Departamento de Ciência de Computadores - FCUP
First Test of Logic Programming
(Duration: 2h)

Date: October 24th, 2016

NB: These are possible solutions. Remember that each one may have a different way of solving the same problem. Although I suggest a call pattern for each predicate, the call pattern may change depending on the query. For example, in problem #1, nth_element/3, I suggest the first argument to be output, but it could be input. Solutions are in blue.

1) Find the Nth element of a list where elements are indexed from 1. For example, the query:

```
?- nth_element(X,[a,b,c,d,e],3).
```

Should return: `X = c`

```
% nth_element(-Element,+List,+Position).
nth_element(X,[X|Xs],0).
nth_element(X,[Y|Ys],N) :-
   N1 is N - 1,
   nth_element(X,Ys,N1).
```

Possible queries:

```
nth_element(X,[1,2,4],3).
nth_element(3,[1,2,4],3).
nth_element(4,[1,2,4],3).
nth_element(4,[1,2,4],X).
```

Try all of the queries above for this program. Do they all succeed? If not, how would you change this program to succeed for all queries?
(I gave full score to everyone that wrote a program that passed the first three queries)

2) 2. Given a list of N sublists, for example,

```
[[a,b,c],[a,b,d],[a],[a,b,d,e,f]]
```

Write a program that returns a list with all elements and their respective frequencies. In the example above, the result should be: `[[a,4],[b,3],[c,1],[d,2],[e,1],[f,1]]`.

```
% freq(+List,-ListFreq).
freq(L,Result) :-
    flatten(L,L1),
    count(L1,Result).

count([],[]).
count([X|Xs],[[X,CountX]|L]) :-
    del_and_count(X,Xs,1,CountX,NewXs), % remove all X from Xs and returns
    count(NewXs,L).                     % count for X and new Xs
                                        % without X
```

```
del_and_count(_,[],C,C,[]).
del_and_count(X,[X|Xs],C,CountX,NewXs) :- % X is found, count + 1, X
    C1 is C + 1, !,                       % is not included in the output list
    del_and_count(X,Xs,C1,CountX,NewXs).
del_and_count(X,[Y|Ys],C,CountX,[Y|NewXs]) :- % X is not found,
    del_and_count(X,Xs,C,CountX,NewXs).       % Y needs to be included in the output list
```

Why do I need the cut in the third clause of `del_and_count/5`?

3) Write a program that accepts as input a text that describes a polynomial expression and returns a Prolog term that represents its derivative. For example, the text "`3x2  4`"+ should return `6 * x` (it is a Prolog term, not a string or a list of ASCII codes).
I gave full score to anyone that wrote something as simple as the code below (not exactly what is requested).

```
:- op(50,fy,x).

deriv(A * x N + Term, B * x M) :-
   B is A * N,
   M is N - 1,
   \+ M = 1, !.
deriv(A * x N + Term, B * x) :-
   B is A * N.
```

Why do I need a cut in the first clause?

4) Given the two implementations of reverse below (e.g., `reverse([1,2],[2,1])`), which one is correct? Are both correct? Explain your reasoning and the differences between the two programs, taking into account efficiency issues. (Just explaining how the two programs execute will not add to your score in this question).

```
Implementation 1:

reverse([],[]).
reverse([X|L1],L2) :-
     reverse(L1,RL1),
     append(RL1,[X]).

Implementation 2:

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

After fixing the mistake of the first program where `append(PL1,[X])` should read `append(PL1,[X],L2)`, the second implementation is much faster than the first one, given that the first one calls recursive predicates twice. In other words, reverse will be called as many times as the length of the input list, and when it finished it starts a new set of recursions again as many times as the length of the input

list to add each element to the final reversed list. The second implementation executes only as many times as the length of the input list.

5) Write a program that can delete an element from an ordered binary tree.

```
% removes internal nodes as well as leaves, and rearrange the tree
del(X,tree(X,L,R,_),_) :- % node X is a leaf, return an empty tree
   var(L), var(R), !.
del(X,tree(X,L,R,_),L) :- % node X has a left subtree, returns left subtree
   nonvar(L), var(R), !.
del(X,tree(X,L,R,_),R) :- % node X has a left subtree, returns left subtree
   var(L), nonvar(R), !.
del(X,tree(X,tree(Y,L1,R1,IY),tree(Z,L2,R2,IZ),_IX),tree(Y,L1,NewR1,IY)) :-
   insert(R1,tree(Z,L2,R2,IZ),NewR1), !. % node X has both subtrees
                                         % need to rearrange the tree
del(X,tree(Y,L,_R,_I),tree(Y,NewTree,_R,_I)) :-
   X @< Y,
   del(X,L,NewTree).
del(X,tree(Y,_L,R,_I),tree(Y,_L,NewTree,_I)) :-
   X @> Y,
   del(X,R,NewTree).

insert(R1,Subtree,Subtree) :- % subtree that is to be rearranged is empty
   var(R1).
insert(tree(X,L,R,IX),Subtree,tree(X,L,Subtree,IX)) :-
   var(R). % insert Subtree on the right of leaf node
insert(tree(X,L,R,IX),Subtree,tree(X,L,NewSubtree,IX)) :-
   nonvar(R),
   insert(R,Subtree,NewSubtree).

%% Queries:
%% ?- del(a,tree(d,tree(c,tree(a,_,_,ia),tree(b,_,_,ib),ic),_,id),T).
%% T = tree(d,tree(c,_A,tree(b,_B,_C,ib),ic),_D,id) ? ;
%% false.
%%
%% ?- del(n,
%%      tree(l,
%%           tree(g,
%%                tree(d,
%%                     tree(c,
%%                          tree(a,_,_,ia),
%%                          tree(b,_,_,ib),
%%                         ic),
%%                     tree(f,
%%                          tree(e,_,_,ie),
%%                          _,
%%                          if),
%%                     id),
%%                tree(j,
%%                     tree(i,_,_,ii),
%%                     tree(k,_,_,ik),
%%                     ij),
%%                ig),
```

```
%%            tree(q,
%%                 tree(n,
%%                      tree(m,_,_,im),
%%                      tree(o,_,tree(p,_,_,ip),io),
%%                      in),
%%                 tree(s,
%%                      tree(r,_,_,ir),
%%                      tree(t,_,tree(u,_,_,iu),it),
%%                      is),
%%                 iq),
%%            il),L).

%% L = tree(l, tree(g, tree(d, tree(c, tree(a,_A,_B,ia),
   %% tree(b,_C,_D,ib),ic), tree(f, tree(e,_E,_F,ie),_G,if),id),
   %% tree(j,tree(i,_H,_I,ii), tree(k,_J,_K,ik),ij),ig),
   %% tree(q,tree(m,_L,tree(o,_M, tree(p,_N,_O,ip),io),im), tree(s,
   %% tree(r,_P,_Q,ir), tree(t,_R, tree(u,_S,_T,iu),it),is),iq),il) ?

%% yes
```

Why do I need all cuts in this program?

6) Suppose we have the following facts:

```
teaches(dr_fred, history).        studies(alice, english).
teaches(dr_fred, english).  studies(angus, english).
teaches(dr_fred, drama).  studies(amelia, drama).
teaches(dr_fiona, physics).  studies(alex, physics).
```

Why Prolog answers “no” (“false”) to the query:

```
?- teaches(dr_fred, Course), !, studies(Student, Course).
```

Because after succeeding `teaches(dr_fred, Course)`, with `Course=history`, and failing the predicate `studies/2`, because there is no fact defined for `studies(?,history)`, the cut operator prevents any other solution for `teaches/2` to be found.