SEMINAR 7 – Recap problems

"Right now I'm having amnesia and déjà vu at the same time. I think I've forgotten this before." – Steven Wright

A. PROLOG

1. The code below is the solution of a student for the problem of removing from a list all the elements that occur multiple times. For example for the list [1,2,3,2,1] the solution is [3].

This solution is made of 3 predicates:
- Predicate *exists*, which checks if an element occurs in a list or not.
- Predicate *removeElem* which removes all occurrences of an element from a list
- Predicate *solution* which removes the elements which occur multiple times.

```
% exists (L:list, E: element)
% flow model: (i,i), (i,o)
% L – the list that we check
% E – the element that we search for
exists([H|T],E) :- H=E.
exists([H|T],E) :- exists(T,E).

% removeElem (L: list, E: element, R: list)
%flow model: (i,i,o), (i,o,i)
% L – the list we remove from
% E – the element to be removed
% R – the resulting list
removeElem([],X,[]).
removeElem([H|T],X,R) :- H=X,removeElem(T,X,R).
removeElem([H|T],X,[H|R]) :- removeElem(T,X,R).

%solution (L: list, R: list)
%flow model (i,o)
% L – the list to be transformed
% R – the resulting list
solution([],[]).
solution([H|T],S) :- exists(T,H),
                     removeElem(T,H,R),
                     !,
                     solution(R,S).
solution([H|T],[H|S]) :-  not(exists(T,H)),
                    solution(T,S).
```

- Is the above code correct?
  - If we look carefully, we can see that we have a problem in the predicate *removeElem*. It is a non-deterministic predicate (it has several solutions), though we want a deterministic predicate. For example, for the call *removeElem([1,2,4,2,1,6], 2, X)* the results will be:

    - [1,4,1,6]
    - [1,4,2,1,6]
    - [1,2,4,1,6]
    - [1,2,4,2,1,6]
  - So, *removeElem* is incorrect because it gives us several solutions (and most of them are incorrect). Is then the whole solution incorrect? Interestingly, if we run predicate *solution*, we could see that it works perfectly. It returns one single solution and that is the correct one. Why?

2. At the lectures you have discussed the code to generate all the combinations of N elements from a list. The code looked like this:

```
%comb(L: list, N: integer, R:list)
%flow model (i,i,o)
%L – the initial list
%N – the number of element for the combinations
%R – the resulting list
comb([E|_], 1, [E]).
comb([_|T], N, R):-
      comb(T, N, R).
comb([H|T],N, [H|R]):-
      N > 1,
      N1 is N - 1,
      comb(T, N1, R).
```

- Assume that we do not want all the possible combinations, we want only those where the elements are in increasing order. Instead of writing another predicate to check if a list represents an increasing sequence, we want to change the current implementation of *comb*, to generate only the correct solutions.
- If we want combinations with increasing element we should change the clause where an element is added into the solution. So we rewrite the third clause in the following way:

```
comb([H|T], N, [H,H1|R]):-
      H < H1,
      N > 1,
      N1 is N - 1,
      comb(T, N1, [H1|R]).
```

- Is this version correct?
- We will get an error at the H < H1 part, because variable H1 is unbound. The third parameter is of type output, it has no value the moment of the call and it will get a value inside the clause (actually from the recursive call). So only after the recursive call can we compare H and H1 when we know for sure that both of them have value.

```
comb([H|T],N, [H,H1|R]):-
      N > 1,
      N1 is N - 1,
      comb(T, N1, [H1|R]),
H < H1.
```

3. **Write a prolog program to generate all subsets of a set, with the property that the difference between any 2 consecutive elements is a multiple of 3. Subsets must have at least 2 elements.**

allsol([3,6,12, 4, 5, 10, 13], R). =>
R = [[3, 6, 12], [3, 6], [3, 12], [6, 12], [4, 10, 13], [4, 10], [4, 13], [10, 13]]

```
%subs(L-lis, R-list) (i,o)
subs([],[]).
subs([H|T], [H|R]):-
    subs(T,R).
subs([_|T], R):-
    subs(T,R).

prop([_]):-!.
prop([H1,H2|T]):-
    D is abs(H1-H2),
    D mod 3 =:= 0,
    prop([H2|T]).

onesol(L, SS):-
    subs(L, SS),
    SS=[_,_|_],
    prop(SS).

allsol(L,R):-
    findall(X, onesol(L,X), R).
```

4. **Given a heterogeneous list composed of numbers and sublists of numbers, delete from every sublist the palindrome numbers. Use collector variable for inverting a number.**

```
%inv_numar(Nr: int, Ncv: int, Rez: int)
%Nr – number to reverse
%Ncv – collector var in which we compute the parial result of the
reverse number
%Rez – the result
%model de flux (i, i, o), (i, i, i)
inv_numar(0, Ncv, Ncv).
inv_numar(Nr, Ncv, Rez):-
Nr > 0,
Cifra is Nr mod 10,
NcvNew is Ncv * 10 + Cifra,
NrNew is Nr div 10,
inv_numar(NrNew, NcvNew, Rez).


%delSub(L:list, R: list)
```

```
%L - lista liniara din care eliminam numerele care sunt palindrom
%LR - lista rezultat
%model de flux (i, o), (i, i)
delSub([], []).
delSub([H|T], [H|TR]):-
            inv_numar(H, 0, HI),
            H \= HI,
            delSub(T, TR).
delSub([H|T], TR):-
            inv_numar(H, 0, HI),
            H = HI,
            delSub(T, TR).



%mainHeter(L: lista, LR: list)
%L - lista eterogena initiala
%LR - lista rezultat
%model de flux (i, o), (i, i)
mainHeter([], []).
mainHeter([H|T], [H|LR]):-
            number(H),
            mainHeter(T, LR).
mainHeter([H|T], [H1|LR]):-
            is_list(H),
            delSub(H, H1),
            mainHeter(T, LR).
```

B. LISP

1. What is the result of the execution of the following instructions?

a. (setq car ‘cdr)
(car ‘(1 2 3 4)) => 1
(eval car ‘(1 2 3 4 5)) => ERROR Too many arguments
(eval (cons car ‘(1 2 3 4)) => ERROR Too many arguments given to CDR
(eval (list car ‘(1 2 3 4)) => ERROR 1 is not a function name
(eval (list car ‘ ’(1 2 3 4)) => (2 3 4)
(eval car) => Error Variable CDR has no value
(apply car ‘(1 2 3 4 5)) => Error, too many arguments given to CDR
(apply #’car ‘(1 2 3 4 5)) => Error, too many arguments given to CAR
(apply car ‘((1 2 3 4 5))) => (2 3 4 5)
(apply #’car ‘((1 2 3 4 5)) => 1
(funcall car ‘((1 2 3 4 5))) => NIL
(funcall #’car ‘((1 2 3 4 5))) => (1 2 3 4 5)
(funcall car ‘(1 2 3 4 5)) => (2 3 4 5)
(funcall #’car ‘(1 2 3 4 5)) => 1

b. (mapcar #'list '(1 2 3 4 5)) => ((1) (2) (3) (4) (5))
(mapcan #'list '(1 2 3 4 5)) => (1 2 3 4 5)
(maplist #'list '(1 2 3 4 5)) => (((1 2 3 4 5)) ((2 3 4 5)) ((3 4 5)) ((4 5)) ((5)))
(mapcon #'list '(1 2 3 4 5)) => ((1 2 3 4 5) (2 3 4 5) (3 4 5) (4 5) (5))
(apply #'append (mapcon #'list '(1 2 3 4 5))) => (1 2 3 4 5 2 3 4 5 3 4 5 4 5 5)

c. (mapcar #'max '(1 2 3 4 5)) => (1 2 3 4 5)
(eval (append '(+) (mapcar #'max '(1 2 3 4 5)))) => 15
(apply #'+ (mapcar #'max '(1 2 3 4 5))) => 15
(apply #'+ (mapcar #'length (maplist #'cdr '(1 2 3 4 5))) => 10

d. (mapcar #'(lambda (a b) (eval (list a b))) '(list max min evenp) '(1 2 3 4 5 6)) => ((1) 2 3 t)
(mapcar #'(lambda (a) (mapcar #'max a)) (maplist #'append '(1 2 3 4))) =>((1 2 3 4) (2 3 4) (3 4) (4))
(mapcar #'(lambda (a) (apply #'max a)) (maplist #'append '(1 2 3 4))) =>(4 4 4 4)

e. (setq x '(1 2 3 4 5))
(setq y '(6 7 8 9 10 11 12))
(mapcar #'(lambda(a b c d) (eval (funcall c d a b))) x y (mapcar #'(lambda (q) 'list) y) (mapcar #'(lambda (v) '+) x)) => (7 9 11 13 15)

2. Create a list with the positions of the maximum element in a non-linear list at any level.

```
(defun maxim (l)
  (cond
   ((null l) -10000)
   ((numberp (car l)) (max (car l) (maxim (cdr l))))
   ((atom (car l)) (maxim (cdr l)))
   )
)

(defun pozitii (l e p)
  (cond
   ((null l) nil)
   ((equal (car l) e) (cons p (pozitii (cdr l) e (+ 1 p))))
   (t (pozitii (cdr l) e (+ 1 p)))
   )
)


(DEFUN transform(l)
(COND
            ((null l) nil)
            ((or (atom (car l))  (numberp (car l))) (cons (car l)
(transform (cdr l))))
```

```
                    (T (APPEND (transform (car l)) (transform (cdr l))))
            )
            )
```

```
    (defun pozMain (l) (pozitii (transform l) (maxim (transform l)) 1))
```

3. Write a function that returns the depth of a tree represented as (node (subtree1) (subtree2) …) using MAP functions.

```
(defun depth (tree)
    (cond
        ((atom tree) 0)
        ( t (+ 1 (apply 'max (mapcar 'depth tree) )))
     )
)
```

4. With lambda. Write a function that remove all elements multiple of n.

```
(defun remn (l n)
  (cond
      ((and (numberp l) (= (mod l n) 0))  nil)
      ((atom l)       (list l))
      ( t  (list (mapcan #'(lambda (a) (remn a n)) l)))
   )
)

(defun main (l n)
  (car (remn l n))
)
```