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

Advanced Logic Programming
Summer Semester 2012
Dr. Günter Kniesel

Assignment 5

Due: Sunday, 27.05.2011, 23:59:59 via SVN

Please note that the lecture and the exercise groups on Wednesday, May 23 will not take place because of the 'dies academicus'.

The week after the dies academicus is entirely free of courses (Whitsun break), so the results of this assignment may be submitted one week later than usual, on May 27.

Task 1. *Double negation* (2 Points)

Given the following program discuss and argue whether r/1 and s/1 are equivalent or not. Provide a simple definition of p/2 on which you can demonstrate your arguments.

Tip: Consider the possible invocation modes of r/1 and s/1.

```
r(X) :- p(a,X).

s(X) :- not(not(p(a,X)).
```

r/1 and s/1 are **not** equivalent. r/1 gives all possible unifications for a free variable X. On the other hand, s/1 only checks if the predicate not(not(p(a, X))) is true. The not/1 predicate doesn't give a unification if the variable is free. not/1 does not unify!

p(a,a) is a simple definition of p/2 that proves the argumentation:

```
?- r(X).
X = a.
?- s(X).
true.
```

Task 2. *Classification and Negation* (3 Points)

Assume we have a database of results of tennis games played by members of a club. The results are represented as facts for the predicate beat/2, meaning that the player mentioned in the first argument has beaten the player in the second argument:

```
beat( tom, jim ).     % tom has beaten jim
beat( ann, tom ).     % ann has beaten tom
beat( pat, jim ).     % pat has beaten jim
```

Your task is to define a predicate "category(Player,Category)" that classifies the players into three categories:

1) ***winner***: A player who won all his or her games.
2) ***fighter***: A player who won some games and lost some.
3) ***loser***: A player who lost all his or her games.

For instance, "?- category(tom, fighter)." should succeed.

Sample solution:

```
/**
* beat(?Winner, ?Looser)
*
* Succeeds iff Arg1 won a tennis match over Arg2.
*/
beat( tom, jim ).
beat( ann, tom ).
beat( pat, jim ).

/**
* category(?Player,?Category)
*
* Succeeds iff Arg1 is a player of the Category represented by Arg2
* as
* winner: Every player who won all his or her games.
* fighter: Every player who won some games and lost some.
* loser: Every player who lost all his or her games.
*/
category(Player, winner):-
      beat(Player,_),
      not(beat(_,Player)).

category(Player, fighter):-
      beat(Player,_),
      beat(_,Player).

category(Player, loser):-
      beat(_,Player),
      not(beat(Player,_)).
```

Task 3. *Grouping consecutive list elements* (3 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.
*    (list,list) (+,?)
*/
group([],[]).
group([X|Xs],[Z|Zs]) :- transfer(X,Xs,Ys,Z), group(Ys,Zs).

/**
* transfer(X,Xs,Ys,Z) Ys is the list that remains from the list Xs
*    when all leading copies of X are removed and transfered to Z
*/
transfer(X,[],[],[X]).
transfer(X,[Y|Ys],[Y|Ys],[X]) :- X \= Y.
transfer(X,[X|Xs],Ys,[X|Zs]) :- transfer(X,Xs,Ys,Zs).
```

Task 4. *Grouping consecutive list elements* (1 Points)

Modify your predicate from Task 3 so that it does not put an element into a sublist if there is no consequtive 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 3 and 4) 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.
*    (list,list) (+,?)
*/
group([],[]).
group([X|Xs],[RZ|Zs]) :- transfer(X,Xs,Ys,Z),
                         stripbrackets(Z, RZ),
                         group(Ys,Zs).
```

```
/**
* stripbrackets(X,Y) :- succeeds when X is a list of single element Y
*    or when X is Y.
*    (X,Y) (+,?)
*/
stripbrackets([X], X) :- !.
stripbrackets(X, X).

/**
* transfer(X,Xs,Ys,Z) Ys is the list that remains from the list Xs
*    when all leading copies of X are removed and transfered to Z
*/
transfer(X,[],[],[X]).
transfer(X,[Y|Ys],[Y|Ys],[X]) :- X \= Y.
transfer(X,[X|Xs],Ys,[X|Zs]) :- transfer(X,Xs,Ys,Zs).
```

Task 5. *Understanding term-based predicates* (6 Points)

Try to understand 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 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). Make the description as concise as possible by using (if appropriate) the ? mode as a generalisation of + and –.
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.

Sample solution:

The predicate is a sorting algorithm. It is a version of bubble sort.

The first and second clauses are the exit conditions. If the first argument is an empty list or a list with only one element it is already sorted.

The third clause describes the case that the first element of the list is the overall smallest element and as such already sorted. To accomplish this we start with a recursive sorting of the tail. After that we compare the first (smallest) element of the result with the original head element. If the head is even smaller than the smallest element of the result we get an overall sorted list if we put it before everything of the result list.

The fourth clause describes the case that the first element of the list is bigger than something in the tail. Again we start with sorting the tail and comparing its smallest element with the original head element. But now the head is bigger so it has to be recursively sorted somewhere behind the first element of the result of the first sorting. We get the overall sorted list if put the smallest element of the first sorting (which is the overall smallest element) before the result of the second sorting.

a)
```
bsort([],[]).
bsort([SingleElement],[SingleElement]).
bsort([SmallestElem|Tail],[SmallestElem,SecSmallestElem|SortedTail]):-
    bsort(Tail,[SecSmallestElem | SortedTail]),
    SmallestElem < SecSmallestElem.
bsort([AnElem|Tail],[SmallestElem|SortedTail]) :-
    bsort(Tail,[ SmallestElem |RemainingTail]),
    not(AnElem <SmallestElem),
    bsort([AnElem | RemainingTail], SortedTail).
```

b)
```
/**
* bsort(+UnsortedList, ?SortedList)
*
* succeeds if Arg1 is a list and Arg2 contains the same
* elements as Arg1 in ascending order. It is a kind of
* bubble sort.
*/
```

c)
```
5 ?- bsort([3,2,1],[1,2,3]).  % (+,+) reverse ordered
list
true .

6?- bsort([3,1,2],[1,2,3]).   % (+,+) unordered list
true .

7 ?- bsort([1,2,3],[1,2,3]).  % (+,+) sorted list
true .

8 ?- bsort([1,2,3],X).  % (+,-) sorted list
X = [1, 2, 3] ;
false.

9 ?- bsort([2,1,3],X).  % (+,-) unordered list
X = [1, 2, 3] ;
false.

10 ?- bsort([3,2,1],X).  % (+,-) reverse ordered list
X = [1, 2, 3] ;
false.

11 ?- bsort(X,Y).  % (-,-)
X = [],
Y = [] ;
```

```
X = [_G376],
Y = [_G376] ;
ERROR: [Thread pdt_console_client_0@localhost] </2:
Arguments are not sufficiently instantiated

14 ?- bsort(X,[1,2,3]).   % (-,+)
X = [1, 2, 3] ;
ERROR: [Thread pdt_console_client_0@localhost] </2:
Arguments are not sufficiently instantiated
```