Logic Programming

Exam 2023

Answers by
David De Potter

Question 1: Elfstedentocht

(a) This is the empty list. It has no elements.

(b) This list contains a single item: the empty list. The tail is empty, so we can rewrite the list as [[ ]].

(c) This list has five items. Its tail is empty again.

(d) The list contains five items. Its head counts as one item and its tail features four items.

(e) This list has 3 or more items, since the tail is a variable and can have any number of items.

(f) This list has four items, no matter what X is instantiated to.

Question 2: Unification

(a) This unification succeeds, as the variable Tooth can be bound to the atom eye.

(b) This query fails since the functors of the complex terms are different.

(c) This query succeeds, because both complex terms have the same functor and arity 1. Prolog's answer will be Eye=Tooth.

(d) The query fails because the items in the list can't be unified: the complex term 1+1 can't be unified with the atom 2.

(e) The query fails because there's a complex term on the right-hand side, whereas the other side has just a list. Using *sort* here is only meant to cause confusion.

(f) Both sides have the same complex term with the functor *sort* and arity 1. The two variables can be instantiated to the two atoms, so that Prolog will respond with: Tooth=eye, Eye=tooth.

Question 3: An odd split

```
% The given database

split([],[],[]).

split([X|L0],[X|L1],L2) :- X > 0, !, split(L0,L1,L2).

split([X|L0],L1,L2) :- X = 0, !, split(L0,L1,L2).

split([X|L0],L1,[X|L2]) :- X < 0, !, split(L0,L1,L2).
```

(a) Split/3 is a recursive predicate: it consists of three recursive rules on lines 2-4, and a base case on the first line. The rules are recursive as they all have a recursive call, i.e. in the rule body the predicate calls itself. The base case stops the recursion. Split/3 is also tail-recursive because the recursive call occurs at the very end of each rule.

(b) The predicate split/3 splits a given list into two lists: one containing all the positive numbers and another containing all the negative ones. Any zeros are ignored. This means that Prolog will respond with: `X=[4,9,6,4], Y=[-5,-2].`

(c) Prolog will respond with: `X=[4,4], Y=[-5,-2].`

(d) cut 1: This is a green cut. Once it's confirmed that $X > 0$, the branches for the other alternatives are cut away. This is efficient because there's no use in checking the other options if we already know that X is positive.

cut 2: This is a green cut for the same reason as the first one. Once it's confirmed that $X = 0$, the open branches for the other options are discarded.

cut 3: This is a blue cut. It doesn't do anything. However, if we change the order of the clauses so that this rule is not the last one anymore, then this blue cut turns into a green one. Thus, the blue cut makes the program more declarative as it allows to change the order of the clauses without any loss of efficiency.

Since this database has only green and blue cuts, we can leave them out without any issues. Leaving out the green cuts will only make the program less efficient.

(e) If we change the last rule by removing the condition $X < 0$, then the second cut turns red: now it's no longer possible to leave that cut out without ending up with a wrong program. If left out, the program will execute both the options on lines 3 and 4 for any zeros in the input list, whereas the last option is only intended for negative numbers.

Question 4: Mean

```
% computes the mean of a list of numbers
% number of predicates used is 2: mean/2 and mean/4

mean(L, Mean) :- mean(L, 0, 0, Mean).

mean([H|T], Sum, Len, Mean):-
  NewSum is Sum+H, NewLen is Len+1,
  mean(T, NewSum, NewLen, Mean).

mean([], Sum, Len, Mean) :- Len > 0, !, Mean is Sum/Len.

mean([], _, _, 0). % special case if input is empty list
```

Question 5: Marsupial

```
% the given database
word([c,a,r]).
word([r,e,s,t]).
word([s,c,a,r,y]).
word([s,t,i,n,g]).
word([i,n,t,e,r,e,s,t,i,n,g]).
word([i,n,t,e,r,e,s,t,i,n,g,l,y]).

% definition of marsupial/1
marsupial(L) :- word(W), L \= W, subseq(W,L), !.

% subseq/2 succeeds iff its 1st argument
% is a subsequence of its 2nd argument
subseq([],_).

subseq([H|T],[H|M]) :- !, subseq(T,M).  % red cut, chars are equal

subseq(W,[_|T]) :- subseq(W,T). % chars are different
```

Question 6: Pick

```
% The given database

pick(X,X).
pick(X,[X|L],L) :- !.
pick(K,[X|L],[Y|M]) :- pick(X,Y), pick(K,L,M).
```

(a) Three clauses.

(b) There are two rules.

(c) There is one fact.

(d) The database defines two predicates: pick/2, pick/3.

(e) See figures 1, 2, 3 for the search trees.

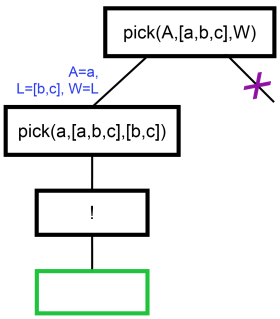

Figure 1: Query i

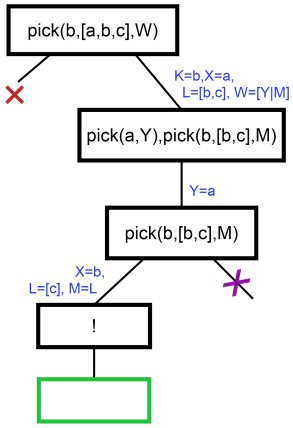

Figure 2: Query ii

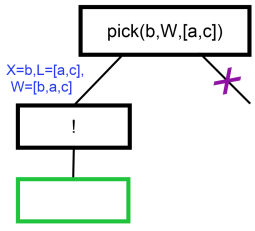

Figure 3: Query iii

Question 7: Cryptarithmetic

```
solve :- select(O, [0,1,2,3,4,5,6,7,8,9], L1),
         select(T, L1, L2),
         select(G, L2, L3),
         select(U, L3, _),
         TO is T*10 + O,
         GO is G*10 + O,
         OUT is O*100 + U*10 + T,
         OUT is TO + GO,
         write('TO  = '), writeln(TO),
         write('GO  = '), writeln(GO),
         write('OUT = '), writeln(OUT).

:- solve.
```

The program yields the following output:

```
TO  = 21
GO  = 81
OUT = 102
```

Question 8: Forever young

```
% The given database

dob('Mick', 1943).
dob('Charlie', 1941).
dob('Ronnie', 1947).
dob('Paul', 1942).
```

(a) Definition using negation

```
% eldest/1 succeeds iff its argument is the
% oldest person in the database

eldest(X) :- dob(X,A1), \+ (dob(_,A2), A2 > A1).
```

(b) Definition using setof/3

```
eldest(X) :- setof(A, N^dob(N,A), Ages),
             maxlist(Ages,Max), dob(X,Max).

% maxlist/2 determines the maximum
% number in its 1st argument
maxlist([H|T],Max) :- maxlist(T,H,Max).

maxlist([],Max,Max).

maxlist([H|T],Curr,Max) :- H > Curr, !, maxlist(T,H,Max).

maxlist([_|T],Curr,Max) :- maxlist(T,Curr,Max).
```