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

Advanced Logic Programming
Summer Semester 2012
Dr. Günter Kniesel

Assignment 6

Due: Sunday, 10.06.2012, 23:59:59 via SVN

Task 1. *Output and backtracking-driven iteration* (2 Points)

a) Assume there is a predicate p/2. How would you print out all results for the call p(X,Y)? Write a query that does this.

```
?- p(X,Y), ...  .  % ← fill in the ...
```

Task 2. *Shifting lists* (3 Points)

In task 5.3 you developed a first version of a linearised list that mostly looked like the following "linear_simple/2":

```
linear_simple([],[]) .
linear_simple([X|L1],[X|L2] ) :- keep_unchanged(X),
                                 linear_simple(L1,L2).
linear_simple([X|L],L2     ) :-  X == [],
                                 linear_simple(L,L2).
linear_simple([X|L1],L4     ) :- \+ keep_unchanged(X),
                                 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 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).
```

Task 4. *Cuts* (2 Points)

a) Consider the following program

```
p(1).
p(2):-!.
p(3).
```

Write all answers to the following queries:

```
1  ?- p(X).
2  ?- p(X), p(Y).
3  ?- p(X), !, p(Y).
```

b) Explain for each case why the program gives that answer.

Tip: As a preparation for the oral exam at the end of the course do it with *pen and paper only*, not by running the program! Or even better: Try to explain it to a colleague. For oral exams, practicing talking is the best preparation! Run it and use the debugger only if you cannot do it with pen and paper!!!

Task 5. *Declarative and Procedural Meaning and Cuts* (3 Points)

Give the declarative and procedural meaning for the predicates p, q, r. Describe the cause for differences and similarities. Recall that the declarative meaning of cut is simply “true” (it always succeeds).

```
p:- a, b.
p:- c.

q:- a, !, b.
q:- c.

r:- c.
r:- a, !, b.
```

Task 6. *Sorting and Cuts* (6 Points)

Where are good places to use cuts to optimise the following sorting programs? What would be changed by the cuts?

Which version (a, b, c and each either with or without cuts) would you recommend to use? Explain the reason behind your recommendation. Implement your preferred version and give it a comprehensive description.

```
a) sort([],[]).
   sort([X],[X]).
   sort([H|T1],[H,S|T2]):- sort(T1,[S|T2]), H < S.
   sort([H|T1],[S|T3])   :- sort(T1,[S|T2]), not(H<S), sort([H|T2],T3).
```

b)
```
sort1([X|Xs], Ys) :- sort1(Xs, Zs), insert(X, Zs, Ys).
sort1([],[]).

insert(X, [], [X]).
insert(X, [Y|Ys], [Y|Zs]  ):- X > Y, insert(X, Ys, Zs).
insert(X, [Y|Ys], [X,Y|Ys]):- X=<Y.
```

c)
```
sort2([X|Xs], Ys):-
    partition(Xs, X, Littles, Bigs),
    sort2(Littles, Ls),
    sort2(Bigs, Bs),
    append(Ls, [X|Bs], Ys).
sort2([],[]).

partition([X|Xs], Y, [X|Ls], Bs):- X=<Y, partition(Xs, Y, Ls, Bs).
partition([X|Xs], Y, Ls, [X|Bs]):-  X>Y, partition(Xs, Y, Ls, Bs).
partition([], Y, [], []).
```