Assignment 4

Due: Friday, 02.06.2017, 15:59 via Git

For help, contact alp-staff@lists.iai.uni-bonn.de (staff only) or alp-course@lists.iai.uni-bonn.de (staff and participants).

Start working on the exercises early enough so that you can contact your tutor in time if you have problems. Don't expect your tutor to be available at midnight or during weekends!

Submit results into the folder "assignment04/" of the git repository of your group.

For each task (except Task 1), submit your implemented predicate as a file named "taskN.pl". At the bottom of the file add a comment containing console output that shows all results of a successful test run of your predicate.

Task 1. *Application of substitution* (3 Points)

Write down the term resulting from applying the respective substitution, or explain why the substitution cannot be applied. For example, write " f(X){X←1} ≡ f(1)".

a) f(X,Y) {X←'Z'} ≡ f('Z',Y)

b) g(X,Y) {X←2, Y← g(X)} ≡ g(2,g(2))

c) h(X,Y) {X←h(Z,Y), Y← h(Z), Z← 3} ≡ h(h(3,h(3)),h(3))

Task 2. *Linear list* (11,5 points)

Implement a predicate "linear(+NestedList, ?LinearList)" that succeeds whenever LinearList is a list whose elements may themselves be arbitrarily deeply nested lists and LinearList contains all elements of NestedList in the same order but without any nesting. For instance,

```
?-linear ([1,[2,3,[a,[b],c]]], [1,2,3,a,b,c]).
```

should succeed but

```
?-linear ([1,[2,3,[a,[b],c]]], [1,2,3,a,c,b]).
```

should fail.

Tip: You may use the predefined predicate `append(A,B,AB)` to implement your version of linear/2. Except for that, your solution must be self-contained.

Sample solution:

```
%%linear(+NestedList, ?LinearList)
%
% Succeeds if is Arg1 is a list (whose elements may themselves
% be arbitrarily deeply nested lists) and Arg2 contains all elements
% from Arg1 in the same order but without any nesting.
% ← for full comment, like above there are  // 1,5 points

linear([],[]).                             // 1 point
linear([H|T], [H|TFlatt]):-                // 1 point
    not(is_list(H)),                       // 1 point
    linear(T,TFlatt).                      // 1 point
linear([H|T], Flatt):-                     // 1 point
    linear(H, Hflatt),                     // 1 point
    linear(T, Tflatt),                     // 1 point
    append(Hflatt, Tflatt, Flatt).         // 1 point

?- linear( [[a,c],b], [c,a,b]).

is_list([]).                               // 1 point
is_list([_|T]) :- is_list(T).              // 1 point
```

Note: Instead of `not(is_list(H))` you could have used the built-in predicate `atomic(H)`. However, the task was to implement yourself everything except append/2, not to use built-in predicates.

Task 3. *Grouping consecutive list elements* (11 Points)

Write a predicate that groups consecutive repeated elements of a list into sublists. If a list contains non-consecutive repeated elements they should be placed in separate sublists. Example:

?- group([1,1,1,1,2,c,c,1,1,d,e,e,e,e],X).

X = [[1,1,1,1],[2],[c,c],[1,1],[d],[e,e,e,e]]

Sample solution:

```
%% group(+L1,?L2)
%
% The list L2 is obtained from the list L1 by grouping repeated occurrences
% of elements into separate sublists. An element occurring only once is
% grouped into a singleton sublist (= a sublist with just one element).
%                                  ← Complete comment 1 point
group([],[]).                                                % 1 p
group([Head|Tail],[HeadList|NestedTail]) :-                  % 1 p
       collect_duplicate_occurrences(Head,Tail,TailRest,HeadList),     % 1 p
       group(TailRest,NestedTail).                                % 1 p


%% collect_duplicate_occurrences(+Elem,+List,?RestOfList,?ElemList)
%
% ElemList contains Elem and all leading copies of Elem from List.
% RestOfList is the list that remains from List when all leading
% copies of Elem are removed and transfered to ElemList.
% Note that if one were to allow L1 to contain free variables one would
% have to use \== instead of \= in the body of collect_duplicate_occurrences/4.
%                                                     % ← Complete comment 1 p
collect_duplicate_occurrences(Elem,[],   [],   [Elem]   ).        % 1 p
collect_duplicate_occurrences(Elem,[H|T],[H|T],[Elem]   )    :-   % 1 p
      Elem \== H.                                                 % 0,5 p
collect_duplicate_occurrences(Elem,[H|T],Trest,[Elem|Erest]) :-   % 1 p
      Elem == H,                                                  % 0,5 p
      collect_duplicate_occurrences(H,T,Trest,Erest).             % 1 p
```

Alternative solution, more compact (does not need a helper) but is harder to extend (see next task):

```
group4([], []).

% ← Complete comment (like before)                                % 1 p

% Input list starts with two consecutive elements:
% Create output list that starts with a nested list that contains the two
consecutive
% elements and has as rest the result of continuing to look for H in T.
%
group4([H,H|T], [[H,H|Y]| XT]) :-                                 % 2 p
      group4([H|T], [[H|Y]|XT]).                                  % 2 p

% Input list starts with an element that does not appear
% as the top of the tail. Note: The tail could be the empty
% list or a list whose first element is different from H!
% In this case, we only put H in an own nested list and
% continue with the tail T.
group4([H|T], [[H]|XT]) :-                                        % 3 p
      [H|_] \= T,                                                 % 2 p
      group4(T, XT).                                              % 1 p
```

Task 4. *Grouping consecutive list elements* (4,5 Points)

Modify your predicate from Task 4 so that it does not put an element into a sublist if there is no consecutive repeated element. Example:

?- group([1,1,1,1,2,c,c,1,1,d,e,e,e,e],X).

X = [[1,1,1,1],2,[c,c],[1,1],d,[e,e,e,e]]

Tip: In both cases (Task 4 and 5) the essential question is “How can your predicate remember whether it had seen the same element in the previous step?”

```
Sample solution:


%% group(+L1,?L2)
%
% The list L2 is obtained from the list L1 by grouping repeated occurrences
% of elements into separate sublists. Elements occurring only once are left
% untouched.
% Note that if one were to allow L1 to contain free variables one would
% have to use == instead of unification (and \== instead of \=in the
% collect_duplicate_occurrences/4 predicate).

group([],[]).
group([Head|Tail],[NestedHead  /* <-0,5 p */ |NestedTail]) :-
       collect_duplicate_occurrences(Head,Tail,TailRest,HeadList),
       eliminate_singleton_lists(HeadList, NestedHead),          % 1 p
       group(TailRest,NestedTail).


%% collect_duplicate_occurrences(+Elem,+List,?RestOfList,?ElemList)
%
% ElemList contains Elem and all leading copies of Elem from List.
% RestOfList is the list that remains from List when all leading
% copies of Elem are removed and transfered to ElemList.
collect_duplicate_occurrences(Elem,[],   [],   [Elem]   ).
collect_duplicate_occurrences(Elem,[H|T],[H|T],[Elem]   )    :- Elem \= H.
collect_duplicate_occurrences(Elem,[H|T],Trest,[Elem|Erest]) :- Elem == H,
     collect_duplicate_occurrences(H,T,Trest,Erest).


%% eliminate_singleton_lists(+X,?Y)
%
% If X is a list of just one element, Y is that element
% otherwise Y is the same list as X.                                 %1

eliminate_singleton_lists([E], E).              % Eliminate singleton list    %1
eliminate_singleton_lists([E1,E2|R], [E1,E2|R]).  % Do nothing for longer lists %1
```

Task 5. *Slice of a list (11 Points)*

Given a list L and two indices, I and J, the slice of L from I to J is the list containing the elements of L from position I to position J (both limits included). Start counting the elements with 1.

Example:

```
?- L =[a,b,c,d,e,f,g,h,i,k], I=3, J=7, slice(L, I,J,Slice).
Slice = [c,d,e,f,g]
```

```
%% slice(?L, +I, +J, ?Slice)
%
% Slice is the list of the elements of L between
% index I and index J (both included).

slice([H|_],1,1,[H]).

slice([H|Tail],1,J,[H|Ys]) :-
   J > 1,
   J1 is J - 1,
   slice(Tail,1,J1,Ys).

slice([_|Tail],I,J,Slice) :-
   I > 1,
   I1 is I - 1,
   J1 is J - 1,
   slice(Tail,I1,J1,Slice).
```

Explanations of the above clauses (numbered by clause number):

1. The slice of L from 1 to 1 is the list containing just the head of L.

2. The slice of L from 1 to J contains the head of L as head and the J-1 elements from the tail of L, that is the slice of the tail from 1 to J-1

3. For I>1, the slice of L from I to J is the slice of the tail of L from I-1 to J-1.