A.

Bledea Mihaela
Alexandra, 921, A
Subject 18

We will use an auxiliary predicate in order to avoid the recursive calls.

aux(V: number, i: number, Y: number)
aux(i, i, o)

aux(v, i)
= i - 2, if v > 1
= v + 1, otherwise

```
aux(V, i, Y):-
    V > 1,
    !,
    Y is i-2.
aux(V, _, Y):-
    Y is V+1.
```

The auxiliary predicate takes the output of the f(J, V) call, more exactley V.

We will decide based on V > 1 whether it should bind i-2 or V+1 to Y.

To be more clear, we will call like that, only once f(J, V), and we will use the output of the call to check on which case we are. So, instead of calling f(J, V) on the first case and maybe conclude we are not on the right case and go

A.

Bledea Mihaela
Alexandra, 921, A
Subject 1 d

to the next one and compute again f(J, V) ~~all over~~, we call it only once, and check at the end of the call its result with the auxiliary predicate on which case we are and decide the final output.

So the new definition will be:

```
f(0,0):-!
f(i,Y):-
    J is i-1,
    f(J,V),
    aux(V,i,Y).
```

Bledea Mihaela
Alexandra 921 A
Subject 18

B.

insert (elem, $l_1 l_2 \dots l_n$)
$= \{elem\} \cup l_1 l_2 \dots l_n$
$= \{l_1\} \cup$ insert (elem, $l_2 \dots l_n$)

With this predicate we insert an element on every position of a list

insert (E: element, L: the list in which we want to insert the element E, R: the result list)

Flow model: (i, i, o)

```
insert( E, L, [E|L]).
insert( E, [H|T], [H|R]):-
        insert( E, T, R).
```

arr($l_1 l_2 \dots l_n$, k) =
$= l_1$, if $k = 1$
$=$ arr($l_2 \dots l_n$, k) , if $k \geq 1$
$=$ insert (l_1, arr($l_2 \dots l_n$, k - 1)) , if $k > 1$

With this predicate we compute the arrangements.

arr(L: - list from which we take the elements,
k: - the number of elements in the arrangement,
R: - result list)

(i, i, o) -> flow model

B.

Bledea Mihaela
Alexandra 921 A
Subject 18

```
arr([E|_], 1, [E]).
arr([_|T], K, R):-
    arr(T, K, R).
arr([H|T], K, R1):-
    K > 1,
    K1 is K-1,
    arr(T, K1, R),
    insert(H, R, R1).
```

checkIncreasing($l_1 l_2 \ldots l_n$) =
= true, if $n = 2$ and $l_1 < l_2$
= checkIncreasing($l_2 \ldots l_n$), if $l_1 < l_2$
= false otherwise

With this predicate we check if elements of a list are in increasing order.

checkIncreasing(L: list)
(i) -> flow model.

```
checkIncreasing([H1, H2]):-
    H1 < H2.
checkIncreasing([H1, H2|T]):-
    H1 < H2,
    checkIncreasing([H2|T]).
```

B.

Bledea Mihaela
Alexandra 921
Subject 1

computeSum($l_1 l_2 \ldots l_n$) =
= 0, if $n=0$
= l_1 + computeSum($l_2 \ldots l_n$), otherwise

With this predicate we compute the sum of the elements of the list.

computeSum(L: list, R: number)
(i, o) -> flow model

```
computeSum([],0).
computeSum([H|T], R1):-
    computeSum(T,R),
    R1 is R+H.
```

oneSol($l_1 l_2 \ldots l_n$, k)
= arr($l_1 l_2 \ldots l_n$, k) , if checkIncreasing($l_1 \ldots l_n$) = true
and computeSum($l_1 \ldots l_n$) % 2 = 0

oneSol (L: list, K: number, R: result list)
With this predicate we compute one possible solution
(i, i, o) -> flow model

```
oneSol(L, K, R):-
    arr(L, K, R),
    checkIncreasing(R),
    computeSum(R, RS),
    RS mod 2 =:= 0.
```

Bledea Mihaela
Alexandra 921 A
Subject 18

B.

allSols(L, K)

= oneSol(L, K) ∪ ... ∪ oneSol(L, K)

basically, with this predicate we do the reunion of the solutions

allSols(L: list, K: number, R: result list)
(i, i, o) -> flow model

allSols(L, K, R) :-
    findall(RP, oneSol(L, K, RP), R).

What I do here is that I compute all the possible arrangements with k elements, then I check whether they are in increasing order, then I compute their sum and check if the sum is even. If all the conditions are fullfilled, then we add that solution to the final result

Bledea Mihaela
Alexandra, 921 A
Subject 18

C.

linearize (l)

= l, if l is null
= [l], if l is an atom
= linearize (l_1) ∪ .... linearize (l_n), otherwise
(l = l_1 ... l_n)

With this function we linearize a non-linear list.

nodesFromLevel (l, level, k) =
= l, if l is an atom and level = k
= [], if l is an atom
= nodesFromLevel (l_1, level + 1, k) ∪ ... ∪ nodesFromLevel (l_n, level + 1, k), otherwise (l = l_1 l_2 ... l_n)

```
(defun nodesFromLevel (l level k)
  (cond
    ((and (atom l) (equal level k)) l)
    ((atom l) nil)
    (t (apply #' linearize (list (mapcar #' (lambda
(a) (nodesFromLevel a (+ 1 level) k)) l))))
  )
)
```

With this function we take the nodes from a given level.

4
d

Bledea Mihaela
Alexandra _ 921, A
Subject 18

C.

```
(defun main( l k)
    (nodesFromLevel l -1 k)
)
```

↳ This is a wrapper function, where we need to set the level at -1, in order for the root to have the level 0, because mapcar will first take the initial list and only afterwards will go through the lists which represent the actual subtrees.

```
(defun linearize (l)
    (cond
        ((null l) l)
        ((atom l) (list l))
        (t (mapcan #'linearize l))
    )
)
```