
  findall(age(N,A),age(N,A),Ps),
  sum(100,Ps,G),G=[_,_|_].

sum(0,_,[]).
sum(Sum,Ps,[N|Ns]):-
  select(age(N,A),Ps,Rem),
  NewSum is Sum-A, NewSum >= 0,
  sum(NewSum,Rem,Ns).
```

The above solution works fine, but it also lists each permutation of a solution as a new one if we ask for all possible solutions by performing the query `?- hundred(X)`. These permutations are in fact duplicates. The question doesn't mention this as a requirement, but if we only want to get unique sets of people whose ages add up to 100, we can rewrite the predicate as below. The idea here is to put each group with age sum equalling 100 in a larger list of lists, and sort each group that is added, so that setof/3 can then recognize and discard any duplicates once the list is complete.

```
hundred(G):-
  findall(age(N,A),age(N,A),Ps),
  setof(Sorted,
        Ns^(sum(100,Ps,Ns),sort(Ns,Sorted)),
        Groups),
  member(G,Groups), G=[_,_|_].

sum(0,_,[]).
sum(Sum,Ps,[N|Ns]):-
  select(age(N,A),Ps,Rem),
  NewSum is Sum-A, NewSum >= 0,
  sum(NewSum,Rem,Ns).
```

(c) middle/1

```
% true iff the person is not the youngest and not the oldest

middle(X) :- age(X,A), \+ youngest(X), \+ oldest(X).

youngest(X) :- age(X,A), \+ (age(_,B), B < A).

oldest(X) :- age(X,A), \+ (age(_,B), B > A).
```

(d) youngish/1

```
% is true iff X is a person that is not the youngest and
% not the oldest and the number of people older than X is
% greater than those younger than X

youngish(X) :- middle(X), olderthan(X,N1),
               youngerthan(X,N2), N1 > N2.

% determines the number of people older than X

olderthan(X,N) :- findall(_,older(_,X),L), length(L,N).

% determines the number of people younger than X

youngerthan(X,N) :- findall(_,older(X,_),L), length(L,N).
```

Question 5: Taking the train

```
% The given database

train(groningen,delfzijl).
train(groningen,leeuwarden).
train(groningen,assen).

direct(X,Y) :- train(X,Y).
direct(X,Y) :- train(Y,X).

route(X,Y,[X,Y]) :- direct(X,Y).
route(X,Y,[X|R]) :- route(Z,Y,R), direct(X,Z).
```

(a) Three facts.

(b) Four rules.

(c) Seven clauses.

(d) There are three predicates defined: train/2, direct/2, route/3.

(e) There are no dynamic predicates.

(f) There's one recursive predicate: route/2.

(g) The database doesn't contain any tail-recursive predicates: route/2 is recursive, but the recursive call doesn't occur at the end of the rule.

(h) By using a semicolon to denote a disjunction:
direct(X,Y) :- train(X,Y); train(Y,X).

Question 6: The negative ones

```
% The given database

negative(L,N):- L=[], N=[].
negative([X|L],[X|N]):- X < 0, !, negative(L,N).
negative([X|L],N):- X > 0, negative(L,N).
```

From figure 1 it's clear that Prolog's answer to the query will be `Neg=[-3,-2]`.

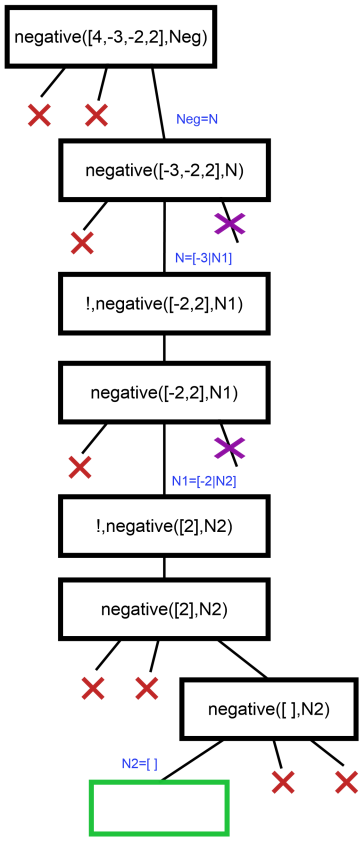

Figure 1: Search tree for *?- negative([4,-3,-2,2],Neg).*