Functional and Logic Programming Fall 2017

S09 : Prolog (Introduction)

Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

Exercise 1

insert

```
% insert(+X,?L1,?L2), succeed if L1 and L2 are sorted and if L2 is L1 with X inserted
insert(X,[],[X]).
insert(X,[E1|L1],[X,E1|L1]) :- E1 > X.
insert(X,[E1|L1],[E1|L2]) :- E1 =< X, insert(X,L1,L2).
```

mergeLists

```
% mergeList(+Xs:list[int],+Ys:list[int],?L3:list), succeed if L3 is sorted and the merge of L1 and L2
mergeList([],[],[]).
mergeList(X,[],X).
mergeList([],Y,Y).
mergeList([X|Xs],[Y|Ys],[X|XYs]) :-
    X < Y,
    mergeList(Xs,[Y|Ys],XYs).
mergeList([X|Xs],[Y|Ys],[Y|XYs]) :-
    mergeList([X|Xs],Ys,XYs).
```

reverseRec

```
% reverseRec(+L1:list,?L2), succeed if L1 is the reverse of L2 and if each nested list of L1 are also
↪  reversed
reverseRec(L1,L2) :- reverseRec(L1,L2,[]).

% recursion with accumulator is simplier for reversing a list
reverseRec([],Acc,Acc).
reverseRec([X|Xs],Ys,Acc) :-
    is_list(X),
    reverseRec(X,Xr),
    reverseRec(Xs,Ys,[Xr|Acc]).
reverseRec([X|Xs],Ys,Acc) :-
    atomic(X),
    reverseRec(Xs,Ys,[X|Acc]).
```

myMin

```
% myMin(+L:list,?X), succeed if X is the smallest element of L
myMin([E|L],X) :- myMin(L,E,X).

% recursion with accumulator used just to pass back the minimal value
myMin([],Acc,Acc).
myMin([X|Xs],E,Acc) :-
    X < E,
    myMin(Xs,X,Acc).
myMin([_|Xs],E,Acc) :- myMin(Xs,E,Acc).
```

Exercise 2

createLA

```
% createLA(?Keys,?Values,?ListAssoc), succeed if ListAssoc is the zip of Keys and Values
createLA([],[],[]).
createLA([],_Values,[]).
createLA(_Keys,[],[]).
createLA([K|Keys],[V|Values],[(K,V)|L]) :- createLA(Keys,Values,L).
```

valuesLA

```
% valuesLA(+LA:list[(k,v)],?Key,?Value), succeed if (Key,Value) is in LA
valuesLA([(Key,Value)|_],Key,Value).
valuesLA([(_,_)|LA],Key,Value) :- valuesLA(LA,Key,Value).
```

Exercise 3

sum_list

```
% sum_list(+L:list[number],?S), succeed if S is the sum of all element is L
sum_list([],0).
sum_list(L,S) :- sum_list(L,S,0).

% using accumulator, sum_list/2 is just the initiation predicat
sum_list([],Acc,Acc).
sum_list([X|Xs],S,Acc) :-
    NewAcc is Acc + X,
    sum_list(Xs,S,NewAcc).
```

reverseRec

```
% reverseRec(+L1:list,?L2), succeed if L1 is the reverse of L2 and if each nested list of L1 are also
↪  reversed
reverseRec(L1,L2) :- reverseRec(L1,L2,[]).

% using accumulator, reverseRec/2 is just the initiation predicat
reverseRec([],Acc,Acc).
reverseRec([X|Xs],Ys,Acc) :-
    is_list(X),
    reverseRec(X,Xr),
    reverseRec(Xs,Ys,[Xr|Acc]).
reverseRec([X|Xs],Ys,Acc) :-
    atomic(X), % for assertion, not necessary
    reverseRec(Xs,Ys,[X|Acc]).
```

myMin

```prolog
% myMin(+L:list,?X), succeed if X is the smallest element of L
myMin([E|L],X) :- myMin(L,E,X).

% recursion with accumulator used just to pass back the minimal value
% myMin/2 is the initiation predicate
myMin([],Acc,Acc).
myMin([X|Xs],E,Acc) :-
    X < E,
    myMin(Xs,X,Acc).
myMin([_|Xs],E,Acc) :- myMin(Xs,E,Acc).
```

fibonacci

```prolog
% succeed if F is the Nth fibonacci number
fibonacci(N,F) :- fibonacci(N,F,0,1).

% implementation of fibonacci/2, which is just the initiation predicat
fibonacci(0,Acc1,Acc1,_).
fibonacci(N,F,Acc1,Acc2) :-
    NextAcc is Acc1 + Acc2,
    N1 is N - 1,
    fibonacci(N1,F,Acc2,NextAcc).
```

Exercise 4

```prolog
% quicksort(+L1:list,?L2),succeed if L2 is L1 sorted
quicksort([],[]).
quicksort([X],[X]).
quicksort([Pivot|Xs],Ys) :-
    split(Pivot,Xs,Xs1,Xs2),
    quicksort(Xs1,Ys1),
    quicksort(Xs2,Ys2),
    mergeList(Ys1,[Pivot|Ys2],Ys).

% split(P,Xs,L1,L2), succeed if L1 and L2 and P are Xs (from a set point of view)
% --> L1 U L2 U {P} <-> {x | x <- Xs}
split(_,[],[],[]).
split(P,[X|Xs],[X|Smaller],GreaterOrEquals) :-
    X < P,
    split(P,Xs,Smaller,GreaterOrEquals).
split(P,[X|Xs],Smaller,[X|GreaterOrEquals]) :-
    X >= P,
    split(P,Xs,Smaller,GreaterOrEquals).

% mergeList(+Xs:list[int],+Ys:list[int],?L3:list), succeed if L3 is sorted and the merge of L1 and L2
mergeList([],[],[]).
mergeList(X,[],X).
mergeList([],Y,Y).
mergeList([X|Xs],[Y|Ys],[X|XYs]) :-
    X < Y,
    mergeList(Xs,[Y|Ys],XYs).
mergeList([X|Xs],[Y|Ys],[Y|XYs]) :-
    mergeList([X|Xs],Ys,XYs).
```