Assignment1 – Prolog

Problem 1.1 (Basic *Prolog* Functions)

Implement the *functions* listed below in *Prolog*. Note that many of them are built-in, but we ask you create your own *predicate*.

1. a *predicate reversing* a *list*
   Test case:
   ```
   ?- myReverse([1,2,3,4,2,5],R).
   R = [5, 2, 4, 3, 2, 1].
   ```

2. a *predicate* removing multiple occurrences of *elements* in a *list*
   Test case:
   ```
   ?- removeDuplicates([1,1,1,1,2,2,3,4,1,2,7],A).
   A = [1, 2, 3, 4, 7].
   ```

   Hint: You may want to *implement* a helper *predicate* `delete(X,LS,RS)`, that removes all instances of `X` in `LS` and *returns* the result in `RS`.

   ---

3. a *predicate* for zipping two *lists*
   `zip` takes two *lists* and outputs a *list* of *pairs* (*represented* as 2-*element lists*) of *elements* at the same index in the two *lists*. If the *lists* do not have the same length, the zipped *list* contains only as many *pairs* as the shorter *list*.

   Create a *Prolog predicate* with 3 *arguments*: the first two are the two *lists* to zip and the third one the result. For instance:

   ```
   ?- zip([1,2,3],[4,5,6],L).
   L = [[1, 4], [2, 5], [3, 6]].
   ```

   ```
   ?- zip([1,2],[3,4,5],L).
   L = [[1, 3], [2, 4]].
   ```

4. a *predicate* for *computing permutations* of a *list*
   Try it out on paper first and understand why this is difficult.

   Test case:
   ```
   ?- myPermutations([1,2,3],P).
   P = [1, 2, 3] ;
   P = [2, 1, 3] ;
   P = [2, 3, 1] ;
   P = [1, 3, 2] ;
   P = [3, 1, 2] ;
   P = [3, 2, 1].
   ```

Note that there are two ways for specifying such a *function*:

(a) *return* a *list* of all *permutations*

(b) *return* a single *permutation* each time such that *Prolog* finds them one by one.

Here we are using the second way, i.e., `myPermutations(L,P)` must in particular be *true* if `P` is some *permutation* of `L`.

Hint: One possible solution is to start with a helper *predicate* `takeout(X,L,M)` that is true *iff* `M` is the result of removing the first occurrence of `X` from `L`. Or equivalently: `M` arises by adding `X` somewhere in `L`. How does this allow you to define the notion of *permutation recursively*?

Solution:

1. the *reverse predicate*

```
% myReverseAcc uses an additional argument (the second one) as an accumulator
% in which the result is built.
% Its invariant is that myReverserAcc(X,Y,Z) iff reverse(X);Y = Z.
% When the first argument is empty, we return the accumulated result.
myReverserAcc([],X,X).
% When the first argument is non-empty, we take its first element and
% prepend it to the accumulator.
myReverserAcc([X|Y],Z,W) :- myReverserAcc(Y,[X|Z],W).
% To compute the reversal, we initialize the accumulator with the empty list.
myReverse(A,R) :- myReverserAcc(A,[],R).
```

2. the remove duplicates *predicate*

```
delete(_,[],[]).
delete(X,[X|T],R) :- delete(X,T,R).
delete(X,[H|T],[H|R]) :- not(X=H), delete(X,T,R).
removeDuplicates([],[]).
removeDuplicates([H|T],[H|R]) :- delete(H,T,S), removeDuplicates(S,R).
```

3. the zip *predicate*

```
zip(L,[],[]).
zip([],L,[]).
zip([H1|T1],[H2|T2],[[H1,H2]|T]) :- zip(T1,T2,T).
```

4. the permute *predicate*

```
takeout(X,[X|T],T).
takeout(X,[H|T1],[H|T2]) :- not(X=H), takeout(X,T1,T2).

% There is exactly one permutation of the empty list.
myPermutations([],[]).
% To find a permutation P of a longer list [H|T], we permute T into Q
% and insert H somewhere into Q.
myPermutations([H|T],P) :- myPermutations(T,Q), takeout(H,P,Q).
```

Note that we defined `takeout` in such a way that the second *argument* is input and the third one output. But *Prolog* does not distinguish input and output: when we use it later, we use the third *argument* as input and the second one as output.

Problem 1.2

1. *Program* a *Prolog predicate* uadd for *addition* and umult for *multiplication* in *unary representation*.

Hint: The *number* 3 in *unary representation* is the *Prolog term* `s(s(s(o)))`, i.e. application of the arbitrary *function* `s` to an arbitrary *argument* `o` *iterated* three times.

Hint: Note that *Prolog* does not allow you to *program* (*binary*) *functions*, so you must come up with a three-place *predicate*. You should use `add(X,Y,Z)` to *mean* $X+Y=Z$ and *program* the *recursive equations* $X+0=X$ (*base case*) and $X+s(Y)=s(X+Y)$.

Solution:

```
uadd(X,o,X).
uadd(X,s(Y),s(Z)) :- uadd(X,Y,Z).

umult(_,o,o).
umult(X,s(Y),Z) :- umult(X,Y,W), uadd(X,W,Z).
```

---

2. Write a *Prolog predicate* ufib that *computes* the n^{th} *Fibonacci Number* (0, 1, 1, 2, 3, 5, 8, 13,… *add* the last two to get the next), using the *addition predicate* above.

Solution:

```
ufib(o,o).
ufib(s(o),s(o)).
ufib(s(s(X)),Y):-ufib(s(X),Z),ufib(X,W),uadd(Z,W,Y).
```

If you have mastered *addition* and *multiplication*, feel free to try your hands on *exponentiation* as well.

Problem 1.3 (Binary Tree)

A *binary tree* of (in this case) *natural numbers* is *inductively* defined as either

- an *expression* of the form `tree(n,t1,t2)` where `n` is a *natural number* (the *label* of the *node*) and `t1` and `t2` are themselves *binary trees* (the *children* of that *node*)
- or `nil` for the *empty tree*. (Normally a *tree* cannot be *empty*, but it is more convenient here to allow an *empty tree* as well.)

In particular, the *nodes* of the form `tree(n,nil,nil)` are the *leaf nodes* of the *tree*, the others are the inner *nodes*.

An example *tree* in *Prolog* would be:

```
tree(1,tree(2,nil,nil),tree(2,nil,nil))
```

1. Write a *Prolog predicate* `construct` that constructs a *binary tree* out of a *list* of (distinct) *numbers* such that for every *subtree* `tree(n,t1,t2)` all *values* in `t1` are smaller than `n` and all *values* in `t2` are larger than `n`.

   Note that there are usually multiple such *trees* for every *list*. One example is:

   ```
   ?- construct([3,2,4,1,5],T).
   T = tree(3, tree(2, tree(1, nil, nil), nil),
       tree(4, nil, tree(5, nil, nil))).
   ```

2. Write *Prolog predicates* `count_nodes` and `count_leaves` that take a *binary tree* and *return* the *number* of *nodes* and *leaves*, respectively.

3. A *binary tree* is symmetric if it is its own mirror image, i.e., all *nodes* have left and right *child* switched. Write a *Prolog predicate* `symmetric` that checks whether a *binary tree* is symmetric.

Solution:

```
% add(X,S,T) inserts a node with label X into tree S yielding tree T
% Inserting into the empty tree yields a tree with a single node.
add(X,nil,tree(X,nil,nil)).
% To insert an element smaller than the root, insert on the left.
add(X,tree(Root,L,R),tree(Root,L1,R)) :- X @< Root, add(X,L,L1).
% To insert an element bigger than the root, insert on the right.
add(X,tree(Root,L,R),tree(Root,L,R1)) :- X @> Root, add(X,R,R1).

% To construct a binary tree T, from a list L, we insert all elements in order.
% We use an accumulator that we initialize with the empty tree.
construct(L,T) :- constructAcc(L,T,nil).
% At the end of the list, we return the accumulator.
constructAcc([],T,T).
% For each element of the list, we add it to the accumulator A (obtaining A1) and recurse.
constructAcc([N|Ns],T,A) :- add(N,A,A1), constructAcc(Ns,T,A1).

% The empty tree has no nodes.
count_nodes(nil,0).
% An inner node has one more node than its child trees together.
count_nodes(tree(_,L,R),N) :- count_nodes(L,NL), count_nodes(R,NR), N is NL+NR+1.
% Note that we do not need an additional case for leaf nodes here.

% The empty tree has no leaves.
count_leaves(nil,0).
% A leaf node has 1 leaf (itself).
count_leaves(tree(_,nil,nil),1).
% An inner node has as many leaves as its child trees together.
count_leaves(tree(_,L,R),N) :- count_leaves(L,NL), count_leaves(R,NR), N is NL+NR.

% The empty tree is symmetric.
symmetric(nil).
% Any other tree is symmetric if its two child trees are mirror images of each other.
symmetric(tree(_,L,R)) :- mirror(L,R).

% The empty tree is its own mirror image.
mirror(nil,nil).
% Otherwise, the mirror image arises by mirroring and swapping the child trees.
mirror(tree(X,L1,R1),tree(X,L2,R2)) :- mirror(L1,R2), mirror(R1,L2).

%A few tests
test1(X):- construct([5,2,4,1,3],Y), count_leaves(Y,X).
% X=2
test2(X):- construct([6,10,5,2,9,4,8,1,3,7],Y), count_leaves(Y,X).
% X=3
symmetric(tree(1,tree(2,nil,nil),tree(2,nil,nil))).
% true.
symmetric(tree(1,tree(3,nil,nil),tree(2,nil,nil))).
% false.
```