Assignment 6

Due: Friday, 17.06.2016, 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.

Submit your implemented predicates as a file named “assignment06/solutions.pl” in the Git repository of your group. Add a file “assignment06/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:

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

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

linearize([X|L1],L4     ) :- is_list(X),
                              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.

Task 2. *Accumulators: Performance evaluation* (2 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:

```
?- 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). Hand in the above lines for all 10 test runs along with the average inferences and CPU time for each of the two predicate versions.

Tip: The above test 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).
```

The meaning of the ! in the make_list_or_int/3 predicate will be explained in the next lecture. If you are curious to understand what happens read the part of Chapter 5 that has not been presented yet.

Task 3. *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.

Task 4. *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 OldFact and adding NewFact to the set of clauses in the program.

Tip: Solve this task after the next lecture or read ahead about “Data Representation via Facts”.