Assignment 5

Due: Friday, 16.06.2017, 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 your implemented predicates as a file named "assignment05/solutions.pl" in the Git
repository of your group. Add a file "assignment05/testRuns.txt" showing the console
output of a session in which you test that **each** solution works for the provided input data
and some queries that represent sensible test cases. If no input data is provided in the text
of the task, create some sensible input data. If input data is represented as facts, include
them into the "solutions.pl" file and add suitable comments.

Task 1. *Linearizing lists using accumulators* (6 Points)

In assignment 5 you developed a version of linearizing a list that looked like the following:

```prolog
linearize([],[]) .

linearize([X|L1],[X|L2] ) :- not( is_list(X) ),
                             linearize(L1,L2).

linearize([X|L1],L4     ) :- linearize(X,L2),
                             linearize(L1,L3),
                             append(L2,L3,L4).
is_list([]).
is_list([_|T]) :- is_list(T) .
```

Your task is to write a version of linearization

linearize_acc(+**L**,?**Res**)

that does not need the expensive call to "append/3". It should use a helper predicate

"linearize_acc(+**L**, +**Acc,** ?**Res**)"

whose additional parameter **Acc** acts as an accumulator of the already linearized part of the
list. New elements should always be inserted in front of the accumulated intermediate result
(not appended at the end).

Tip 1: Start by determining the proper initialization value for the accumulator argument and
then work out how you can "grow" it as described above.

```
/**
 * linearize_acc(+NestedList, ?FlatList)
 *
 * Succeeds whenever Arg1 is a list whose elements may be
 * arbitrarily deeply nested lists and Arg2 contains all
 * elements inside Arg1 in the same order but without any nesting.
 *
 * Uses linear_acc/3 which itself uses an accumulator to remember
 * all already visited elements in the proper order.
 */
linearize_acc(L,Res) :- linear_acc(L,[],Res).
linear_acc([],Acc,Acc).
linear_acc([X|L],Acc,[X|Lnew]) :-  var(X),
                                    linear_acc(L,Acc,Lnew).
linear_acc([X|L],Acc,[X|Lnew]) :-  nonvar(X),
                                    not(is_list(X)),
                                    linear_acc(L,Acc,Lnew).
linear_acc([X|L],Acc,Lnew)     :-  nonvar(X),
                                    is_list(X),
                                    linear_acc(L,Acc,Lhelp),
                                    linear_acc(X,Lhelp,Lnew).

?- linear_acc([1,X,2,[4,[],5],[6],7],Res).
Res = [1, X, 2, 4, 5, 6, 7].
```

Task 2.   *Cuts in linearization?* (3 Points)

Consider whether the predicate linearize_acc/3 from the previous task offers the opportunity for using cuts. If you think it does, insert a cut / cuts at the proper place(s) and add a brief explanation why you did it. Otherwise, explain why you think this is not a good case for using cuts.

```
%% linearize_acc(+NestedList, ?FlatList)
%
% Succeeds whenever Arg1 is a list whose elements may be
% arbitrarily deeply nested lists and Arg2 contains all
% elements inside Arg1 in the same order but without any nesting.
%
% Uses linear_acc/3 which itself uses an accumulator to remember
% all already visited elements in the proper order.
linearize_acc(L,Res) :- nonvar(L), linear_acc(L,[],Res).

linear_acc([],Acc,Acc).
linear_acc([X|L],Acc,[X|Lnew]) :-  var(X),
                                    !,      % ← avoid the redundant nonvar(X) test in the next clauses
                                    linear_acc(L,Acc,Lnew).
linear_acc([X|L],Acc,[X|Lnew]) :-  not(is_list(X)),
                                    !,      % ← avoid the redundant is_list(X) test in the next clause
                                    linear_acc(L,Acc,Lnew).
linear_acc([X|L],Acc,Lnew)     :-  linear_acc(L,Acc,Lhelp),
                                    linear_acc(X,Lhelp,Lnew).
```

← 1 Point for each correct cut and 1 point for each correct explanation of what it does.

Task 3. *Accumulators: Performance evaluation* (3 Points)

Compare the runtime of the naive and accumulator-based version of linearization (see previous task) by running the following test code 10 times and averaging the results for each version.

Submit:

- the implementation of the two versions of linearization that you compared,
- the operating system (name, version) and machine (CPU, GHz, RAM) that you used to run the benchmark
- the console output of the call below, for each of the 10 test runs
- and the average inferences and CPU time for each of the two predicate versions:

```
?- make_list(150,5,L),
   time( linearize_acc (L,_) ),
   time( linearize(L,_) ).
```

The output will have the following structure:

```
% ... inferences, ... CPU in ... seconds (100% CPU, ... Lips)
% ... inferences, ... CPU in ... seconds (100% CPU, ... Lips)
L = ... the first elements of the generated test list...
```

Here, „inferences" is the number of resolution steps performed, „CPU" is the CPU time in milliseconds and „Lips" is the abbreviation for „Linear Inferences per Second" (inferences / CPU * 1000).

Tip: The above test call uses the following helper predicates:

```
%% make_list(+Elems,+NestedElems,?List) is det
%
% Arg3 is a list with Arg1 random elements that are either
% integers or lists created with make_list/3.
% Arg 2 is the length of nested lists (at every level
% of nesting).
%
make_list(0,_,[]) :- !.
make_list(I,IS,[E|T]) :-
    make_head(IS,E), I_1
    is I-1,
    make_list(I_1,IS,T).
```

```
%% make_head(+L,?Res) is det
%
% Res is either
%    - a random integer value between 0 and 100000 or
%    - a list of length L that can itself have nested
%      sublists of length L.
% The choice between the two options is made randomly
% with a probability of 20% (1 of 5) for the list case.
%

make_head(I,E) :-
    NestIt is random(5),
    make_list_or_int(NestIt,I,E).

make_list_or_int (1,I,E) :- !, make_list(I,I,E).
make_list_or_int (N,_,E) :- E is random(100000).
```

Sample solution:

The results differ widely because the lists on which the flattening is tested can be very different (because they were randomly generated). However, the accumulator version is much faster and needs fewer inferences. The simple version often runs out of global stack.

Task 4. *Replace list elements* (3 Points)

Implement the predicate replace(+OldList, +OldElem, +NewElem, ?NewList). It should succeed whenever NewList is the list that can be obtained from OldList by replacing each occurrence of OldElem by NewElem.

Sample solution:
```
%% replace(+OldList, +OldElem, +NewElem, ?NewList).
%
% Succeed whenever NewList is the list that can be obtained from OldList by
% replacing EACH occurrence of OldElem by NewElem.

replace([],_,_,[]).

replace([Eold|Rest],Eold,Enew,[Enew|NewRest]) :-    % Replace
    replace(Rest,Eold,Enew,NewRest).

replace([X|Rest],Eold,Enew,[X|NewRest]) :-          % Keep head
    not(X==Eold),
    replace(Rest,Eold,Enew,NewRest).
```

Task 5.   *Replace facts* (3 Points)

Implement the predicate replace(+OldFact, +NewFact). It should succeed if and only if OldFact existed. When it succeeds, it should have the side-effect of deleting the first occurrence of OldFact and adding NewFact to the set of clauses in the program.

Sample solution:

```
%% replace replace(+OldFact, +NewFact)
%
% Succeed if and only if at least one clause whose head unifies with
% oldFact exists. Then the  first clause whose head unifies with
% Oldfact will be deleted and a copy of newfact will be added at
% the end.

replace(A, B) :-retract (A),assert(B).
```

For the solution to work properly, we assume that the arguments passed at run-time represent predicates that are declared as "dynamic". Otherwise, the attempt to assert or retract clauses would throw an exception.

Example of the above case, including the displayed error message …

```
isFatherOf(c, a).
isFatherOf(c, b).

?- retract(isFatherOf(c, a)).
 ERROR: [Thread pdt_console_client_0_Default Process] retract/1: No
 permission to modify static procedure `isFatherOf/2'
 ERROR: [Thread pdt_console_client_0_Default Process]
```