Seminar 2- Lists in Prolog

- Write a predicate to remove from a list all the elements that appear only once. For example: [1,2,1,4,1,3,4] =>[1,1,4,1,4]

-go through the list. If head is in tail, keep the element + recursive call

- If head is not in the tail => recursive call

[1,2,1,4,1,3,4]

1 is in [2,1,4,1,3,4] => 1 + recursive call [2,1,4,1,3,4]

2 is not in [1,4,1,3,4] => recursive call [1,4,1,3,4]

1 is in [4,1,3,4] => 1 + recursive call [4,1,3,4]

4 is in [1,3,4] => 4 + recursive call [1,3,4]

1 is not in [3,4] => recursive call [3,4]

3 is not [4] => recursive call [4]

4 is not in [] => []

a. We should check in the initial list
b. We need to count how many times the element occurs

Count the occurrences of an element in a list

Count(x, l1,…,ln) = { 0, if n =0

1 + Count(x, l2,…,ln), if l1=x

Count(x, l2,…,ln), otherwise

}

%Count(X: elem, L: list, C: int)

%flow model (I, I, o) (I, I, I)

Count(_, [], 0):- !.

Count(X, [X|T], Cnt):- !,

Count(X, T, Oldcnt),

Cnt is Oldcnt + 1.

Count(X, [_|T], Cnt):-

Count(X, T, Cnt).

Count(1, [1,2,3,1,2,3], C).

C = 2

Count(1, [1,2,3,1,2,3], 2).

True

Count(1, [1,2,3,1,2,3], 4).

False

1, [1,2,3, 4,1,2,1] =>

3;

False

Remove(l1l2..ln, p1p2..pm)={ [], n = 0

Remove(l2..ln, p1..pm), count(l1, p1..pm) = 1

L1 + remove(l2..ln, p1..pm), otherwise

```
/*
Remove(l: List, c: List, out: List)
Flow model: (I, I, o), (I, I, I)
*/
Remove([], _, []).
Remove([H|T], C, O) :-
        Count(H, C, 1) ,
        !,
        Remove(T, C, O).
Remove([H|T], C, [H|O]) :-
```

Remove(T, C, O).

<u>Collector variable</u>

<u>->Example without collector variable</u>

Count(1, [1,2,3,4,1,2])   = 2

1 + count (1, [2,3,4,1,2]) = 1+1 = 2

Count (1, [3,4,1,2]) = 1

Count(1, [4,1,2]) = 1

Count (1, [1,2]) = 1

1 + Count(1, [2]) = 1 + 0 = 1

Count (1, []) = 0

0

<u>->Example with collector variable</u>

Count(1, [1,2,3,4,1,2], 0) =2

Count(1, [2,3,4,1,2], 1) =2

Count(1, [3,4,1,2], 1) =2

Count(1, [4,1,2], 1) =2

Count(1, [1,2], 1) =2

Count(1, [2], 2) = 2

Count(1, [], 2) = 2

Count(1, [], 0) = 0

Count(el,l_1,…,l_n,col) = { col , if list is empty

Count(el,l_2,…,l_n,col + 1), if el = l_1

Count(el,l_2,…,l_n,col), otherwise}

% Count(V:int, L:list, Counter:int, Res:int)

Flow : (I,I,I, i), (I,I,I,o)

```
Count(_, [], Counter, Counter).

%Count(_, [], Counter, Res):- Res = Counter.

Count(V, [H|T], L, R) :- V =:=H,

        L1 is  L + 1,

        Count(V,T,L1,R).

Count(V, [H|T], L, R) :-

        Count(V,T,L,R).
```

```
Remove([1,2,3,1,2], [1,2,3,1,2]) = [1,2,1,2]
    1 U remove([2,3,1,2], [1,2,3,1,2]) =[1,2,1,2]
        2 U remove([3,1,2], [1,2,3,1,2]) =[2,1,2]
            Remove([1,2], [1,2,3,1,2]) = [1,2]
                1 U remove([2], [1,2,3,1,2]) =[1,2]
                    2 U remove([], [1,2,3,1,2]) = [2]
                        []
Remove([1,2,3,1,2], [1,2,3,1,2], []) =
    Remove([2,3,1,2], [1,2,3,1,2], [1]) =
        Remove([3,1,2], [1,2,3,1,2], [2,1]) =
            Remove([1,2], [1,2,3,1,2], [2,1]) =
                Remove([2], [1,2,3,1,2], [1,2,1]) =
                    Remove([], [1,2,3,1,2], [2,1,2,1]) =
                        [2,1,2,1]
```

2. given a list of numbers, remove all the increasing sequences of numbers from it. Ex: [1,2,4,6,5,7,8,2,1] => [2, 1]

- Take first two elements. L1 < l2 do not add them
- L1 > l2 add l2 to the list.

[1,2,4,6,5,7,8,2,1]

    [4,6,5,7,8,2,1]

[5,7,8,2,1]

[8,2,1]

2 U [1]

- Add extra parameter, the last removed element

L1 < l2 < l3 => call(l2...ln)

L1 < l2 > l3 => call(l3...ln)

L1 > l2 => l1 U call(l2...ln)

[H1, H2, H3|T]

[H2,H3|T]

[]

[E]

[E1,E2]

Seminar 3: Heterogeneous lists in Prolog

- It is a list in which elements are of different types. Ex [1, 2, a, [1,2,3,4], 5, b, [1,6,7], 7].
- [H|T] - to divide a list in head (H) and tail (T)
    - is_list(H) - checks if H is a list
    - number(H) - checks if H is a number
    - atom(H) - checks if H is a symbol

[H|T]	T = 3	T = [4,5,6]
H = 2	[2 | 3]	[2, 4, 5, 6]
H = [1,2,3]	[[1,2,3] | 3]	[[1,2,3], 4, 5, 6]

Obs: [1,2,3,4,5,6 | …]   w

1. You are given a heterogeneous list, made of numbers and lists of numbers. You will have to remove the odd numbers from the sublists that have a mountain aspect (a list has a mountain aspect if it is made of an increasing sequence of elements, followed by a decreasing sequence of elements).

   Ex: [1, 2, [1, 2, 3, 2], 6, [1, 2], [1,4,5,6,7,1], 8, 2, [4, 3, 1], 11, 5, [6, 7, 6], 8] => [1, 2, [2, 2], 6, [1, 2], [4, 6], 8, 2, [4, 3, 1], 11, 5, [6, 6], 8]

- Check if a list is a mountain
- Removes the odd numbers from a linear list
- Main predicate to process the heterogeneous list

- Check if a list is a mountain
    - Version 1
        - Check (and removes) if it is increasing
        - Check if it is all decreasing when it starts decreasing
    - **Version 2**
        - Extra parameter to see on which "side" of the mountain we are currently
    - Version 3
        - Look for the peak of the mountain (l1 < l2 > l3)
        - Check out for (l1 > l2 < l3)

Parameter Side can be:

- 0 - increasing

- 1 - decreasing

Mountain(L1...Ln, Side) { False if n < 3

True if L1 < L2 > L3 and n = 3

True if L1 > L2 > L3 and Side = 1 and n = 3

Mountain(L2...Ln, Side) if L1 < L2 and Side = 0

Mountain(L2...Ln, 1) if L1 > L2

// Mountain (L2...Ln, 1) if L1 > L2 and Side = 1

False, otherwise

Mountain([5,4,3,2,1] , 0)

Moutain([4,3,2,1], 1)

Mountain([3,2,1], 1)

True

```
% Mountain(L: List, Side: Integer)
% (I, i) - deterministic
Mountain([L1, L2, L3], Side) :-
	L1 < L2, L2 > L3, Side = 0;
	L1 > L2, L2 > L3, Side = 1.
Mountain([H1, H2 | T], 0) :-
	H1 < H2,
	Mountain([H2 | T], 0).
Mountain([H1, H2 | T], _) :-
	H1 > H2,
	Mountain([H2 | T],  1).
```

- Remove all the odd numbers from a linear list

%removeOdd(L:list)

RemoveOdd(l1l2...ln) = { [], n=0

L1 (+) / U removeOdd(l2l3...ln), l1%2 = 0, RemoveOdd(l2l3...ln), otherwise }

```
% flow model: (I,o) deterministic
RemoveOdd([],[]).
RemoveOdd([H|T],[H|Res]):-
    H mod 2 =:= 0,!,
    RemoveOdd(T,Res).
RemoveOdd([_|T],Res):-
    RemoveOdd(T,Res).
```

- Process the heterogeneous list and remove the odd elements from sublists with mountain aspect.

RemoveOddFromMountain(l1l2....ln) = {

[], n = 0

RemoveOdd(l1) (+) RemoveOddFromMountain(l2, ...ln), if l1 is list and l1_1 < l1_2 and Mountain(l1, 0)

l1 (+) RemoveOddFromMountain(l2,...ln), otherwise}

```
%removeOddFromMountain(L:list, Lrez:list)
%(I,o) deterministic
RemoveOddFromMountain([],[]).
RemoveOddFromMountain([H|T], [Res1 | Res2]) :- is_list(H),
    H = [H1, H2|_],
    H1 < H2,
    Mountain(H, 0),!,
    RemoveOdd(H, Res1),
    RemoveOddFromMountain(T, Res2).
```

RemoveOddFromMountain([H|T], [H | Res]) :-

RemoveOddFromMountain(T, Res).

2. Consider the following predicates:

%predicate for odd numbers

%odd(I)

Odd(1).

Odd(3).

Odd(5).

Odd(7).

Odd(9).

%even(o)

Even(X):-odd(N1), odd(N2), X is N1 + N2, X < 9.

Even(X):- odd(N1), X is N1 * 2, X > 9.

?- Even(X).

Will return in this order:

2, 4, 6, 8, 4, 6, 8, 6, 8, 8, 10, 14, 18

Even(X):-!, odd(N1), odd(N2), X is N1 + N2, X < 9.

Even(X):- odd(N1), X is N1 * 2, X > 9.

?- Even(X).

Will return in this order:

2, 4, 6, 8, 4, 6, 8, 6, 8, 8

Even(X):-odd(N1), !, odd(N2), X is N1 + N2, X < 9.

Even(X):- odd(N1), X is N1 * 2, X > 9.

?- Even(X).

Will return in this order:

2, ,4 ,6, 8

Even(X):- odd(N1), odd(N2),!, X is N1 + N2, X < 9.

Even(X):- odd(N1), X is N1 * 2, X > 9.

Will return in this order

2

3. Let’s consider the following predicate

P(E, L, [E|L]).

P(E, [H|T],[H|L1]):-

P(E, T, L1).

What does this predicate do?

- What is the flow model?
- (I,I, o) => insert an element at every possible position in a list – non-determ.
  - P(11, [1,2,3], R).
    - [11, 1, 2, 3]
    - [1, 11, 2, 3]
    - [1, 2, 11, 3]
    - [1 ,2, 3, 11]
- (I,I,I) =>checks if we can get to parameter 3, by inserting E somewhere in the list L.
- (o, I, I) => what element should be inserted in the first list to get the second
  - P(E, [1,2,3], [1,2,5,3])
    - E = 5.
- (o,o, I)
  - P(E, L, [1,2,3,4])
    - E = 1, [2,3,4]
    - E = 2, [1, 3, 4]
    - E = 3, [1, 2, 4]
    - E = 4, [1,2,3]

%(I,I,o)

P(e, l1...ln) =

1. E (+) l1...ln
2. L1 (+) p(e, l2...ln)

Seminar 4: Backtracking in Prolog

1. We are given a sequence a1, a2... an composed of distinct integer numbers. We have to display all the subsequences which have a valley aspect. For example: [5, 3, 4, 2, 7, 11, 1, 8, 6] some of the solutions would be: [5,3,4], [3, 2, 1, 4, 5, 7], [11, 1, 4, 7], etc. (there are 8828 solutions, out of 986328 possibilities).

- We will have a collector variable (a list) in which we will build our solution. We will have a sequence of decreasing elements followed by a sequence of increasing elements in this list.
- We will have a parameter, called Direction, which shows on which part of the valley we are:
  - 0 for the decreasing part
  - 1 for the increasing part
- We will add elements to the beginning of the candidate list:
  - [7]
  - [6, 7]
  - [3, 6, 7]
  - [4, 3, 6, 7]
  - [5, 4, 3, 6, 7]
- There are 4 possible cases:
  - Direction = 1 and element to be added less than the first element of the list (ex add 8 in the list [9, 11]) - we can add the element, Direction stays 1.
  - Direction = 1 and element to be added greater than the first element of the list (ex add 9 in the list [8, 11]) - we can add the element, Direction becomes 0
  - Direction = 0 and element to be added less than the first element of the list(ex: add 7 in the list [9, 8, 11]) - it is not possible
  - Direction = 0 and element to be added greater than the first element of the list (ex: add 10 in the list [9,8, 11]) - we can add the element, Direction stays 0.
- When direction is 0 we have a solution
- How do we generate the elements to be added in the cadidate solution?

GetElem(L1L2...Ln) = { 1. L1

2. GetElem(L2...Ln)

% flow model: (I, 0), (I, i)

% GetElem(list, number)

GetElem([H|_], H).

GetElem([_|T], N) :-

GetElem(T, N).

GetElem([1,2,3,4], N).

N = 1, [2,3,4]

N = 2 [1, 3, 4]

N = 3 [1,2, 4]

N = 4 [1,2,3]

GetElem2(l1l2...ln) = { 1. (l1, l2...ln)

2. (e, l1 U list), (e, list) = getElem2(l2...ln)

% (I, O, O) - non-deterministic

% (list, element, list)

GetElem2([H|T], H, T).

GetElem2([H|T], El, [H|L]) :- GetElem2(T, El, L).

We will use getElem2 in the implementation.

The predicate that generates the solutions:

Parameters:

- Input list
- Direction
- Collector variable
- Result (only in Prolog)

Solution(l1,l2,...ln,d,c1,c2,...cn){1. Solution(list, d, e(+)c), (e, list) = getElem2(l1l2...ln), e <c1, d =1,

2.Solution(list, 0, e(+)c), (e, list) = getElem2(l1l2...ln) e>c1, d=1,

3.Solution(list, d, e(+)c), (e, list) = getElem2(l1l2...ln) e>c1, d=0

4.C, d=0

% solution(list,element,list,list) (I,I,I,o) -non-det

Solution(_,0,Col,Col).

Solution(L,D,[H|T],Res):-

```
        GetElem2(L,E,LNew),
        E < H,
        D =:= 1,
        Solution(LNew,D,[E,H|T],Res).
Solution(L,D,[H|T],Res):-
        GetElem2(L,E,LNew),
        E > H,
        D =:= 1,
        Solution(LNew,0,[E,H|T],Res).
Solution(L,D,[H|T],Res):-
        GetElem2(L,E,LNew),
        E > H,
        D =:= 0,
        Solution(LNew,D,[E,H|T],Res).

?-Solution([1,2,3,4,5], 1, [], R).
        False.
```

SolutionWrapper(l1l2...ln) = solution(list, 1, [e]), (e, list) = getElem2(l1...ln)

[4] 1

[5, 4] 0 => [5,4]

[6,5,4] 0 => [6,5,4]

[4, 9], 1

[5,4,9] 0 =>[5,4,9]

SolutionWrapper(l1l2...ln) = solution(list2, 1, [e1, e2]), (e1, list) = getElem2(l1...ln), (e2, list2) = getElem2(list), e1 < e2

% list, list

% (io)

SolutionWrapper(L1L2Ln, Res):-

GetElem2(L1L2Ln, E1, List1),

GetElem2(List1, E2, List2),

E1 < E2,

Solution(List2, 1, [E1, E2], Res).

SolutionWrapper([1,2,3,4], R).

R = [3,2,4];

R = [4,2,3];

…

SolutionWrapper2(l1...ln) = $\bigcup$ solutionWrapper(l1...ln)

%solutionWrapper2(List, List) (I, o)

SolutionWrapper2(L, R):- findall(Result, solutionWrapper(L, Result),   R).

What if we want to generate only those solutions in which the elements are in the same relative order as in the initial list?

[5, 3, 4, 2, 7, 11, 1, 8, 6]

=> [5,4, 2, 7, 11] - will be generated

=> [7, 4, 11] - will not be generated

[5, 3, 4, 2, 7, 11, 1, 8, 6] => getElem3 returns 7 return also [5, 3, 4, 2]

Return an element and the list until that element

GetElem3(l1l2...ln) = { 1. (l1, [])

2. (e, l1 U list), (e, list) = getElem3(l2...ln)

Lisp:

- Given a list find the minimum odd numerical atom from it.

(7 2 (4 A b 2 (9 2 H 1) K 3) (4 2 7) S) => 1

minimum(l1...ln) = {

- 1000, N=0
- Min(l1, minimum(l2...ln)), L1 is an odd number
- Minimum(l2...ln) l1 is a nonnumerical atom
- Min(minimum(l1), minimum(l2...ln)) l1 is a list
-

Seminar 5: Recursive Programming in Lisp

1. Define a function which merges, without keeping the doubles, two sorted linear lists. Ex: (1 3 5 7 9) and (2 3 6 7 8) => (1 2 3 5 6 7 8 9).

Linear list = list without a sublist

A general list in Lisp (3 6 A B (6 1 D E (T H 7)) 3 2 (T Y 7)).

“on the superficial level” - the list can have sublists, but you can ignore them

MergeLists(l_1,…,l_n , k_1,…,k_m) = {

(), if both lists empty

L_1 U mergeLists(l_2,….,l_n , k_1,…,k_m), if second list empty

K_1 U mergeLists(l_1,…,l_n , k_2,…,k_m), if first list empty

L_1 U mergeLists(l_2,…,l_n , k_1,…,k_m), if l_1 < k_1

L_1 U mergeLists(l_2,…,l_n , k_2,…,k_m), if l_1 = k_1

K_1 U mergeLists(l_1,…,l_n , k_2,…,k_m), otherwise

}

```
(defun mergeLists(L K)
    (cond
        ((and (equal L nil) (equal K nil)) nil)
        ;((and (null L) (null K)) nil)
        ((equal K nil) (cons (car L) (mergeLists (cdr L) K)))
        ((equal L nil) (cons (car K) (mergeLists L (cdr K))))
        ((< (car L) (car K)) (cons (car L) (mergeLists (cdr L) K)))
        ((= (car K) (car L)) (cons (car l) (mergeLists  (cdr L) (cdr K))))
        (t (cons (car k) (mergeLists L (cdr K))))
        ))
```

	cons	list	append

‘A ‘B	(A . B)	(A B)	Error
‘A ‘(B C D)	(A B C D)	(A (B C D))	Error
‘(A B C) ‘(D E F)	((A B C) D E F)	((A B C) (D E F))	(A B C D E F)
‘(A B C) ‘D	((A B C) . D)	((A B C) D)	(A B C . D)
‘A ‘B ‘C ‘D	Error	(A B C D)	Error
‘(A B) ‘(C D) ‘(E F)	Error	((A B) (C D) (E F))	(A B C D E F)
‘(A B) ‘C ‘(D E) ‘F	Error	((A B) C (D E) F)	Error
‘(A B) ‘(D E) ‘F	Error	((A B) (D E) F)	(A B D E . F)

For ‘(A B C) ‘D => (append ‘(A B C) (list ‘D)) =>(A B C D)

For ‘(A B C) ‘(D E F) => (append (list ‘(A B C)) ‘(D E F)) => ((A B C) D E F)

For ‘A ‘(B C D) => (append (list ‘A) ‘(B C D)) => (A B C D)

2. Define a function to remove all the occurrences of an element from a list.

RemoveAll(l1...ln, elem) = {

[], n=0

RemoveAll(l2..ln, elem), l1 atom and l1 = elem

L1 U removeAll(l2..ln, elem), l1 atom and l1 != elem

RemoveAll(l1, elem) U removeAll(l2..ln,elem), l1 list

}

```
(defun removeAll (l elem)
(cond
((null l) nil)
((and (atom (car l)) (equal (car l) elem)) (removeAll (cdr l) elem))
((atom (car l))  (cons (car l) (removeAll (cdr l) elem)))
(t (cons (removeAll (car l) elem) (removeAll (cdr l) elem)))
))
```

3. Build a list with the positions of the minimum number from a linear list.

Ex: (T 7 S 3 A B 3 C 2 5 1 D 1 2 1) => (11 13 15)

PozMin (T 7 S 3 A B 3 C 2 5 1 D 1 2 1) 1 (index) 0 (current minimum) () (pos. of the min) 1(flag)

PozMin (7 S 3 A B 3 C 2 5 1 D 1 2 1) 2 0 () 1

PozMin (S 3 A B 3 C 2 5 1 D 1 2 1) 3 7 (2) 0

PozMin (3 A B 3 C 2 5 1 D 1 2 1)	4	7	(2)	0
PozMin (A B 3 C 2 5 1 D 1 2 1)	5	3	(4)	0
PozMin (B 3 C 2 5 1 D 1 2 1)	6	3	(4)	0
PozMin (3 C 2 5 1 D 1 2 1)	7	3	(4)	0
PozMin (C 2 5 1 D 1 2 1)	8	3	(4 7)	0
PozMin(2 5 1 D 1 2 1)	9	3	(4 7)	0
PozMin(5 1 D 1 2 1)	10	2	(9)	0
PozMin(1 D 1 2 1)	11	2	(9)	0
PozMin(D 1 2 1)	12	1	(11)	0
PozMin(1 2 1)	13	1	(11)	0
PozMin(2 1)	14	1	(11 13)	0
PozMin(1)	15	1	(11 13)	0
PozMin ()	16	1	(11 13 15)	0

Return (11 13 15)

PozMin(l1, l2, …. ln, I, m, o1, o2, … om, f) = o1, o2, … om, if n = 0

PozMin(l2, … ln, I + 1, l1, I, 0) if l1 is a number and f = 1

PozMin(l2, … ln, I + 1, l1, I, f) if l1 is a number and l1 < m

PozMin(l2, … ln, I + 1, m, o1, … om U I, f) if l1 is a number and l1 = m

PozMin(l2, … ln, I + 1, m, o1, … om, f) otherwise

Seminar 6 – MAP functions (MAPCAR)

(defun triple(n) (* n 3))

(MAPCAR #'triple '(1 2 3 4 5 6)) => (list (triple 1) (triple 2) (triple 3) (triple 4) (triple 5) (triple 6)) => (3 6 9 12 15 18)

(MAPCAR #'triple '(1 A 2 B 3 C))? => Error

(defun triple(n)

(cond

  ((numberp n) (* n 3))

  (t n)

)

)

(MAPCAR #'triple '(1 lisA 2 B 3 C)) => (3 A 6 B 9 C)

(MAPCAR #'triple '(1 A (2 B) 3 C)) ? (3 A (2 B) 9 C)

(defun triple(n)

(cond

  ((numberp n) (* 3 n))

  ((atom n) n)

  (t (MAPCAR #'triple n))

)

)

(MAPCAR #'triple '(1 A (2 B) 3 C)) => (3 A (6 B) 9 C)

(triple '(1 A (2 B) 3 C)) => (3 A (6 B) 9 C)

Triple(n) =

3*n, if n is a number

n, if n is a non-numerical atom

$U_{i=1}{}^{n}$ triple(n_I) , otherwise (if n is a list)

- Compute the product of the numerical atoms from a list. Ex: (product '(1 A (2 C 3) 6 (A (B 5) 2) 5)) => 2*3*6*5*2*5 => 1800

Product(n) = { n if n is a number

{ 1 if n is a non numerical atom

{ ∏ i=1, n product(n_i)   if n is a list

(defun Product (n)

(cond

((numberp n) n)

((atom n) 1)

(t (apply #'* (mapcar #'Product n)))

)

)

(prod '(1 A (2 C 3) 6 (A (B 5) 2) 5))=>

(prod 1) => 1

(prod 'A) => 1

(prod '(2 C 3)) => 6

(prod 2) => 2

(prod 'C) => 1

(prod 3) => 3   => (2 1 3) => apply * => 6

(prod 6) => 6

(prod '(A (B 5) 2)) => 10

(prod 'A) => 1

(prod '(B 5)) => 5

(prod ‘B) => 1

(prod 5) => 5 => (1 5) => apply * => 5

(prod 2) => 2 => (1 5 2) => apply * => 10

(prod 5) => 5 => (1 1 6 6 10 5) => apply * => 1800

(apply #’* ‘(1 A 2)) => Error

(apply #’* ‘(1 2 (3 2) 4)) => Error

- Compute the number of nodes from the even levels of an n-ary tree, represented as (root (subtree_1) (subtree_2) … (subtree_n)). The level of the root is 1. Ex:

(A

(B

(D

(G)

(H)

)

(E

(I)

)

)

(C

(F

(J

(L)

)

)

)

(K

(M)

(N)

(O

(P)

)

)

)

=> 8

CountEvenLevelNodes(L, level) = { 1, if level % 2 == 0, L is an atom

0, if level % 2 == 1, L is an atom

Σ I=1,n CountEvenLevelNodes(L_I, level+1) }

```
(defun countEvenLevelNodes(L level)
(cond
((and (atom L) (= 0 (mod level 2))) 1)
;((and (atom L) (evenp level)) 1)
((atom L) 0)x`
        ;(t (apply #'+ (mapcar #'countEvenLevelNodes L  (+ level 1))))
        (t (apply #'+ (mapcar #'(lambda (a) (countEvenLevelNodes a (+ level 1))) L)))
)
)
```

MAIN FUNCTION

CountNodeS(L) = countEvenLevelNodes(L, 0)

(MAPCAR #'CONS '(A B C) '(1 2 3)) => (list (cons A 1) (cons B 2) (cons C 3)) => ((A . 1) (B . 2) (C . 3))

(MAPCAR #'CONS '(A B C) '(1 2)) => ((A . 1) (B . 2))

- You are given a nonlinear list. Compute the number of sublists (including the initial list) where the first numeric atom (on any level) is 5. For example, for the list (A 5 (B C D) 2 1 (G (5 H) 7 D) 11 14) the lists that should be counted are: (5 H), the initial list, (G (5 H) 7 D) => 3

We will have two functions:

- To check if one list fulfils the condition => without mapcar
- To count how many lists we have where the first numerical atom is 5. => MAPCAR

We assume that we have the function verify implemented:

verify (l1..ln) = > true if l1..ln respects the condition

False, otherwise

Count how many lists we have where the first numerical atom is 5.

CountSublists(L) = { 0 if L is an atom

{ 1 + ∑i=1, n CountSublists(L_i) if verify(L) is true

{ ∑i=1, n CountSublists(L_i) otherwise

```
(defun CountSublists (L)
(cond
((atom L) 0)
((verify L) (+ 1 (apply #'+ (mapcar #'CountSublists L))))
(t (apply #'+ (mapcar #'CountSublists L)))
)
)
```

How can we implement verify?

The list is empty => false

L1 is 5 => true

L1 is a number => false

L1 is an atom => verify(l2...ln)

L1 is a list => verify(l1) OR verify(l2...ln)

Solutions:

- Make verify return 3 different values:
  - 0 if first number is 5
  - 1 if firstn umber is not 5
  - 2 if there are no numbers
- Have a function to check if a list contains a number or not.
- Transform the list into a linear one. (optionally: keep only the numbers)

Seminar 7- Recap

Prolog

1. Remove from a list all the elements that occur multiple times. Ex: [1,2,3,2,1] => [3].

This solution is made of 3 functions

- One to check if an element appears in a list
- One to remove all the occurrences of an element from a list
- The main function.

%exists(L:list, E:element)

% flow model (I,I)

exists([H|T], E):- E = H, !.

exists([H|T], E):- exists(T, E).

%removeElem(L:list, E:element, R:list)

%flow model (I,I, o)

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)

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 this implementation going to work?

Exists([1,2,3,2,1], 1) => true ;

Exists([2,3,2,1], 1) => exists([3,2,1], 1) => …. => exists([1], 1) => true

RemoveElem([1,2,3,2,1], 2, R).

1 U RemoveElem([2,3,2,1], 2, R)

RemoveElem([3,2,1], 2, R)

3 U RemoveElem([2,1], 2, R)

RemoveElm([1], 2, R)

1 U removeElem([], 2, R).

[]

RemoveElem ([1,2,3,2,1], 2, R).

=> R = [1, 3, 1];

R = [1, 3, 2, 1];

R = [1, 2, 3, 1];

R = [1,2,3,2,1];

False

2. Combinations...

```
%comb(L:list, N:integer, R:list)
%flow model (I, I, o)
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).
```

- Combinations where elements are in increasing order

Comb([E|_] , 1, [E]).

Comb([_|T] , N, R):-

    Comb(T, N, R).

Comb([H|T], N, [H, H1 | R]):-

    % H < H1,

    N > 1,

    N1 is N – 1,

    Comb(T, N1, [H1 | R]),

    H < H1.

Comb[1,3, 2], 2) => [1,3] [1,2], [3, 2]

Comb => [1,3], [1,2]

Lisp:

a.

(setq car ‘cdr)

(car ‘(1 2 3 4)) => 1

(eval car ‘(1 2 3 4)) => Error Too many parameters passed to eval

(eval (cons car ‘(1 2 3 4))) => Error Too parameters passed to CDR

(cdr 1 2 3 4)

;(cons ‘A ‘(B C D)) => (A B C D)

(eval (list car ‘(1 2 3 4))) => Error 1 is not a function

(cdr (1 2 3 4))

(eval (list car ‘ ‘(1 2 3 4)) => (2 3 4)

(eval car)

Cdr

> car

=> cdr

(eval car)

> cdr => error. Variable CDR has no value.

(apply car '(1 2 3 4 5)) <=> (cdr 1 2 3 4 5) => Error. Too many parameters passed to CDR

(apply #'car '(1 2 3 4 5)) => Error. Too many parameters passed to CAR

(apply car '((1 2 3 4 5))) <=> (cdr '(1 2 3 4 5)) => (2 3 4 5)

(apply #'car '((1 2 3 4 5))) => 1

(funcall car '((1 2 3 4 5))) <=> (Cdr '((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

A'.

(setq a '(1 2 3 4))

(setq b 'a)

(setq c '(length '(1 2 3 4)))

(print a) => (1 2 3 4)

(print b) => A

(print c) => (length '(1 2 3 4))

(eval c) => 4

(eval b) => (1 2 3 4)

(eval a) => Error. 1 is not a function

b.

(mapcar #'list '(1 2 3 4 5)) <=> (list (list 1) (list 2) (list 3) (list 4) (list 5)) => ((1)(2)(3)(4)(5))

(mapcan #'list '(1 2 3 4 5)) <=> (nconc (list 1) (list 2) (list 3) (list 4) (list 5)) => (1 2 3 4 5)

(maplist #'list '(1 2 3 4 5)) <=> (list (list '(1 2 3 4 5)) (list '(2 3 4 5)) (list '(3 4 5)) (list '(4 5)) (list '(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)

(apply #'max '(1 2 3 4 5)) => 5

(eval (append '(+) (mapcar #'max '(1 2 3 4 5)))) => 15

(append '(+) '(1 2 3 4 5)) => (+ 1 2 3 4 5)

(apply #'+ (mapcar #'max '(1 2 3 4 5))) => 15
```

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)  ; (list list list list list list list)

(mapcar #'(lambda(v) '+) x) ; (+ + + + +)

) => (7 9 11 13 15)
```