Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III
Prof. Dr. A. B. Cremers

Advanced Logic Programming
Summer Semester 2011
Dr. Günter Kniesel

Assignment 5

Due: Sunday, 15.05.2011, 23:59:59 via SVN

Submit your solution for this assignment into your group's SVN as an **Eclipse Project** named "assignment05".
Programming exercises should be submitted as **Prolog files** in this project.
Theoretical exercises should be submitted as **PDF files** in this project.

Task 1. *Shifting lists* (3 Points)

In task 4.4 you developed a first version of a linearised list that mostly looked like the following "linear_simple/2":

```
linear_simple([],[]) .
linear_simple([[]|L],L2     ) :- linear_simple(L,L2).
linear_simple([X|L1],[X|L2] ) :- atomic(X),
                                 linear_simple(L1,L2).
linear_simple([X|L1],L4     ) :- linear_simple(X,L2),
                                 linear_simple(L1,L3),
                                 append(L2,L3,L4).
```

Your new task is to write a version of linearisation, "linear_acc(+L,?Res)" that does not need the expensive call to "append/3". It should use a ternary helper predicate "linear_acc(+L,+Acc,?Res)" whose additional parameter Acc acts as an accumulator of the already linear part of the list. New elements should always be inserted in front of the accumulated intermediate result (not appended at the end).

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

Task 2. *Understanding term-based predicates* (6 Points)

Imagine you are trying to understand a program written by a colleague. Among others, you find the following uncommented predicate definition:

```
p([],[]).

p([A],[A]).

p([A|B],[A,C|D]):- p(B,[C|D]), A < C.

p([A|B],[C|D])  :- p(B,[C|E]), not(A<C), p([A|E],D).
```

Your primary task is to find out what the predicate does.

a) (2 points) Describe how each clause and each sub goal contributes to its functionality.
b) (1 point) Improve the readability of the predicate by proper renaming of the predicate and its variables.
c) (2 points) Write a comprehensive documentation that describes its general intention, the invocation modes in which it can be used and the differences between the modes (modes that behave the same way should be described together).
d) (1 point) Exemplify each mode by a sample query and its result. This is useful for documentation and as a basis for automated testing.

Tip for c and d): When describing the predicate, consider any "type information" that you can deduce from its arguments. That includes not only the modes but also any constraints (implicit assumptions in the code) about the values passed in each argument.

Task 3. *Accumulators: Performance evaluation* (2 Points)

Compare the runtime of the simple and accumulator-based version from Task 1 by running the following test code 10 times and averaging the results for each version:

```
?- make_list(150,5,L),
   time( linear_acc (L,_) ),
   time( linear_simple(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).
```