Assignment 3

Due: Friday, 26.05.2017, 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 results into the folder "assignment03/" of the git repository of your group.

For each task, submit your implemented predicate as a file named "taskN.pl". At the bottom of the file add a comment containing console output that shows all results of a successful test run of your predicate.

Task 1. *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 2. *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))).
```

Sample solution:

Logically the predicates are equivalent but operationally they are **not** equivalent:

- If r/1 is called with a free variable as argument it returns a binding for the variable when it succeeds.
- If s/1 is called with a free variable as argument it does not bind the variable since not/1 succeeds only if its argument goal fails!

"p(a,a)." is a simple definition of p/2 that illustrates the principle:

```
?- r(X).
X = a.
?- s(X).
true.   % ← No binding for X here!
```

Task 3. *Shifting lists* (2 Points)

Implement a predicate "shift(List1, List2)" so that List2 is List1 'shifted rotationally' by one element to the left, which means that the element that is shifted out on the left-hand-side comes in again on the right-hand-side. For instance,

```
?- L1 = [1,2,3,4,5], shift(L1, L2), shift(L2, L3).
```

should produce

```
L1 = [1,2,3,4,5]
L2 = [2,3,4,5,1]
L3 = [3,4,5,1,2]
```

Sample solution:

```
%%shift(?List1, ?List2)   // <- 1 point for %%, signature, and modes
%
% Succeeds iff Arg2 is Arg1 'shifted rotationally' by one element
% to the left            // <- 0,5 points for relational comment
%
shift([],[]).                          // 1 point
shift([A|T],Result):-                  // 1 point
      append(T,[A],Result).            // 1 point
```

Task 4. *Mapping list elements* (4 Points)

Implement a predicate "translate(DigitList, WordList)" that succeeds if DigitList is a list of digits and each element in WordList is the English word for the element at the same position of DigitList. For instance,

```
?- translate( [1,2,3], L2).
```

should produce

```
L2 = [one, two, three]
```

and

```
?- translate( L1, [one, two, three]).
```

should produce

```
L1 = [1,2,3]
```

Tip: Start by considering how to represent the mapping of digits to words and vice versa. To check whether a term is a digit use the following helper predicate, which uses built-in predicates that we haven't discussed yet – see the SWI-Prolog online documentation:

```
digit(X) :- number(X), X>=0, X<10.
```

Sample solution:

```
%% translate(?DigitList, ?WordList) // <- 1 point (%%, signature, modes)
%
% Succeeds iff DigitList is a list of digits and each element in
% WordList is the English word for the digit at the same position
% in DigitList.                    // <- 0,5 points for relational comment
% Note that if one would allow DigitList to contain free variables,
% the new variables should appear at the same position of the
% WordList (Think: Why not the same variables?)

translate([],[]).                                    // 1 point
translate([Digit|Digits], [Name|Names]):-            // 1 point
     digit_name(Digit,Name),                         // 1 point
     translate(Digits,Names).                        // 1 point

%% digit_name(?Digit,?Name) // <- 1,5 point for full comment, like above
%
% Succeeds if Arg2 is the textual name of the digit (0..9) from Arg1.

digit_name(0, zero).      // 2 points for all 9 correct clauses
digit_name(1, one).
digit_name(2, two).
digit_name(3, three).
digit_name(4, four).
digit_name(5, five).
digit_name(6, six).
digit_name(7, seven).
digit_name(8, eight).
digit_name(9, nine).
```

Task 5. *Linear list* (4 Points)

Implement a predicate "linear(+NestedList, ?LinearList)" that succeeds whenever NestedList is a list whose elements may be arbitrarily deeply nested lists and LinearList contains all elements of NestedList in the same order but without any nesting. For instance,

```
?-linear ( [1,[2,3],[a,[b],c]], [1,2,3,a,b,c]).
```

should succeed but

```
?-linear ( [1,[2,3],[a,[b],c]], [1,2,3,a,c,b]).
```

should fail (because the order of c and b in argument 1 and argument 2 differs).

Tip: You may use the predefined predicate `append(A,B,AB)` to implement your version of linear/2. Except for that, your solution must be self-contained.

Sample solution:

```
%%linear(+NestedList, ?LinearList)
%
% Succeeds if is Arg1 is a list (whose elements may themselves
% be arbitrarily deeply nested lists) and Arg2 contains all elements
% from Arg1 in the same order but without any nesting.
% ← for full comment, like above there are  // 1,5 points

linear([],[]).                                  // 1 point
linear([H|T], [H|TFlatt]):-                     // 1 point
    not(is_list(H)),                            // 1 point
    linear(T,TFlatt).                           // 1 point
linear([H|T], Flatt):-                          // 1 point
    linear(H, Hflatt),                          // 1 point
    linear(T, Tflatt),                          // 1 point
    append(Hflatt, Tflatt, Flatt).              // 1 point

is_list([]).                                    // 1 point
is_list([_|T]) :- is_list(T).                   // 1 point
```

Note: Instead of `not(is_list(H))` you could have used the built-in predicate `atomic(H)`. However, the task was to implement yourself everything except append/2, not to use built-in predicates.