```
; A. Fie următoarea definiţie de funcţie LISP
(DEFUN Fct(F L)
  (COND
    ((NULL L) NIL)
    ((FUNCALL F (CAR L)) (CONS (FUNCALL F (CAR L)) (Fct F (CDR L))))
  (T NIL)
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv
(FUNCALL F (CAR L)), fără a redefini logica clauzelor și fără a
; folosi o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați
răspunsul.



(defun fct(f l)
  ((lambda (a)
    (cond
      ((null l) nil)
      (a (cons a (fct f (cdr l))))
      (t nil)
     )
    ) (FUNCALL f (car l))
  )
)

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista submulţimilor
%    cu număr par de elemente. Se vor scrie modelele matematice și
modelele de flux pentru
%    predicatele folosite.
%    Exemplu- pentru lista L=[2,3,4] ⇒ [[],[2,3],[2,4],[3,4]] (nu
neapărat în această ordine)

% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1

% subset(L:list, R:result list)
% (i,o)

subset([],[]).
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
    subset(T,R).

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
```

```
    myLength(T,R),
    R1 is R + 1.


% checkEven(l1l2...ln) =
% = true, if myLength(l1l2...ln) % 2  == 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    myLength(L,N),
    N mod 2 =:= 0.

% oneSol(l1l2...ln) =
% subset(l1l2...ln), if checkEven(l1l2...ln) = true

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkEven(R).

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).
; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice
impare situate pe un nivel par, cu numărul natural succesor.
;    Nivelul superficial se consideră 1. Se va folosi o funcție MAP.
;    Exemplu pentru lista (1 s 4 (3 f (7))) va rezulta (1 s 4 (4 f (7))).



; replaceElems(l, count) =
; = l + 1, if l is a number and l % 2 = 1 and count % 2 = 0
; = l, if l is an atom
; = replaceElems(l1, count + 1) U ... U replaceElems(ln, count + 1),
otherwise (l = l1l2...ln)

(defun replaceElems(l count)
  (cond
    ((and (and (numberp l) (equal (mod l 2 ) 1)) (equal (mod count 2) 0))
(+ 1 l))
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 count))) l))
  )
)

(defun main(l)
  (replaceElems l 0)
)

; A. Fie următoarea definiţie de funcţie în LISP


(DEFUN F(L)
  (COND
```

```
    ((ATOM L) -1)
    ((> (F (CAR L)) 0) (+ (CAR L) (F (CAR L)) (F (CDR L))))
    (T (F (CDR L)))
  )
)
```

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR L)), fără a redefini logica
; clauzelor și fără a folosi o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(defun f(l)
  (cond
    ((atom l) -1)
    (t ((lambda (x)
         (cond
           ((> x 2)(+ (car l) x (f (cdr l))))
           (t (f (cdr l)))
         )
       ) (f (car l))
      )
    )
  )
)
```

% B. Să se scrie un program PROLOG care generează lista submulţimilor cu valori din intervalul
%   [a, b], având număr par de elemente pare și număr impar de elemente impare.
%   Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
% Exemplu- pentru a=2 și b=4 ⇒ [[2,3,4]]

```
% subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)
```

```
countEven([],0).
countEven([H|T],R1):-
    H mod 2 =:= 0,
    !,
    countEven(T,R),
    R1 is R + 1.
countEven([_|T],R):-
    countEven(T,R).

% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countOdd(l2...ln), if l1 % 2 == 1
% = countOdd(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)

countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).


% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,R),
    R=:=0,
    !,
    false.
checkEven(L):-
    countEven(L,R),
    R mod 2 =:= 0.


% checkOdd(l1l2...ln) =
% = true, if countOdd(l1l2...ln) % 2 = 1
% = false, otherwise

% checkOdd(L:list)
% (i)

checkOdd(L):-
    countOdd(L,R),
    R mod 2 =:= 1.

% createList(a, b) =
% = [], if a = b + 1
% = {a} U createList(a + 1, b), otherwise
```

```
% createList(A:number, B:number, R:list)
% (i,i,o)

createList(A,B,[]):-
    A =:= B + 1.
createList(A,B,[A|R]):-
    A1 is A + 1,
    createList(A1,B,R).


% oneSol(l1l2...ln, r1r2...rn) =
% = subsets(l1l2...ln), if checkOdd(subsets(l1l2...ln)) = true and
checkEven(subsets(l1l2...ln)) = true

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subsets(L,R),
    checkOdd(R),
    checkEven(R).


allSols(A,B,R):-
    createList(A,B,L),
    findall(RPartial, oneSol(L, RPartial), R).

; C. Un arbore n-ar se reprezintă în LISP astfel (nod subarbore1
subarbore2 .....).
;    Se cere să se determine lista nodurilor de pe nivelul k.
;    Nivelul rădăcinii se consideră 0. Se va folosi o funcție MAP.
;    Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) k=2 => (g d f)   b) k=5 => ()



; nodesFromLevel(l, level, k) =
; = (list l), if l is an atom and level = k
; = [], if l is an atom
; = nodesFromLevel(l1, level + 1, k) U ... U nodesFromLevel(ln, level +
1, k) , otherwise (l = l1l2...ln)



(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))))
  )
)


; linearize(l) =
; = l, if l is an atom
; = linearize(l1) U ... U linearize(ln), otherwise (l = l1l2...ln)
```

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


; removeNil(l1l2...ln) =
; = [], if n = 0
; = removeNil(l2...ln), if l1 = []
; = {l1} U removeNil(l2...ln), otherwise


(defun removeNil(l)
  (cond
    ((null l) nil)
    ((equal (car l) nil) (removeNil (cdr l)))
    (t (cons (car l) (removeNil (cdr l))))
  )
)
; A. Fie următoarea definiţie de funcţie LISP
(DEFUN F(L)
  (COND
    ((NULL L) NIL)
    ((> (F (CAR L)) 0) (CONS (F (CAR L)) (F (CDR L))))
    (T (F (CAR L)))
  )
)

; Rescrieți această definiție pentru a evita apelul recursiv repetat (F
(CAR L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f(l)
  (cond
    ((null l) nil)
    (t ((lambda (x)
         (cond
           ((> x 0) (cons x (f (cdr l))))
           (t x)
         )
        )(f (car l))
       )
    )
  )
)

% B. Pentru o valoare N dată, să se genereze lista permutărilor cu
elementele N, N+1,….,2*N-1
%   având proprietatea că valoarea absolută a diferenţei dintre două
valori consecutive
%   din permutare este <=2. Se vor scrie modelele matematice și modelele
de
%   flux pentru predicatele folosite.
```

```
% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1, perm(l2...ln)), otherwise

% perm(L:list, R: result list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a >= b
% = b - a, otherwise

absDiff(A,B,R):-
    A >= B,
    R is A - B.
absDiff(A,B,R):-
    A < B,
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) <= 2
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) <= 2
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R =< 2.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R =< 2,
    checkAbsDiff([H2|T]).

% createList(n,m) =
% = [], if n = m + 1
% = {n} U createList(n+1, m), otherwise

% createList(N:number, M:number, R:result list)
```

```
% (i,i,o)

createList(N,M,[]):- N =:= M + 1.
createList(N,M,[N|R]):-
    N1 is N + 1,
    createList(N1,M, R).

% oneSol(l1l2...ln, r1r2...rm) =
% = perm(l1l2...ln, r1r2...rm), if checkAbsDiff(l1l2...ln) = true

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

%allSols(N:number, R:result list)
% (i,o)

allSols(N,R):-
    M is 2 * N - 1,
    createList(N,M,RL),
    findall(RPartial,oneSol(RL,RPartial),R).
; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....)
; Se cere să se înlocuiască nodurile de pe nivelurile impare din arbore
cu o valoare e dată. Nivelul rădăcinii se consideră a fi
; 0. Se va folosi o funcție MAP.
; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f))) şi e=h => (a (h
(g)) (h (d (h)) (h)))


; replaceNodesFromLevel(l, elem, level) =
; = elem, if l is an atom and level % 2 == 1
; = l, if l is an atom
; = replaceNodesFromLevel(l1, elem, level + 1) U ... U
replaceNodesFromLevel(ln, elem, level + 1), otherwise where l = l1l2...ln


(defun replaceNodesFromLevel(l elem level)
  (cond
    ((and (atom l) (equal (mod level 2) 1)) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNodesFromLevel a elem (+ level 1)))
l))
  )
)

(defun main(l elem)
  (replaceNodesFromLevel l elem -1)
)
% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(50, 1):-!.
f(I,Y):-J is I+1, f(J,S), S<1, !, K is I-2, Y is K.
f(I,Y):-J is I+1, f(J,Y).

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
ambele clauze,
```

```
% fără a redefini logica clauzelor. Justificați răspunsul.

f2(50, 1):-!.
f2(I,Y):-
    J is I + 1,
    f2(J,S),
    aux(I,S,Y).

aux(I,S,Y):-
    S < 1,
    !,
    Y is I - 2.
aux(_,S,S).
% B. Dându-se o listă formată din numere întregi, să se genereze lista
submulţimilor
%   cu k elemente numere impare, în progresie aritmetică.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[1,5,2,9,3] şi k=3 ⇒ [[1,5,9],[1,3,5]]
%  (nu neapărat în această ordine)

% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)
```

```
sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 ==
% = countEven(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)

countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).

% checkOdd(l1l2...ln, n) =
% = true, if countOdd(l1l2...ln) = n
% = false, otherwise

% checkOdd(L:list, N:number)
% (i)

checkOdd(L, N):-
    countOdd(L,R),
    R =:= N.

% progression(l1l2...ln) =
% = true, if n = 3 and l2 = (l1 + l2)/2
% = progression(l2...ln), if l2 = (l1 + l2)/2
% = false, otherwise

% progression(L:list)
% (i)

progression([H1,H2,H3]):- H2 =:= (H1 + H3) /2.
progression([H1,H2,H3|T]):-
```

```
    H2 =:= (H1 + H3) /2,
    progression([H2,H3|T]).

% oneSol(l1l2...ln, k) =
% = subsets(l1l2...ln), if checkOdd(subsets(l1l2...ln), K) = true and
% progression(subsets(l1l2...ln)) = true

% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    subsets(L,R),
    checkOdd(R,K),
    progression(R).

% allSols(L:list, K:number, R:result list)
% (i,i,o)

allSols(L,K,R):-
    sortare(L,RL),
    findall(RPartial, oneSol(RL,K,RPartial),R).
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;    lista iniţială in care atomii de pe nivelul k au fost inlocuiti cu 0
;    (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.
; Exemplu pentru lista (a (1 (2 b)) (c (d)))
; a) k=2 => (a (0 (2 b)) (0 (d))) b) k=1 => (0 (1 (2 b)) (c (d))) c) k=4
=>lista nu se modifică


; replaceElems(l, level, k) =
; = 0, if l is an atom and if level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k) U ... U replaceElems(ln, level + 1,
k), otherwise (l = l1l2...ln)

(defun replaceElems(l level k)
  (cond
    ((and (atom l) (equal level k)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k)) l))
  )
)


(defun main(l k)
  (replaceElems l 0 k)
)

; A. Fie următoarea definiţie de funcţie în LISP

(DEFUN F(L)
  (COND
    ((NULL L) NIL)
    ((LISTP (CAR L)) (APPEND (F (CAR L)) (F (CDR L)) (CAR (F (CAR L)))))
    (T (LIST(CAR L)))
  )
)
```

```
; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.



(defun f1(l)
  (cond
    ((null l) nil)
    ((listp (car l)) ((lambda (x) (append (x (f (cdr l)) (car x)))) (f1 (car l))))
    (t (list (car l)))
  )
)
% B. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista submulţimilor cu număr
%    par de elemente. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
% Exemplu- pentru lista L=[2,3,4] ⇒ [[],[2,3],[2,4],[3,4]] (nu neapărat în această ordine)



% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
```

```
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% myLength(l1l2...ln) =
% = 0 , if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.

% checkEven(l1l2...ln) =
% = true, if myLength(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    myLength(L,R),
    R mod 2 =:= 0.

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subsets(L,R),
    checkEven(R).

allSols(L,R):-
    findall(RPartial, oneSol(L,RPartial),R).

; C. Să se substituie valorile numerice cu o valoare e dată, la orice
nivel al unei liste neliniare.
;    Se va folosi o funcție MAP.
```

```
; Exemplu, pentru lista (1 d (2 f (3))), e=0 rezultă lista (0 d (0 f
(0))).



; replaceNumbers(l, elem) =
; = elem, if l is a number
; = l, if l is an atom
; = replaceNumbers(l1, elem) U ... U replaceNumbers(ln, elem), otherwise
(l = l1l2...ln)


(defun replaceNumbers(l elem)
  (cond
    ((numberp l) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNumbers a elem)) l))
  )
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
f(list, integer),
%    având modelul de flux (i, o):

f([], 0).
f([H|T],S):-f(T,S1),S1<H,!,S is H.
f([_|T],S):-f(T,S1),S is S1.

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
ambele clauze, fără a redefini logica clauzelor. Justificați
% răspunsul.


f1([],0).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,H):-
    S1 < H.
aux(S1,_,S1).
% B. Dându-se o listă formată din numere întregi, să se genereze lista
submulţimilor
%   cu k elemente numere impare, în progresie aritmetică.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[1,5,2,9,3] şi k=3 ⇒ [[1,5,9],[1,3,5]]
%  (nu neapărat în această ordine)

% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insertFirst(l2...ln, elem)

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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

```
    insertFirst(T,E,R).


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 ==
% = countEven(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)
```

```
countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).

% checkOdd(l1l2...ln, n) =
% = true, if countOdd(l1l2...ln) = n
% = false, otherwise

% checkOdd(L:list, N:number)
% (i)

checkOdd(L, N):-
    countOdd(L,R),
    R =:= N.

% progression(l1l2...ln) =
% = true, if n = 3 and l2 = (l1 + l2)/2
% = progression(l2...ln), if l2 = (l1 + l2)/2
% = false, otherwise

% progression(L:list)
% (i)

progression([H1,H2,H3]):- H2 =:= (H1 + H3) /2.
progression([H1,H2,H3|T]):-
    H2 =:= (H1 + H3) /2,
    progression([H2,H3|T]).

% oneSol(l1l2...ln, k) =
% = subsets(l1l2...ln), if checkOdd(subsets(l1l2...ln), K) = true and
% progression(subsets(l1l2...ln)) = true

% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    subsets(L,R),
    checkOdd(R,K),
    progression(R).

% allSols(L:list, K:number, R:result list)
% (i,i,o)

allSols(L,K,R):-
    sortare(L,RL),
    findall(RPartial, oneSol(RL,K,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;    lista iniţială in care atomii de pe nivelul k au fost inlocuiti cu 0
;    (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.
; Exemplu pentru lista (a (1 (2 b)) (c (d)))
```

```
; a) k=2 => (a (0 (2 b)) (0 (d))) b) k=1 => (0 (1 (2 b)) (c (d))) c) k=4
=>lista nu se modifică


; replaceElems(l, level, k) =
; = 0, if l is an atom and if level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k) U ... U replaceElems(ln, level + 1,
k), otherwise (l = l1l2...ln)

(defun replaceElems(l level k)
  (cond
    ((and (atom l) (equal level k)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k)) l))
  )
)


(defun main(l k)
  (replaceElems l 0 k)
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
f(list, integer),
%    având modelul de flux (i, o):

f([], -1).
f([H|T],S):-H>0, f(T,S1),S1<H,!,S is H.
f([_|T],S):-f(T,S1), S is S1.

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
ambele clauze, fără a redefini logica clauzelor. Justificați
% răspunsul.

f1([], -1).
f1([H|T],S):-
    H>0,
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,H):-
    S1 < H.
aux(S1,_,S1).
% B. Să se scrie un program PROLOG care generează lista aranjamentelor de
k elemente
%    dintr-o listă de numere întregi, pentru care produsul elementelor e
mai mic decât
%    o valoare V dată. Se vor scrie modelele matematice și modelele de
flux pentru predicatele folosite.
%
% Exemplu- pentru lista [1, 2, 3], k=2 și V=7 ⇒
[[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]] (nu neapărat în această ordine)


% insert(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)
```

```prolog
% insert(L:list, E:element, R:list)
% (i,i,o)

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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) < v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP < V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)

allSols(L,K,V,R):-
    findall(RPartial,oneSol(L,K,V,RPartial),R).
```

```
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să aibă ca rezultat
;    lista iniţială in care atomii de pe nivelul k au fost inlocuiti cu 0
;    (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.
; Exemplu pentru lista (a (1 (2 b)) (c (d)))
; a) k=2 => (a (0 (2 b)) (0 (d))) b) k=1 => (0 (1 (2 b)) (c (d))) c) k=4 =>lista nu se modifică


; replaceElems(l, level, k) =
; = 0, if l is an atom and if level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k) U ... U replaceElems(ln, level + 1, k), otherwise (l = l1l2...ln)

(defun replaceElems(l level k)
  (cond
    ((and (atom l) (equal level k)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k)) l))
  )
)


(defun main(l k)
  (replaceElems l 0 k)
)
```

```
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG având modelul de flux (i, o):

f([],-1).
f([H|T],S):-f(T,S1), S1<1, S is S1-H, !.
f([_|T],S):-f(T,S).

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în ambele clauze, fără a redefini logica clauzelor.
% Justificați răspunsul.

f1([],-1).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
    S1 < 1,
    S is S1 - H.
aux(_,S,S).
% B. Să se scrie un program PROLOG care generează lista submulţimilor de sumă S dată, cu elementele unei liste,
%  astfel încât numărul elementelor pare din submulțime să fie par.
% Exemplu- pentru lista [1, 2, 3, 4, 5, 6, 10] şi S=10 ⇒ [[1,2,3,4],[4,6]].


% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insertFirst(l2...ln, elem)
```

```
% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
```

```
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)

countEven([],0).
countEven([H|T],R1):-
    H mod 2 =:= 0,
    !,
    countEven(T,R),
    R1 is R + 1.
countEven([_|T],R):-
    countEven(T,R).

% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,RL),
    RL mod 2 =:= 0.

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln, s) =
% = true, if computeSum(l1l2...ln) = s
% = false, otherwise

checkSum(L,S):-
    computeSum(L,RS),
    RS =:= S.

% oneSol(L:list, S:number, R:list)
% (i,i,o)

oneSol(L,S,R):-
    subsets(L,R),
    checkEven(R),
    checkSum(R,S).

%allSols(L:list, S:number, R:list)
% (i,i,o)

allSols(L,S,R):-
    sortare(L,LS),
```

```
    findall(RPartial, oneSol(LS,S,RPartial),R).
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să aibă ca rezultat lista
; iniţială din care au fost eliminate toate apariţiile unui element e. Se va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (A)) şi e este A => (1 (2 (3)) NIL)
;         b) dacă lista este (1 (2 (3))) şi e este A => (1 (2 (3)))


; removeElem(l, elem) =
; = nil, if l is an atom and l = elem
; = list(l), if l is an atom
; = removeElem(l1, elem) U ... U removeElem(ln, elem), otherwise (l = l1l2...ln)

(defun removeElem(l elem)
  (cond
    ((and (atom l) (equal l elem)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElem a elem)) l)))
  )
)


(defun main(l elem)
  (car (removeElem l elem))
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG având modelul de flux (i, o):

f([],-1).
f([H|T],S):-f(T,S1),S1>0,!,S is S1+H.
f([_|T],S):-f(T,S1),S is S1.

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în ambele clauze, fără a redefini logica clauzelor.
%  Justificați răspunsul.

f1([],-1).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
    S1 > 0,
    !,
    S is S1 + H.
aux(S1,_,S1).
% B. Să se scrie un program PROLOG care generează lista aranjamentelor de k elemente
%    dintr-o listă de numere întregi, pentru care produsul elementelor e mai mic decât
%    o valoare V dată. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
%
```

```
% Exemplu- pentru lista [1, 2, 3], k=2 și V=7 ⇒
[[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]] (nu neapărat în această ordine)


% insert(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) < v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP < V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
```

```
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)

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

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;  lista iniţială din care au fost eliminaţi toţi atomii numerici pari
situaţi pe un nivel impar.
;  Nivelul superficial se consideră a fi 1. Se va folosi o funcţie MAP.

; Exemplu a) dacă lista este (1 (2 A (4 A)) (6)) => (1 (2 A (A)) (6))
;         b) dacă lista este (1 (2 (C))) => (1 (2 (C)))


; removeElems(l,level) =
; = nil, if l is a number and l % 2 == 0 and level % 2 == 1
; = list(l), if l is an atom
; = removeElems(l1, level + 1) U ... removeElems(ln, level + 1),
otherwise (l = l1l2...ln)

(defun removeElems(l level)
  (cond
    ((and (and (numberp l) (equal (mod l 2) 0)) (equal (mod level 2) 1))
nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElems a (+ 1 level))) l)))
  )
)

(defun main(l)
  (car (removeElems l 0))
)
% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(0, 0):-!.
f(I,Y):-J is I-1, f(J,V), V>1, !, K is I-2, Y is K.
f(I,Y):-J is I-1, f(J,V), Y is V+1.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
ambele clauze, fără a redefini logica clauzelor.
% Justificați răspunsul.

f1(0, 0):-!.
f1(I,Y):-
    J is I-1,
    f1(J,V),
    aux(V,I,Y).

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

```
    Y is V + 1.
% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista aranjamentelor
%   cu număr par de elemente, având suma număr impar. Se vor scrie
modelele matematice și modelele
%   de flux pentru predicatele folosite.

%Exemplu- pentru lista L=[2,3,4] ⇒ [[2,3],[3,2],[3,4],[4,3]] (nu
neapărat în această ordine)


% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(elem, l2...ln)

% insert(E:element, L:list, R:result list)
% (i,i,o)

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

% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

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).

%sum(l1l2...ln) =
% = 0 , if n = 0
% = l1 + sum(l2...ln), otherwise

% sum(L:list, R:number)
% (i,o)

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

% checkSum(L) =
% = true, if sum(L) % 2 == 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    sum(L,R),
    R mod 2 =:= 1.
```

```
% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.

% oneSol(L,K,R):-
% = arr(L,K,R), if checkSum(R) = true

% oneSol(L:list, K:number, R:result list)
% (i,i,o)

oneSol(L,K,R):-
    arr(L,K,R),
    checkSum(R).

% checkEven(n)
% = n, if n % 2 = 0
% = n - 1, otherwise

% checkEven(N:number, R:number)
% (i,o)

checkEven(N,N):-
    N mod 2 =:= 0,
    !.
checkEven(N,N1):-
    N mod 2 =:= 1,
      N1 is N - 1.


% myAppend(l1l2...ln, p1p2...pm) =
% = p1p2...pm, if n = 0
% = {l1} U myAppend(l2...ln, p1p2...pm), otherwise

% myAppend(L:list, P:list, R:list)
% (i,i,o)

myAppend([],P,P).
myAppend([H|T],P,[H|R]):-
    myAppend(T,P,R).


%finalSol(l1l2...ln, k, r, rr) =
% = r, if k = 0
% = finalSol(l1l2...ln, k - 2,
myAppend(findall(RPartial,oneSol(l1l2...ln,k,RPartial),RF),r), rr), if k
> 0

% finalSol(L:list, K:number, R:list, RR:list)
% (i,i,i,o)
```

```
finalSol(_,0,RR,RR):-!.
finalSol(L,K,R,RR):-
    K > 0,
    findall(RPartial,oneSol(L,K,RPartial),RF),
    K1 is K - 2,
    myAppend(RF,R,RRR),
    finalSol(L,K1,RRR,RR).

% main(l1l2...ln, r)
% = finalSol(l1l2...ln, re, [], r),  where re is
checkEven(myLength(l1l2...ln))

% main(L:list, R:list)
% (i,o)

main(L,R):-
    myLength(L,RL),
    checkEven(RL,RE),
    finalSol(L,RE,[],R).

; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....)
; Se cere să se înlocuiască nodurile de pe nivelul k din arbore cu o
valoare e dată. Nivelul rădăcinii se consideră a fi 0.
; Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f))) şi e=h
;  a) k=2 => (a (b (h)) (c (h (e)) (h)))
;  b) k=4 => (a (b (g)) (c (d (e)) (f)))


; replaceElems(l, level, k, elem) =
; = elem, if l is an atom and level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k, elem) U ... U replaceElems(ln, level +
1, k, elem) , otherwise (l = l1l2...ln)

(defun replaceElems(l level k elem)
  (cond
    ((and (atom l) (equal level k)) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k elem)) l))
  )
)

(defun main(l k elem)
  (replaceElems l -1 k elem)
)
; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(G L)
  (COND
    ((NULL L) NIL)
    ((> (FUNCALL G L) 0) (CONS (FUNCALL G L) (F (CDR L))))
    (T (FUNCALL G L))
  )
)
```

```
; Rescrieți această definiție pentru a evita apelul repetat (FUNCALL G
L), fără a redefini logica clauzelor și fără a folosi o funcție
; auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(g l)
  (cond
    ((null l) nil)
    (t ((lambda (x)
         (cond
           ((> x 0) (cons x (f1 (cdr l))))
           (t x)
         )
        )(FUNCALL g l)
       )
    )
  )
)
% B. Să se scrie un program PROLOG care generează lista submulţimilor cu
valori din intervalul
%    [a, b], având număr par de elemente pare și număr impar de elemente
impare.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru a=2 și b=4 ⇒ [[2,3,4]]



% subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)

countEven([],0).
countEven([H|T],R1):-
    H mod 2 =:= 0,
    !,
    countEven(T,R),
    R1 is R + 1.
```

```
countEven([_|T],R):-
    countEven(T,R).

% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 ==
% = countEven(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)

countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).


% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,R),
    R=:=0,
    !,
    false.
checkEven(L):-
    countEven(L,R),
    R mod 2 =:= 0.


% checkOdd(l1l2...ln) =
% = true, if countOdd(l1l2...ln) % 2 = 1
% = false, otherwise

% checkOdd(L:list)
% (i)

checkOdd(L):-
    countOdd(L,R),
    R mod 2 =:= 1.

% createList(a, b) =
% = [], if a = b + 1
% = {a} U createList(a + 1, b), otherwise

% createList(A:number, B:number, R:list)
% (i,i,o)

createList(A,B,[]):-
    A =:= B + 1.
```

```
createList(A,B,[A|R]):-
    A1 is A + 1,
    createList(A1,B,R).


% oneSol(l1l2...ln, r1r2...rn) =
% = subsets(l1l2...ln), if checkOdd(subsets(l1l2...ln)) = true and
checkEven(subsets(l1l2...ln)) = true

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subsets(L,R),
    checkOdd(R),
    checkEven(R).


allSols(A,B,R):-
    createList(A,B,L),
    findall(RPartial, oneSol(L, RPartial), R).

; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....)
; Se cere să se înlocuiască nodurile de pe nivelurile impare din arbore
cu o valoare e dată. Nivelul rădăcinii se consideră a fi
; 0. Se va folosi o funcție MAP.
; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f))) şi e=h => (a (h
(g)) (h (d (h)) (h)))


; replaceNodesFromLevel(l, elem, level) =
; = elem, if l is an atom and level % 2 == 1
; = l, if l is an atom
; = replaceNodesFromLevel(l1, elem, level + 1) U ... U
replaceNodesFromLevel(ln, elem, level + 1), otherwise where l = l1l2...ln


(defun replaceNodesFromLevel(l elem level)
  (cond
    ((and (atom l) (equal (mod level 2) 1)) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNodesFromLevel a elem (+ level 1)))
l))
  )
)

(defun main(l elem)
  (replaceNodesFromLevel l elem -1)
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
f(list, integer), având modelul de flux (i, o):

f([], 0).
f([H|T],S):-f(T,S1),H<S1,!,S is H+S1.
f([_|T],S):-f(T,S1), S is S1+2.
```

```
% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în ambele clauze, fără a redefini logica clauzelor.
%  Justificați răspunsul.

f1([], 0).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
    H < S1,
    !,
    S is H + S1.
aux(S1,_,S):-
    S is S1 + 2.

% B. Să se scrie un program PROLOG care generează lista submulţimilor formate cu elemente unei liste
%   listă de numere întregi, având număr suma elementelor număr impar și număr par nenul de elemente pare.
%   Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
% Exemplu- pentru lista [2,3,4] ⇒ [[2,3,4]]


% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insertFirst(l2...ln, elem)

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
```

```
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)

countEven([],0).
countEven([H|T],R1):-
    H mod 2 =:= 0,
    !,
    countEven(T,R),
    R1 is R + 1.
countEven([_|T],R):-
    countEven(T,R).

% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,RL),
    RL =:= 0,
    !,
    false.
checkEven(L):-
    countEven(L,RL),
    RL mod 2 =:= 0.
```

```
% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 = 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 1.

% oneSol(L:list, S:number, R:list)
% (i,i,o)

oneSol(L,R):-
    subsets(L,R),
    checkEven(R),
    checkSum(R).

%allSols(L:list, S:number, R:list)
% (i,i,o)

allSols(L,R):-
    sortare(L,LS),
    findall(RPartial, oneSol(LS,RPartial),R).
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;    lista iniţială in care atomii de pe nivelul k au fost inlocuiti cu 0
;    (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.
; Exemplu pentru lista (a (1 (2 b)) (c (d)))
; a) k=2 => (a (0 (2 b)) (0 (d))) b) k=1 => (0 (1 (2 b)) (c (d))) c) k=4
=>lista nu se modifică


; replaceElems(l, level, k) =
; = 0, if l is an atom and if level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k) U ... U replaceElems(ln, level + 1,
k), otherwise (l = l1l2...ln)

(defun replaceElems(l level k)
  (cond
    ((and (atom l) (equal level k)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k)) l))
```

```
  )
)


(defun main(l k)
  (replaceElems l 0 k)
)
; A. Fie următoarea definiţie de funcţie LISP
(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (CAR L) 0)
      (COND
        ((> (CAR L) (F (CDR L))) (CAR L))
        (T (F (CDR L)))
      )
    )
    (T (F (CDR L)))
  )
)


; Rescrieți această definiție pentru a evita apelul recursiv repetat (F
(CDR L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(l)
  (cond
    ((null l) 0)
    (((lambda (x)
      (cond
        ((> (car l) 0)
          (cond
            ((> (car l) x) (car l))
            (t x)
          )
        )
        (t x)
      )
     )(f1 (cdr l))
    )
   )
  )
)
% B. Să se scrie un program PROLOG care generează lista aranjamentelor de
k elemente dintr-o listă
%    de numere întregi, având produs P dat. Se vor scrie modelele
matematice și modelele de flux
%    pentru predicatele folosite.

% Exemplu- pentru lista [2, 5, 3, 4, 10], k=2 şi P=20 ⇒
[[2,10],[10,2],[5,4],[4,5]] (nu neapărat în această ordine)


% insert(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)
```

```
% insert(L:list, E:element, R:list)
% (i,i,o)

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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) = v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP = V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)

allSols(L,K,V,R):-
    findall(RPartial,oneSol(L,K,V,RPartial),R).
```

```
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să aibă ca rezultat lista
; iniţială din care au fost eliminate toate apariţiile unui element e. Se va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (A)) şi e este A => (1 (2 (3)) NIL)
;         b) dacă lista este (1 (2 (3))) şi e este A => (1 (2 (3)))


; removeElem(l, elem) =
; = nil, if l is an atom and l = elem
; = list(l), if l is an atom
; = removeElem(l1, elem) U ... U removeElem(ln, elem), otherwise (l = l1l2...ln)

(defun removeElem(l elem)
  (cond
    ((and (atom l) (equal l elem)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElem a elem)) l)))
  )
)


(defun main(l elem)
  (car (removeElem l elem))
)

% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer), având modelul de flux (i, o):

f(100, 0):-!.
f(I,Y):-J is I+1, f(J,V), V>2, !, K is I-2, Y is K+V-1.
f(I,Y):-J is I+1, f(J,V), Y is V+1.

% Rescrieți această definiţie pentru a evita apelul recursiv f(J,V) în ambele clauze, fără a redefini logica clauzelor.
% Justificați răspunsul.

f1(100, 0):-!.
f1(I,Y):-
    J is I+1,
    f1(J,V),
    aux(V,I,Y).

aux(V,I,Y):-
    V > 2,
    !,
    Y is I - 2 + V - 1.
aux(V,_,Y):-
    Y is V + 1.
% B. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista permutărilor având proprietatea
%   că valoarea absolută a diferenţei dintre două valori consecutive din permutare este <=3.
```

```
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru lista L=[2,7,5] ⇒ [[2,5,7], [7,5,2]] (nu neapărat în
această ordine)

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1, perm(l2...ln)), otherwise

% perm(L:list, R: result list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a >= b
% = b - a, otherwise

absDiff(A,B,R):-
    A >= B,
    R is A - B.
absDiff(A,B,R):-
    A < B,
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) <= 3
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) <= 3
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R =< 3.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R =< 3,
    checkAbsDiff([H2|T]).
```

```
% oneSol(l1l2...ln, r1r2...rm) =
% = perm(l1l2...ln, r1r2...rm), if checkAbsDiff(l1l2...ln) = true

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

%allSols(N:number, R:result list)
% (i,o)

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).
; C. Să se substituie un element e prin altul e1 la orice nivel impar al unei liste neliniare.
;    Nivelul superficial se consideră 1.

; De exemplu, pentru lista (1 d (2 d (d))), e=d şi e1=f rezultă lista (1 f (2 d (f))).
; Se va folosi o funcţie MAP.


; replaceElem(l, newElem, elem, level) =
; = elem, if l is an atom and l = elem and level % 2 == 1
; = l, if l is an atom
; = replaceElem(l1, elem, level + 1) U ... U replaceElem(ln, elem, level + 1), otherwise (l = l1l2...ln)


(defun replaceElem (l elem newElem level)
  (cond
    ((and (and (atom l) (equal elem l)) (equal (mod level 2) 1)) newElem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElem a elem newElem (+ 1 level))) l))
  )
)

(defun main(l elem newElem)
  (replaceElem l elem newElem 0)
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG având modelul de flux (i, o):

f([],0).
f([H|T],S):-f(T,S1),S1>=2,!,S is S1+H.
f([_|T],S):-f(T,S1),S is S1+1.


% Rescrieţi această definiţie pentru a evita apelul recursiv f(T,S) în ambele clauze, fără a redefini
% logica clauzelor. Justificaţi răspunsul.

f1([],0).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
```

```
    S1 >= 2,
    !,
    S is S1 + H.
aux(S1,_,S):-
    S is S1 + 1.

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista permutărilor având proprietatea
%   că valoarea absolută a diferenţei dintre două valori consecutive din
permutare este <=3.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru lista L=[2,7,5] ⇒ [[2,5,7], [7,5,2]] (nu neapărat în
această ordine)

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1, perm(l2...ln)), otherwise

% perm(L:list, R: result list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a >= b
% = b - a, otherwise

absDiff(A,B,R):-
    A >= B,
    R is A - B.
absDiff(A,B,R):-
    A < B,
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) <= 3
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) <= 3
% = false, otherwise

% checkAbsDiff(L:list)
% (i)
```

```
checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R =< 3.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R =< 3,
    checkAbsDiff([H2|T]).


% oneSol(l1l2...ln, r1r2...rm) =
% = perm(l1l2...ln, r1r2...rm), if checkAbsDiff(l1l2...ln) = true

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

%allSols(N:number, R:result list)
% (i,o)

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;  lista iniţială din care au fost eliminaţi toţi atomii numerici
multipli de 3.
;  Se va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (6)) => (1 (2 A (A)) NIL)
;         b) dacă lista este (1 (2 (C))) => (1 (2 (C)))


; removeElem(l, elem) =
; = nil, if l is number and l % 3 == 0
; = list(l), if l is an atom
; = removeElem(l1, elem) U ... U removeElem(ln, elem), otherwise (l =
l1l2...ln)

(defun removeElem(l)
  (cond
    ((and (numberp l) (equal (mod l 3) 0)) nil)
    ((atom l) (list l))
    (t (list (mapcan #'removeElem l)))
  )
)


(defun main(l)
  (car (removeElem l))
)
; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CAR L)) 2) (+ (CAR L) (F (CDR L))))
    (T (F (CAR L)))
  )
```

```
)

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(l)
  (cond
    ((null l) 0)
    (((lambda (x)
      (cond
        ((> x 2) (+ (car l) (f1 (cdr l))))
        (t x)
      )
     )(f (car l))
    )
   )
  )
)
% B. Pentru o valoare N dată, să se genereze lista permutărilor cu elementele N, N+1,….,2*N-1
%   având proprietatea că valoarea absolută a diferenţei dintre două valori consecutive
%   din permutare este <=2. Se vor scrie modelele matematice și modelele de
%   flux pentru predicatele folosite.


% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1, perm(l2...ln)), otherwise

% perm(L:list, R: result list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a >= b
% = b - a, otherwise
```

```
absDiff(A,B,R):-
    A >= B,
    R is A - B.
absDiff(A,B,R):-
    A < B,
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) <= 2
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) <= 2
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R =< 2.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R =< 2,
    checkAbsDiff([H2|T]).

% createList(n,m) =
% = [], if n = m + 1
% = {n} U createList(n+1, m), otherwise

% createList(N:number, M:number, R:result list)
% (i,i,o)

createList(N,M,[]):- N =:= M + 1.
createList(N,M,[N|R]):-
    N1 is N + 1,
    createList(N1,M, R).

% oneSol(l1l2...ln, r1r2...rm) =
% = perm(l1l2...ln, r1r2...rm), if checkAbsDiff(l1l2...ln) = true

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

%allSols(N:number, R:result list)
% (i,o)

allSols(N,R):-
    M is 2 * N - 1,
    createList(N,M,RL),
    findall(RPartial,oneSol(RL,RPartial),R).
; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice
care sunt mai mari
;     decât o valoare k dată și sunt situate pe un nivel impar, cu
numărul natural predecesor.
;     Nivelul superficial se consideră 1. Se va folosi o funcție MAP.

; Exemplu pentru lista (1 s 4 (3 f (7))) și
; a) k=0 va rezulta (0 s 3 (3 f (6))) b) k=8 va rezulta (1 s 4 (3 f (7)))
```

```
; replaceNumbers(l, k, level) =
; = l - 1, if l is a number and l > k and level % 2 == 1
; = l, if l is an atom
; = replaceNumbers(l1, k, level + 1) U ... U replaceNumbers(ln, k, level
+ 1), otherwise (l = l1l2...ln)

(defun replaceNumbers(l k level)
  (cond
    ((and (and (numberp l) (> l k)) (equal (mod level 2) 1)) (- l 1))
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNumbers a k (+ 1 level))) l))
  )
)

(defun main(l k)
  (replaceNumbers l k 0)
)

;A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CDR L)) 2) (+ (F (CDR L)) (CAR L)))
    (T (+ (F (CDR L)) 1))
  )
)

; Rescrieți această definiție pentru a evita apelul recursiv repetat (F
(CDR L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.



(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
         (cond
           ((> x 2) (+ x (car l)))
           (t (+ x 1))
         )
        )(f1 (cdr l))
       )
    )
  )
)

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista aranjamentelor
%   cu număr par de elemente, având suma număr impar. Se vor scrie
modelele matematice și modelele
%   de flux pentru predicatele folosite.

%Exemplu- pentru lista L=[2,3,4] ⇒ [[2,3],[3,2],[3,4],[4,3]] (nu
neapărat în această ordine)
```

```
% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(elem, l2...ln)

% insert(E:element, L:list, R:result list)
% (i,i,o)

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

% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

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).

%sum(l1l2...ln) =
% = 0 , if n = 0
% = l1 + sum(l2...ln), otherwise

% sum(L:list, R:number)
% (i,o)

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

% checkSum(L) =
% = true, if sum(L) % 2 == 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    sum(L,R),
    R mod 2 =:= 1.

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
```

```
    myLength(T,R),
    R1 is R + 1.

% oneSol(L,K,R):-
% = arr(L,K,R), if checkSum(R) = true

% oneSol(L:list, K:number, R:result list)
% (i,i,o)

oneSol(L,K,R):-
    arr(L,K,R),
    checkSum(R).

% checkEven(n)
% = n, if n % 2 = 0
% = n - 1, otherwise

% checkEven(N:number, R:number)
% (i,o)

checkEven(N,N):-
    N mod 2 =:= 0,
    !.
checkEven(N,N1):-
    N mod 2 =:= 1,
      N1 is N - 1.


% myAppend(l1l2...ln, p1p2...pm) =
% = p1p2...pm, if n = 0
% = {l1} U myAppend(l2...ln, p1p2...pm), otherwise

% myAppend(L:list, P:list, R:list)
% (i,i,o)

myAppend([],P,P).
myAppend([H|T],P,[H|R]):-
    myAppend(T,P,R).


%finalSol(l1l2...ln, k, r, rr) =
% = r, if k = 0
% = finalSol(l1l2...ln, k - 2,
myAppend(findall(RPartial,oneSol(l1l2...ln,k,RPartial),RF),r), rr), if k
> 0

% finalSol(L:list, K:number, R:list, RR:list)
% (i,i,i,o)

finalSol(_,0,RR,RR):-!.
finalSol(L,K,R,RR):-
    K > 0,
    findall(RPartial,oneSol(L,K,RPartial),RF),
    K1 is K - 2,
    myAppend(RF,R,RRR),
    finalSol(L,K1,RRR,RR).

% main(l1l2...ln, r)
```

```
% = finalSol(l1l2...ln, re, [], r),  where re is
checkEven(myLength(l1l2...ln))

% main(L:list, R:list)
% (i,o)

main(L,R):-
    myLength(L,RL),
    checkEven(RL,RE),
    finalSol(L,RE,[],R).

; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....).
;    Se cere să se determine calea de la radăcină către un nod dat.
;    Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) nod=e => (a c d e) b) nod=v => ()


; myAppend(l1l2...ln, p1p2...pm) =
; = p1p2...pm, if n = 0
; = {l1} U myAppend(l2...ln, p1p2...pm), otherwise

(defun myAppend(l p)
  (cond
    ((null l) p)
    (t (cons (car l) (myAppend (cdr l) p)))
  )
)

; reverseSuperficial(l1l2...ln) =
; = [], if n = 0
; = myAppend(reverseSuperficial(l2...ln), list(l1)), otherwise

(defun reverseSuperficial(l)
  (cond
    ((null l) nil)
    (t (myAppend (reverseSuperficial (cdr l) ) (list (car l))))
  )
)


; linearize(l) =
; = [], if l = []
; = list(l), if l is an atom
; = linearize(l1) U ... U linearize(ln), otherwise (l = l1l2...ln)

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


; path(l, node, collector) =
```

```
; = collector, if l is an atom and l equal node
; = nil, if l is an atom
; = path(l1, node, {l1,1} U collector) U ... U path(ln, node, {ln,1} U
collector), otherwise (where l = l1l2...ln)


(defun path (l node collector)
    (cond
        ((and (atom l) (eq l node)) collector)
        ((atom l) nil)
        (t (apply #'linearize (list (mapcar #'(lambda (a) (path a node
(cons (car l) collector))) l))))
    )
)

(defun pathMain (tree node)
    (reverse (path tree node nil))
)
;A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CAR L)) 2) (+ (F (CDR L)) (F(CAR L))))
    (T (+ (F (CAR L)) 1))
  )
)

; Rescrieți această definiție pentru a evita apelul recursiv repetat (F
(CAR L)), fără a redefini logica clauzelor și fără a folosi
; o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați
răspunsul.


(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
          (cond
            ((> x 2) (+ (f1 (cdr l)) x))
            (t (+ x 1))
          )
        )(f1 (car l))
       )
    )
  )
)
% B. Să se scrie un program PROLOG care generează lista submulţimilor de
sumă pară, cu elementele unei liste.
% Se vor scrie modelele matematice și modelele de flux pentru predicatele
folosite.
% Exemplu- pentru lista L=[2, 3, 4] ⇒ [[],[2],[4],[2,4]] (nu neapărat în
această ordine)

% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1
```

```
% subset(L:list, R:result list)
% (i,o)

subset([],[]).
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
    subset(T,R).


% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 == 0
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 0.


% oneSol(l1l2...ln) =
% subset(l1l2...ln), if checkSum(l1l2...ln) = true

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkSum(R).

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;  lista iniţială din care au fost eliminaţi toţi atomii numerici
multipli de 3.
;  Se va folosi o funcţie MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (6)) => (1 (2 A (A)) NIL)
;         b) dacă lista este (1 (2 (C))) => (1 (2 (C)))


; removeElem(l, elem) =
```

```
; = nil, if l is number and l % 3 == 0
; = list(l), if l is an atom
; = removeElem(l1, elem) U ... U removeElem(ln, elem), otherwise (l = l1l2...ln)

(defun removeElem(l)
  (cond
    ((and (numberp l) (equal (mod l 3) 0)) nil)
    ((atom l) (list l))
    (t (list (mapcan #'removeElem l)))
  )
)


(defun main(l)
  (car (removeElem l))
)
```

```
% A. Fie următoarea definiţie de predicat PROLOG f(list, integer), având modelul de flux (i, o):

f([], -1):-!.
f([_|T], Y):- f(T,S), S<1, !, Y is S+2.
f([H|T], Y):- f(T,S), S<0, !, Y is S+H.
f([_|T], Y):- f(T,S), Y is S.

%Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în clauze, fără a redefini logica clauzelor. Justificați răspunsul.

f1([], -1):-!.
f1([_|T], Y):-
    f1(T,S),
    aux(S,_,Y).


aux(S,_,Y):-
    S < 1,
    !,
    Y is S + 2.
aux(S,H,Y):-
    S < 0,
    !,
    Y is S + H.
aux(S,_,S).

% B. Să se scrie un program PROLOG care generează lista aranjamentelor de k elemente dintr-o listă
%    de numere întregi, având produs P dat. Se vor scrie modelele matematice și modelele de flux
%    pentru predicatele folosite.

% Exemplu- pentru lista [2, 5, 3, 4, 10], k=2 și P=20 ⇒
[[2,10],[10,2],[5,4],[4,5]] (nu neapărat în această ordine)


% insert(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)
```

```
% insert(L:list, E:element, R:list)
% (i,i,o)

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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) = v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP = V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)

allSols(L,K,V,R):-
    findall(RPartial,oneSol(L,K,V,RPartial),R).
```

```
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să aibă ca rezultat
;    lista iniţială din care au fost eliminaţi toţi atomii de pe nivelul k
;    (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d)))
; a) k=2 => (a ((2 b)) ((d)))
; b) k=1 => ((1 (2 b)) (c (d)))
; c) k=4 =>lista nu se modifică



; removeElems(l, k, level) =
; = nil , if l is an atom and level = k
; = (list l), if l is an atom
; = removeElems(l1, k, level + 1) U ... U removeElems(ln, k, level + 1), otherwise (l = l1l2...ln)


(defun removeElems (l k level)
  (cond
    ((and (atom l) (equal level k)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda(a) (removeElems a k (+ level 1))) l)))
  )
)


(defun main(l k)
  (car (removeElems l k 0))
)
; A. Fie următoarea definiţie de funcţie în LISP

(DEFUN F(L1 L2)
  (APPEND (F (CAR L1) L2)
    (COND
      ((NULL L1) (CDR L2))
      (T (LIST (F (CAR L1) L2) (CAR L2)))
    )
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR L1) L2), fără a redefini logica clauzelor și fără a folosi
; o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(l1 l2)
  ((lambda (x)
    (append x
      (cond
        ((null l1) (cdr l2))
        (t (list x (car l2)))
      )
     )
    )(f1 (car l1) l2)
```

```
  )
)

% B. Dându-se o listă formată din numere întregi, să se genereze lista
submulţimilor cu k
%    elemente în progresie aritmetică.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu - pentru lista L=[1,5,2,9,3] şi k=3 ⇒
[[1,2,3],[1,5,9],[1,3,5]] (nu neapărat în această ordine)


% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(elem, l2...ln)

% insert(E:element, L:list, R:list)
% (i,i,o)

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

% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k-1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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(l1l2...ln)
% = true, if n = 2 and l1 < l2
% = checkIncreasing(l2...ln), if l1 < l2
% = false, otherwise

% checkIncreasing(L:list)
% (i)

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

% checkArithMean(l1l2...ln) =
% = true, if n = 3 and l2 = (l1 + l3) / 2
% = checkArithMean(l2...ln), if l2 = (l1 + l3) / 2
% = false, otherwise
```

```
% checkArithMean(L:list)
% (i)

checkArithMean([H1,H2,H3]):-
    H2 =:= (H1 + H3)/2.
checkArithMean([H1,H2,H3|T]):-
    H2 =:= (H1 + H3)/2,
    checkArithMean([H2,H3|T]).


% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    arr(L,K,R),
    checkIncreasing(R),
    checkArithMean(R).

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

; C. Să se substituie valorile numerice cu o valoare e dată, la orice
nivel
;    al unei liste neliniare. Se va folosi o funcție MAP.

; Exemplu, pentru lista (1 d (2 f (3))), e=0 rezultă lista (0 d (0 f
(0))).


; replaceNumbers(l, elem) =
; = elem, if l is a number
; = l, if l is an atom
; = replaceNumbers(l1, elem) U ... U replaceNumbers(ln, elem), otherwise
(l = l1l2...ln)

(defun replaceNumbers(l elem)
  (cond
    ((numberp l) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNumbers a elem)) l))
  )
)

; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(N)
  (COND
    ((= N 1) 1)
    ((> (F (- N 1)) 2) (- N 2))
    ((> (F (- N 1)) 1) (F (- N 1)))
    (T (- (F (- N 1)) 1))
  )
)

; Rescrieți această definiție pentru a evita apelul repetat (F (- N 1)),
fără a redefini logica clauzelor și fără a folosi o funcție
; auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.
```

```
(defun f1(n)
  (cond
    ((= n 1) 1)
    (t ((lambda (x)
         (cond
           ((> x 2) (- n 2))
           ((> x 1) x)
           (t (- x 1))
         )
       )(f (- n 1))
      )
    )
  )
)
```

% B. Să se scrie un program PROLOG care generează lista combinărilor de k elemente
%   cu numere de la 1 la N, având diferența între două numere consecutive din
%   combinare număr par.
%   Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
% Exemplu- pentru N=4, k=2 ⇒ [[1,3],[2,4]] (nu neapărat în această ordine)

```
% comb(l1l2...ln, k) =
% = l1, if k = 1 and n >= 1
% = comb(l2...ln, k), if k >= 1
% = {l1} U comb(l2...ln, k - 1), if k > 1

% comb(L:list, K:number, R:list)
% (i,i,o)

comb([E|_],1,[E]).
comb([_|T],K,R):-
    comb(T,K,R).
comb([H|T],K,[H|R]):-
    K > 1,
    K1 is K - 1,
    comb(T, K1, R).

% createList(n, i) =
% = [], if i = n + 1
% = {i} U createList(n, i + 1), otherwise

% createList(N:number, I:number, R:number)
% (i,i,o)

createList(N,I,[]):-
    I =:= N + 1.
createList(N,I,[I|R]):-
    I1 is I + 1,
    createList(N,I1,R).


% absDiff(a, b) =
```

```
% = a - b, if a > b
% = b - a, otherwise

% absDiff(A:number, B:number, R:number)
% (i,i,o)

absDiff(A,B,R):-
    A > B,
    !,
    R is A - B.
absDiff(A,B,R):-
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) % 2 == 0
% = checkAbsDiff(l2...ln), of absDiff(l1,l2) % 2 == 0
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R mod 2 =:= 0.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R mod 2 =:= 0,
    checkAbsDiff([H2|T]).

% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    comb(L,K,R),
    checkAbsDiff(R).


allSols(N,K,R):-
    createList(N,1,L),
    findall(RPartial, oneSol(L,K,RPartial), R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie care să aibă
ca
;    rezultat lista iniţială in care atomii de pe nivelurile pare au
;    fost înlocuiți cu 0 (nivelul superficial se consideră 1).
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d))) se obține (a (0 (2 b)) (0
(d)))


; replaceAtoms(l level) =
; = 0, if l is an atom and lvel % 2 == 0
; = l, if l is an atom
; = replaceAtoms(l1, level + 1) U replaceAtoms(ln, level + 1), otherwise
(l = l1l2...ln)

(defun replaceAtoms(l level)
```

```
  (cond
    ((and (atom l) (eq (mod level 2) 0)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceAtoms a (+ 1 level))) l))
  )
)

(defun main(l)
  (replaceAtoms l 0)
)

; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CAR L)) 1) (F (CDR L)))
    (T (+ (F (CAR L)) (F (CDR L))))
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR
L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
          (cond
            ((> x 1) (f1 (cdr l)))
            (t (+ x (f1 (cdr l))))
          )
        )(f1 (car l))
      )
    )
  )
)

% B. Să se scrie un program PROLOG care generează lista submulţimilor cu
N elemente,
%    cu elementele unei liste, astfel încât suma elementelor dintr-o
submulțime să
%    fie număr par. Se vor scrie modelele matematice și modelele de flux
pentru predicatele folosite.
% Exemplu- pentru lista L=[1, 3, 4, 2] şi N=2 ⇒ [[1,3], [2,4]]


% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln

% insert(E:element, L:list, R:list)
% (i,i,o)

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

% arr(l1l2...ln, k) =
```

```
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k-1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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(l1l2...ln)
% = true, if n = 2 and l1 < l2
% = checkIncreasing(l2...ln), if l1 < l2
% = false, otherwise

% checkIncreasing(L:list)
% (i)

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

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 == 0
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 0.


% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
```

```
    arr(L,K,R),
    checkIncreasing(R),
    checkSum(R).

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

; C. Un arbore n-ar se reprezintă în LISP astfel
;    ( nod subarbore1 subarbore2 .....).
;    Se cere să se determine calea de la radăcină
;    către un nod dat. Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) nod=e => (a c d e)
; b) nod=v => ()


; myAppend(l1l2..ln, p1p2...pm) =
; = p1p2...pm, if n = 0
; = {l1} U myAppend(l2...ln, p1p2...pm), otherwise

(defun myAppend(l p)
  (cond
    ((null l) p)
    (t (cons (car l) (myAppend (cdr l) p)))
  )
)

; reverseSuperficial(l1l2...ln) =
; = [], if n = 0
; = myAppend(reverseSuperficial(l2...ln), list(l1)), otherwise

(defun reverseSuperficial(l)
  (cond
    ((null l) nil)
    (t (myAppend (reverseSuperficial (cdr l)) (list (car l))))
  )
)

; linearize(l) =
; = [], if l = []
; = list(l), if l is an atom
; = linearize(l1) U ... linearize(ln), otherwise (l = l1l2...ln)

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


; path(l, node, collector)
; = collector, if l is an atom and l = node
; = nil, if l is an atom
; = path(l1, node, {l1,1} U collector) U ... U path(ln, node, {ln,1} U
collector), otherwise (l = l1l2...ln)
```

```
(defun path(l node collector)
  (cond
    ((and (atom l) (eq l node)) collector)
    ((atom l) nil)
    (t (apply #' linearize (list (mapcar #' (lambda (a) (path a node
(cons (car l) collector))) l))))
  )
)


(defun main(l node)
  (reverseSuperficial (path l node nil))
)
```

```
% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(1, 1):-!.
f(K,X):-K1 is K-1, f(K1,Y), Y>1, !, K2 is K1-1, X is K2.
f(K,X):-K1 is K-1, f(K1,Y), Y>0.5, !, X is Y.
f(K,X):-K1 is K-1, f(K1,Y), X is Y-1.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
clauze,
% fără a redefini logica clauzelor. Justificați răspunsul.

f1(1, 1):-!.
f1(K,X):-
    K1 is K-1,
    f1(K1,Y),
    aux(Y,K1,X).

aux(Y,K1,X):-
    Y > 1,
    !,
    X is K1 - 1.
aux(Y,_,Y):-
    Y > 0.5,
    !.
aux(Y,_,X):-
    X is Y - 1.

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista submulţimilor
%    cu cel puţin N elemente având suma divizibilă cu 3.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[2,3,4] și N=1 ⇒ [[3],[2,4],[2,3,4]] (nu
neapărat în această ordine)


% subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1
```

```
% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 3 == 0
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 3 =:= 0.

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.

% checkLength(l1l2...lm, n) =
% = true, if myLength(l1l2...lm) >= n
% = false, otherwise

% checkLength(L:list, N:number)
% (i,i)

checkLength(L,N):-
    myLength(L,RL),
    RL >= N.

% oneSol(L:list, N:number, R:list)
% (i,i,o)
```

```
oneSol(L,N,R):-
    subsets(L,R),
    checkSum(R),
    checkLength(R,N).

allSols(L,N,R):-
    findall(RPartial, oneSol(L,N,RPartial),R).
```

```
; C. Un arbore n-ar se reprezintă în LISP astfel (nod subarbore1
subarbore2 .....).
;    Se cere să se determine numărul de noduri de
;    pe nivelul k. Nivelul rădăcinii se consideră 0.
;    Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) k=2 => nr=3 (g d f) b) k=4 => nr=0 ()


; computeSum(l1l2...ln) =
; = 0, if n = 0
; = computeSum(l1) + computeSum(l2...ln), if l1 is a list
; = l1 + computeSum(l2...ln), otherwise

(defun computeSum (l)
  (cond
    ((null l) 0)
    ((listp (car l)) (+ (computeSum (car l)) (computeSum (cdr l))))
    (t (+ (car l) (computeSum (cdr l))))
  )
)

;  nrNodes(l k level)
; = list(1), if l is an atom and k = level
; = list(0), if l is an atom
; = nrNodes(l1, k, level + 1) + ... + nrNodes(ln, k, level + 1),
otherwise (l = l1l2...ln)

(defun nrNodes(l k level)
  (cond
    ((and (atom l) (eq k level)) 1)
    ((atom l) 0)
    (t (apply #'computeSum (list (mapcar #' (lambda (a) (nrNodes a k (+ 1
level))) l))))
  )
)

(defun main(l k)
  (nrNodes l k -1)
)
; A. Fie G o funcție LISP și fie următoarea definiție

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (G L) 2) (+(G L) (F (CDR L))))
    (T (G L))
  )
```

```
)

; Rescrieți această definiție pentru a evita apelul repetat (G L), fără a redefini logica clauzelor și fără a folosi o funcție
; auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
         (cond
           ((> x 2) (+ x (f1 (cdr l))))
           (t x)
         )
        )(G l)
      )
    )
  )
)

% B. Să se scrie un program PROLOG care generează lista submulţimilor formate cu elemente unei
%    liste listă de numere întregi, având suma elementelor număr impar și număr impar de elemente
%    impare. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

% Exemplu- pentru lista [2,3,4] ⇒ [[2,3],[3,4],[2,3,4]] (nu neapărat în această ordine)


% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
```

```
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countOdd(l2...ln), if l1 % 2 == 1
% = countOdd(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)

countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).

% checkOdd(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 1
% = false, otherwise

% checkOdd(L:list)
% (i)

checkOdd(L):-
    countOdd(L,RL),
    RL mod 2 =:= 1.

% computeSum(l1l2...ln) =
```

```
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 = 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 1.

% oneSol(L:list, S:number, R:list)
% (i,i,o)

oneSol(L,R):-
    subsets(L,R),
    checkOdd(R),
    checkSum(R).

%allSols(L:list, S:number, R:list)
% (i,i,o)

allSols(L,R):-
    sortare(L,LS),
    findall(RPartial, oneSol(LS,RPartial),R).
; C. Un arbore n-ar se reprezintă în LISP astfel (nod subarbore1
subarbore2 .....).
;    Se cere să se determine numărul de noduri de
;    pe nivelul k. Nivelul rădăcinii se consideră 0.
;    Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) k=2 => nr=3 (g d f) b) k=4 => nr=0 ()


; computeSum(l1l2...ln) =
; = 0, if n = 0
; = computeSum(l1) + computeSum(l2...ln), if l1 is a list
; = l1 + computeSum(l2...ln), otherwise

(defun computeSum (l)
  (cond
    ((null l) 0)
    ((listp (car l)) (+ (computeSum (car l)) (computeSum (cdr l))))
    (t (+ (car l) (computeSum (cdr l))))
  )
)
```

```
;  nrNodes(l k level)
; = list(1), if l is an atom and k = level
; = list(0), if l is an atom
; = nrNodes(l1, k, level + 1) + ... + nrNodes(ln, k, level + 1),
otherwise (l = l1l2...ln)

(defun nrNodes(l k level)
  (cond
    ((and (atom l) (eq k level)) 1)
    ((atom l) 0)
    (t (apply #'computeSum (list (mapcar #' (lambda (a) (nrNodes a k (+ 1
level))) l))))
  )
)

(defun main(l k)
  (nrNodes l k -1)
)
% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(20, -1):-!.
f(I,Y):-J is I+1, f(J,V), V>0, !, K is J, Y is K.
f(I,Y):-J is I+1, f(J,V), Y is V-1.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
ambele clauze, fără a redefini
% logica clauzelor. Justificați răspunsul.

f1(20, -1):-!.
f1(I,Y):-
    J is I+1,
    f1(J,V),
      aux(V,J,Y).

aux(V,J,J):-
    V > 0,
    !.
aux(V,_,Y):-
    Y is V - 1.

% B. Să se scrie un program PROLOG care generează lista submulţimilor de
sumă S dată, cu elementele unei liste,
%  astfel încât numărul elementelor pare din submulțime să fie par.
% Exemplu- pentru lista [1, 2, 3, 4, 5, 6, 10] şi S=10 ⇒ [[1,2,3,4],
[4,6]].


% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

insertFirst(L,E,[E|L]).
```

```
% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)

countEven([],0).
countEven([H|T],R1):-
    H mod 2 =:= 0,
```

```
    !,
    countEven(T,R),
    R1 is R + 1.
countEven([_|T],R):-
    countEven(T,R).

% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,RL),
    RL mod 2 =:= 0.

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln, s) =
% = true, if computeSum(l1l2...ln) = s
% = false, otherwise

checkSum(L,S):-
    computeSum(L,RS),
    RS =:= S.

% oneSol(L:list, S:number, R:list)
% (i,i,o)

oneSol(L,S,R):-
    subsets(L,R),
    checkEven(R),
    checkSum(R,S).

%allSols(L:list, S:number, R:list)
% (i,i,o)

allSols(L,S,R):-
    sortare(L,LS),
    findall(RPartial, oneSol(LS,S,RPartial),R).

; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice pare cu numărul natural succesor.
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (1 s 4 (2 f (7))) va rezulta (1 s 5 (3 f (7))).
```

```
; replaceNumbers(l) =
; = l + 1, if l is a number and l % 2 == 0
; = l, if l is an atom
; = replaceNumbers(l1) U ... U replaceNumbers(ln), otherwise (l =
l1l2...ln)

(defun replaceNumbers(l)
  (cond
    ((and (numberp l) (eq (mod l 2) 0)) (+ 1 l))
    ((atom l) l)
    (t (mapcar #' replaceNumbers l))
  )
)

% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(0, -1):-!.
f(I,Y):-J is I-1, f(J,V), V>0, !, K is J, Y is K+V.
f(I,Y):-J is I-1, f(J,V), Y is V+I.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
ambele clauze,
% fără a redefini logica clauzelor. Justificați răspunsul.


f1(0, -1):-!.
f1(I,Y):-
    J is I-1,
    f1(J,V),
    aux(V,J,I,Y).

aux(V,J,_,Y):-
    V > 0,
    !,
    Y is J + V.
aux(V,_,I,Y):-
    Y is I + V.

% B. Să se scrie un program PROLOG care generează lista combinărilor de k
elemente
%    cu numere de la 1 la N, având diferența între două numere
consecutive din
%    combinare număr par.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru N=4, k=2 ⇒ [[1,3],[2,4]] (nu neapărat în această
ordine)

% comb(l1l2...ln, k) =
% = l1, if k = 1 and n >= 1
% = comb(l2...ln, k), if k >= 1
% = {l1} U comb(l2...ln, k - 1), if k > 1

% comb(L:list, K:number, R:list)
% (i,i,o)
```

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

% createList(n, i) =
% = [], if i = n + 1
% = {i} U createList(n, i + 1), otherwise

% createList(N:number, I:number, R:number)
% (i,i,o)

createList(N,I,[]):-
    I =:= N + 1.
createList(N,I,[I|R]):-
    I1 is I + 1,
    createList(N,I1,R).


% absDiff(a, b) =
% = a - b, if a > b
% = b - a, otherwise

% absDiff(A:number, B:number, R:number)
% (i,i,o)

absDiff(A,B,R):-
    A > B,
    !,
    R is A - B.
absDiff(A,B,R):-
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) % 2 == 0
% = checkAbsDiff(l2...ln), of absDiff(l1,l2) % 2 == 0
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R mod 2 =:= 0.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R mod 2 =:= 0,
    checkAbsDiff([H2|T]).

% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    comb(L,K,R),
    checkAbsDiff(R).
```

```
allSols(N,K,R):-
    createList(N,1,L),
    findall(RPartial, oneSol(L,K,RPartial), R).
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca
;    rezultat lista iniţială din care au fost eliminaţi toţi
;    atomii nenumerici de pe nivelurile pare (nivelul superficial se
consideră 1).
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d))) rezultă (a (1 (2 b)) ((d)))


; removeElems(l level)
; = list(l), if l is a number
; = [], if l is an atom and level % 2 == 0
; = list(l), if l is an atom
; = removeElems(l1, level + 1) U ... U removeElems(l2, level + 1),
otherwise

(defun removeElems(l level)
  (cond
    ((numberp l) (list l))
    ((and (atom l) (eq (mod level 2) 0)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElems a (+ 1 level))) l)))
  )
)

(defun main(l)
  (car (removeElems l 0))
)

; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(N)
  (COND
    ((= N 0) 0)
    ((> (F (- N 1)) 1) (- N 2))
    (T (+ (F (- N 1)) 1))
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (- N
1)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(n)
  (cond
    ((= n 0) 0)
    (t ((lambda (x)
         (cond
           ((> x 1) (- n 2))
           (t (+ x 1))
         )
```

```
        )(f1 (- n 1))
       )
     )
   )
 )

% B. Să se scrie un program PROLOG care generează lista permutărilor
mulţimii 1..N,
%    cu proprietatea că valoarea absolută a diferenţei între 2 valori
consecutive
%    din permutare este >=2.
%    Se vor scrie modelele matematice şi modelele de flux pentru
predicatele folosite.

% Exemplu- pentru N=4 ⇒ [[3,1,4,2], [2,4,1,3]] (nu neapărat în această
ordine)


% createList(n,i) =
% = [], if i = n + 1
% = {i} U createList(n, i + 1), otherwise

% createList(N:number, I:number, R:list)
% (i,i,o)

createList(N,I,[]):-
    I =:= N + 1.
createList(N,I,[I|R]):-
    I1 is I + 1,
    createList(N,I1,R).

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(elem,l2...ln)

% insert(E:element, L:list, R:list)
% (i,i,o)

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

% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1,perm(l2...ln)), otherwise

% perm(L:list, R:list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a > b
% = b - a, otherwise
```

```
% absDiff(A:number, B:number, R:number)
% (i,i,o)

absDiff(A,B,R):-
    A > B,
    !,
    R is A - B.
absDiff(A,B,R):-
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if absDiff(l1,l2) >= 2
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) >= 2
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R >= 2.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R >= 2,
    !,
    checkAbsDiff([H2|T]).

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

allSols(N,R):-
    createList(N,1,L),
    findall(RPartial, oneSol(L,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie care să aibă
ca
;    rezultat lista iniţială in care atomii de pe nivelurile pare au
;    fost înlocuiți cu 0 (nivelul superficial se consideră 1).
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d))) se obține (a (0 (2 b)) (0
(d)))


; replaceAtoms(l level) =
; = 0, if l is an atom and lvel % 2 == 0
; = l, if l is an atom
; = replaceAtoms(l1, level + 1) U replaceAtoms(ln, level + 1), otherwise
(l = l1l2...ln)

(defun replaceAtoms(l level)
  (cond
    ((and (atom l) (eq (mod level 2) 0)) 0)
    ((atom l) l)
```

```
    (t (mapcar #' (lambda (a) (replaceAtoms a (+ 1 level))) l))
  )
)

(defun main(l)
  (replaceAtoms l 0)
)

% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(100, 1):-!.
f(K,X):-K1 is K+1, f(K1,Y), Y>1, !, K2 is K1-1, X is K2+Y.
f(K,X):-K1 is K+1, f(K1,Y), Y>0.5, !, X is Y.
f(K,X):-K1 is K+1, f(K1,Y), X is Y-K1.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
clauze, fără a redefini
% logica clauzelor. Justificați răspunsul.

f1(100, 1):-!.
f1(K,X):-
    K1 is K+1,
    f1(K1,Y),
    aux(Y,K1,X).

aux(Y,K1,X):-
    Y>1,
    !,
    X is K1 - 1 + Y.
aux(Y,_,Y):-
    Y > 0.5,
    !.
aux(Y,K1,X):-
    X is Y - K1.

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista submulţimilor
%    cu cel puţin N elemente având suma divizibilă cu 3.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[2,3,4] și N=1 ⇒ [[3],[2,4],[2,3,4]] (nu
neapărat în această ordine)


% subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
```

```
    subsets(T,R).

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 3 == 0
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 3 =:= 0.

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.

% checkLength(l1l2...lm, n) =
% = true, if myLength(l1l2...lm) >= n
% = false, otherwise

% checkLength(L:list, N:number)
% (i,i)

checkLength(L,N):-
    myLength(L,RL),
    RL >= N.

% oneSol(L:list, N:number, R:list)
% (i,i,o)

oneSol(L,N,R):-
    subsets(L,R),
    checkSum(R),
    checkLength(R,N).

allSols(L,N,R):-
    findall(RPartial, oneSol(L,N,RPartial),R).
```

```
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;    lista iniţială din care au fost eliminaţi toţi atomii de pe nivelul
k
;    (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d)))
; a) k=2 => (a ((2 b)) ((d)))
; b) k=1 => ((1 (2 b)) (c (d)))
; c) k=4 =>lista nu se modifică



; removeElems(l, k, level) =
; = nil , if l is an atom and level = k
; = (list l), if l is an atom
; = removeElems(l1, k, level + 1) U ... U removeElems(ln, k, level + 1),
otherwise (l = l1l2...ln)


(defun removeElems (l k level)
  (cond
    ((and (atom l) (equal level k)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda(a) (removeElems a k (+ level 1))) l)))
  )
)


(defun main(l k)
  (car (removeElems l k 0))
)

; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) NIL)
    ((> (F (CAR L)) 0) (CONS (F (CAR L)) (F (CDR L))))
    (T (F (CAR L)))
  )
)

; Rescrieți această definiție pentru a evita apelul recursiv repetat (F
(CAR L)), fără a redefini
; logica clauzelor și fără a folosi o funcție auxiliară. Nu folosiți SET,
SETQ, SETF. Justificați răspunsul


(defun f1(l)
  (cond
    ((null l) nil)
    (t ((lambda (x)
         (cond
           ((> x 0) (cons x (f1 (cdr l))))
           (t x)
         )
```

```
        ) (f1 (car l))
       )
    )
  )
)
% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista aranjamentelor
%    cu N elemente care se termină cu o valoare impară și au suma S dată.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[2,7,4,5,3], N=2 și S=7 ⇒ [[2,5], [4,3]] (nu
neapărat în această ordine)

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

arr([H|_],1,[H]).
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).

% checkLastValue(l1l2...ln) =
% = true, if n = 1 and l1 % 2 == 1
% = checkLastValue(l2...ln, v), if n > 1
% = false, otherwise

% checkLastValue(L:list)
% (i)

checkLastValue([H]):-
    H mod 2 =:= 1.
checkLastValue([_|T]):-
    checkLastValue(T).
```

```
% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln, s) =
% = true, if computeSum(l1l2...ln) = s
% = false, otherwise

% checkSum(L:list, S:number)
% (i,i)

checkSum(L,S):-
    computeSum(L,RS),
    RS =:= S.

% oneSol(L:list,N:number, S:number, R:list)
% (i,i,o)

oneSol(L,N,S,R):-
    arr(L,N,R),
    checkLastValue(R),
    checkSum(R,S).

allSols(L,N,S,R):-
    findall(RPartial, oneSol(L,N,S,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie care să aibă
ca
;    rezultat lista iniţială in care atomii de pe nivelurile pare au
;    fost înlocuiţi cu 0 (nivelul superficial se consideră 1).
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d))) se obține (a (0 (2 b)) (0
(d)))


; replaceAtoms(l level) =
; = 0, if l is an atom and lvel % 2 == 0
; = l, if l is an atom
; = replaceAtoms(l1, level + 1) U replaceAtoms(ln, level + 1), otherwise
(l = l1l2...ln)

(defun replaceAtoms(l level)
  (cond
    ((and (atom l) (eq (mod level 2) 0)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceAtoms a (+ 1 level))) l))
  )
)
```

```
(defun main(l)
  (replaceAtoms l 0)
)
```

% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer), având modelul de flux (i, o):

```
f(100, 0):-!.
f(I,Y):-J is I+1, f(J,V), V>2, !, K is I-2, Y is K+V-1.
f(I,Y):-J is I+1, f(J,V), Y is V+1.
```

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în ambele clauze, fără a
% redefini logica clauzelor. Justificați răspunsul.

```
f1(100, 0):-!.
f1(I,Y):-
    J is I+1,
    f1(J,V),
    aux(V,I,Y).

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

% B. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista aranjamentelor
% cu N elemente care se termină cu o valoare impară și au suma S dată.
% Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

% Exemplu- pentru lista L=[2,7,4,5,3], N=2 și S=7 ⇒ [[2,5], [4,3]] (nu neapărat în această ordine)

```
% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)
```

```
arr([H|_],1,[H]).
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).

% checkLastValue(l1l2...ln) =
% = true, if n = 1 and l1 % 2 == 1
% = checkLastValue(l2...ln, v), if n > 1
% = false, otherwise

% checkLastValue(L:list)
% (i)

checkLastValue([H]):-
    H mod 2 =:= 1.
checkLastValue([_|T]):-
    checkLastValue(T).


% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln, s) =
% = true, if computeSum(l1l2...ln) = s
% = false, otherwise

% checkSum(L:list, S:number)
% (i,i)

checkSum(L,S):-
    computeSum(L,RS),
    RS =:= S.

% oneSol(L:list,N:number, S:number, R:list)
% (i,i,o)

oneSol(L,N,S,R):-
    arr(L,N,R),
    checkLastValue(R),
    checkSum(R,S).

allSols(L,N,S,R):-
    findall(RPartial, oneSol(L,N,S,RPartial),R).
; C. Un arbore n-ar se reprezintă în LISP astfel (nod subarbore1
subarbore2 .....).
```

```
;    Se cere să se determine lista nodurilor de pe nivelul k.
;    Nivelul rădăcinii se consideră 0. Se va folosi o funcție MAP.
;    Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) k=2 => (g d f)   b) k=5 => ()



; nodesFromLevel(l, level, k) =
; = (list l), if l is an atom and level = k
; = [], if l is an atom
; = nodesFromLevel(l1, level + 1, k) U ... U nodesFromLevel(ln, level +
1, k) , otherwise (l = l1l2...ln)



(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))))
  )
)


; linearize(l) =
; = l, if l is an atom
; = linearize(l1) U ... U linearize(ln), otherwise (l = l1l2...ln)

(defun linearize(l)
  (cond
    ((atom l) (list l))
    (t (apply #' removeNil (list (mapcan #' linearize l))))
  )
)


; removeNil(l1l2...ln) =
; = [], if n = 0
; = removeNil(l2...ln), if l1 = []
; = {l1} U removeNil(l2...ln), otherwise


(defun removeNil(l)
  (cond
    ((null l) nil)
    ((equal (car l) nil) (removeNil (cdr l)))
    (t (cons (car l) (removeNil (cdr l))))
  )
)

; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CAR L)) 1) (F (CDR L)))
    (T (+ (F (CAR L)) (F (CDR L))))
```

```
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR
L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
          (cond
            ((< x 1) (f1 (cdr l)))
            (t (+ x (f1 (cdr l))))
          )
        )(f1 (car l))
      )
    )
  )
)

% B. Să se scrie un program PROLOG care generează lista submulţimilor cu
N elemente,
%    cu elementele unei liste, astfel încât suma elementelor dintr-o
submulțime să
%    fie număr par. Se vor scrie modelele matematice și modelele de flux
pentru predicatele folosite.
% Exemplu- pentru lista L=[1, 3, 4, 2] şi N=2 ⇒ [[1,3], [2,4]]


% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln

% insert(E:element, L:list, R:list)
% (i,i,o)

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

% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k-1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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(l1l2...ln)
% = true, if n = 2 and l1 < l2
```

```
% = checkIncreasing(l2...ln), if l1 < l2
% = false, otherwise

% checkIncreasing(L:list)
% (i)

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

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 == 0
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 0.


% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    arr(L,K,R),
    checkIncreasing(R),
    checkSum(R).

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

```
 ; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....)
; Se cere să se înlocuiască nodurile de pe nivelul k din arbore cu o
valoare e dată. Nivelul rădăcinii se consideră a fi 0.
; Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f))) şi e=h
;  a) k=2 => (a (b (h)) (c (h (e)) (h)))
;  b) k=4 => (a (b (g)) (c (d (e)) (f)))
```

```
; replaceElems(l, level, k, elem) =
; = elem, if l is an atom and level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k, elem) U ... U replaceElems(ln, level +
1, k, elem) , otherwise (l = l1l2...ln)

(defun replaceElems(l level k elem)
  (cond
    ((and (atom l) (equal level k)) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k elem)) l))
  )
)

(defun main(l k elem)
  (replaceElems l -1 k elem)
)

; A. Fie următoarea definiţie de funcţie în LISP

(DEFUN F(L)
  (COND
    ((NULL L) NIL)
    ((LISTP (CAR L)) (APPEND (F (CAR L)) (F (CDR L)) (CAR (F (CAR L)))))
    (T (LIST(CAR L)))
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR
L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul


(defun f1(l)
  (cond
    ((null l) nil)
    ((listp (car l)) ((lambda (x) (append x (f1 (cdr l)) (car x))) (f1
(car l))))
    (t (list(car l)))
  )
)

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista permutărilor având proprietatea
%   că valoarea absolută a diferenţei dintre două valori consecutive din
permutare este <=3.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru lista L=[2,7,5] ⇒ [[2,5,7], [7,5,2]] (nu neapărat în
această ordine)

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)
```

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


% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1, perm(l2...ln)), otherwise

% perm(L:list, R: result list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a >= b
% = b - a, otherwise

absDiff(A,B,R):-
    A >= B,
    R is A - B.
absDiff(A,B,R):-
    A < B,
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) <= 3
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) <= 3
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R =< 3.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R =< 3,
    checkAbsDiff([H2|T]).


% oneSol(l1l2...ln, r1r2...rm) =
% = perm(l1l2...ln, r1r2...rm), if checkAbsDiff(l1l2...ln) = true

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

%allSols(N:number, R:result list)
% (i,o)

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).
```

; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice pare cu numărul natural succesor.
;   Se va folosi o funcție MAP.

; Exemplu pentru lista (1 s 4 (2 f (7))) va rezulta (1 s 5 (3 f (7))).

```
; replaceNumbers(l) =
; = l + 1, if l is a number and l % 2 == 0
; = l, if l is an atom
; = replaceNumbers(l1) U ... U replaceNumbers(ln), otherwise (l = l1l2...ln)

(defun replaceNumbers(l)
  (cond
    ((and (numberp l) (eq (mod l 2) 0)) (+ 1 l))
    ((atom l) l)
    (t (mapcar #' replaceNumbers l))
  )
)
```

; A. Fie următoarea definiţie de funcţie LISP

```
(DEFUN F(N)
  (COND
    ((= N 0) 0)
    ((> (F (- N 1)) 1) (- N 2))
    (T (+ (F (- N 1)) 1))
  )
)
```

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (- N 1)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(defun f1(n)
  (cond
    ((= n 0) 0)
    (t ((lambda (x)
         (cond
           ((> x 1) (- n 2))
           (t (+ x 1))
         )
       ) (f1 (- n 1))
      )
    )
  )
)
```

% B. Scrieţi un program PROLOG care determină dintr-o listă formată din numere întregi lista
%   subşirurilor cu cel puţin 2 elemente, formate din elemente în ordine strict crescătoare.
%   Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
% Exemplu- pentru lista [1, 8, 6, 4] ⇒
[[1,8],[1,6],[1,4],[6,8],[4,8],[4,6],[1,4,6],

```
%                                      [1,4,8],[1,6,8],[4,6,8],[1,4,6,8]]
(nu neapărat în această ordine)





% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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

% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).



% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1

% subset(L:list, R:result list)
% (i,o)

subset([],[]).
```

```
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
    subset(T,R).


% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.

% checkLength(l1l2...ln) =
% = true, if myLength(l1l2...ln) >= 2
% = false, otherwise

% checkLength(L:list)
% (i)

checkLength(L):-
    myLength(L,R),
    R >= 2.


% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkLength(R).

allSols(L,R):-
    sortare(L,LS),
    findall(RPartial, oneSol(LS,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie care să aibă
ca
;    rezultat lista iniţială in care atomii de pe nivelurile pare au
;    fost înlocuiți cu 0 (nivelul superficial se consideră 1).
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d))) se obține (a (0 (2 b)) (0
(d)))


; replaceAtoms(l level) =
; = 0, if l is an atom and lvel % 2 == 0
; = l, if l is an atom
; = replaceAtoms(l1, level + 1) U replaceAtoms(ln, level + 1), otherwise
(l = l1l2...ln)

(defun replaceAtoms(l level)
```

```
  (cond
    ((and (atom l) (eq (mod level 2) 0)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceAtoms a (+ 1 level))) l))
  )
)

(defun main(l)
  (replaceAtoms l 0)
)

% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(0, 0):-!.
f(I,Y):-J is I-1, f(J,V), V>1, !, K is I-2, Y is K.
f(I,Y):-J is I-1, f(J,V), Y is V+1.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
ambele clauze,
% fără a redefini logica clauzelor. Justificați răspunsul.

f1(0, 0):-!.
f1(I,Y):-
    J is I-1,
    f1(J,V),
    aux(V,I,Y).

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

% B. Să se scrie un program PROLOG care generează lista submulţimilor cu
suma număr impar,
%  cu valori din intervalul [a, b]. Se vor scrie modelele matematice și
modelele de flux pentru
%   predicatele folosite.

% Exemplu- pentru a=2 și b=4 ⇒ [[2,3],[3,4],[2,3,4]] (nu neapărat în
această ordine)


% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1

% subset(L:list, R:result list)
% (i,o)

subset([],[]).
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
    subset(T,R).
```

```
% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 == 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,S),
    S mod 2 =:= 1.


% createList(a, b) =
% = [], if a = b + 1
% = {a} U createList(a + 1, b), otherwise

% createList(A:number, B:number, R:list)
% (i,i,o)

createList(A,B,[]):-
    A =:= B + 1.
createList(A,B,[A|R]):-
    A1 is A + 1,
    createList(A1,B,R).

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkSum(R).

allSols(A,B,R):-
    createList(A,B,L),
    findall(RPartial,oneSol(L,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca
;    rezultat lista iniţială din care au fost eliminaţi toţi
;    atomii nenumerici de pe nivelurile pare (nivelul superficial se
consideră 1).
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d))) rezultă (a (1 (2 b)) ((d)))
```

```
; removeElems(l level)
; = list(l), if l is a number
; = [], if l is an atom and level % 2 == 0
; = list(l), if l is an atom
; = removeElems(l1, level + 1) U ... U removeElems(l2, level + 1),
otherwise

(defun removeElems(l level)
  (cond
    ((numberp l) (list l))
    ((and (atom l) (eq (mod level 2) 0)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElems a (+ 1 level))) l)))
  )
)

(defun main(l)
  (car (removeElems l 0))
)

% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(100, 1):-!.
f(K,X):-K1 is K+1, f(K1,Y), Y>1, !, K2 is K1-1, X is K2+Y.
f(K,X):-K1 is K+1, f(K1,Y), Y>0.5, !, X is Y.
f(K,X):-K1 is K+1, f(K1,Y), X is Y-K1.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
clauze, fără a redefini
% logica clauzelor. Justificați răspunsul.

f1(100, 1):-!.
f1(K,X):-
    K1 is K+1,
    f1(K1,Y),
    aux(Y,K1,X).

aux(Y,K1,X):-
    Y > 1,
    !,
    X is K1 - 1 + Y.
aux(Y,_,Y):-
    Y > 0.5,
    !.
aux(Y,K1,X):-
    X is Y - K1.

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista aranjamentelor
%   cu număr par de elemente, având suma număr impar. Se vor scrie
modelele matematice și modelele
%   de flux pentru predicatele folosite.

%Exemplu- pentru lista L=[2,3,4] ⇒ [[2,3],[3,2],[3,4],[4,3]] (nu
neapărat în această ordine)
```

```
% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(elem, l2...ln)

% insert(E:element, L:list, R:result list)
% (i,i,o)

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

% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

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).

%sum(l1l2...ln) =
% = 0 , if n = 0
% = l1 + sum(l2...ln), otherwise

% sum(L:list, R:number)
% (i,o)

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

% checkSum(L) =
% = true, if sum(L) % 2 == 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    sum(L,R),
    R mod 2 =:= 1.

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
```

```
    myLength(T,R),
    R1 is R + 1.

% oneSol(L,K,R):-
% = arr(L,K,R), if checkSum(R) = true

% oneSol(L:list, K:number, R:result list)
% (i,i,o)

oneSol(L,K,R):-
    arr(L,K,R),
    checkSum(R).

% checkEven(n)
% = n, if n % 2 = 0
% = n - 1, otherwise

% checkEven(N:number, R:number)
% (i,o)

checkEven(N,N):-
    N mod 2 =:= 0,
    !.
checkEven(N,N1):-
    N mod 2 =:= 1,
      N1 is N - 1.


% myAppend(l1l2...ln, p1p2...pm) =
% = p1p2...pm, if n = 0
% = {l1} U myAppend(l2...ln, p1p2...pm), otherwise

% myAppend(L:list, P:list, R:list)
% (i,i,o)

myAppend([],P,P).
myAppend([H|T],P,[H|R]):-
    myAppend(T,P,R).


%finalSol(l1l2...ln, k, r, rr) =
% = r, if k = 0
% = finalSol(l1l2...ln, k - 2,
myAppend(findall(RPartial,oneSol(l1l2...ln,k,RPartial),RF),r), rr), if k
> 0

% finalSol(L:list, K:number, R:list, RR:list)
% (i,i,i,o)

finalSol(_,0,RR,RR):-!.
finalSol(L,K,R,RR):-
    K > 0,
    findall(RPartial,oneSol(L,K,RPartial),RF),
    K1 is K - 2,
    myAppend(RF,R,RRR),
    finalSol(L,K1,RRR,RR).

% main(l1l2...ln, r)
```

```
% = finalSol(l1l2...ln, re, [], r),  where re is
checkEven(myLength(l1l2...ln))

% main(L:list, R:list)
% (i,o)

main(L,R):-
    myLength(L,RL),
    checkEven(RL,RE),
    finalSol(L,RE,[],R).

; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice
impare situate pe un nivel par,
;    cu numărul natural succesor.
;    Nivelul superficial se consideră 1. Se va folosi o funcție MAP.

; Exemplu pentru lista (1 s 4 (3 f (7))) va rezulta (1 s 4 (4 f (7))).


; replaceElems(l level) =
; = l + 1, if l is a number and l % 2 == 1 and level % 2 == 0
; = l, if l is an atom
; = replaceElems(l1, level + 1) U ... U replaceElems(ln, level + 1),
otherwise (l = l1l2...ln)

(defun replaceElems(l level)
  (cond
    ((and (and (numberp l) (eq (mod l 2) 1)) (eq (mod level 2) 0)) (+ 1
l))
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level))) l))
  )
)

(defun main(l)
  (replaceElems l 0)
)

; A. Fie următoarea definiţie de funcţie în LISP

(DEFUN F(L1 L2)
  (APPEND (F (CAR L1) L2)
    (COND
      ((NULL L1) (CDR L2))
      (T (LIST (F (CAR L1) L2) (CAR L2)))
    )
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR
L1) L2), fără a redefini logica clauzelor și fără a folosi
; o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați
răspunsul.


(defun f1(l1 l2)
  ((lambda (x)
    (append x
```

```
      (cond
        ((null l1) (cdr l2))
        (t (list x (car l2)))
      )
    )
  )(f (car l1) l2)
 )
)
```

% B. Scrieţi un program PROLOG care determină dintr-o listă formată din numere întregi lista
% subşirurilor cu cel puţin 2 elemente, formate din elemente în ordine strict crescătoare.
% Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
% Exemplu- pentru lista [1, 8, 6, 4] ⇒ [[1,8],[1,6],[1,4],[6,8],[4,8],[4,6],[1,4,6],
% [1,4,8],[1,6,8],[4,6,8],[1,4,6,8]] (nu neapărat în această ordine)

```
% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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

% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)
```

```
sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).



% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1

% subset(L:list, R:result list)
% (i,o)

subset([],[]).
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
    subset(T,R).


% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.

% checkLength(l1l2...ln) =
% = true, if myLength(l1l2...ln) >= 2
% = false, otherwise

% checkLength(L:list)
% (i)

checkLength(L):-
    myLength(L,R),
    R >= 2.


% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkLength(R).

allSols(L,R):-
    sortare(L,LS),
    findall(RPartial, oneSol(LS,RPartial),R).
```

```
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să aibă ca rezultat
;  lista iniţială din care au fost eliminaţi toţi atomii numerici multipli de 3.
;  Se va folosi o funcţie MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (6)) => (1 (2 A (A)) NIL)
;         b) dacă lista este (1 (2 (C))) => (1 (2 (C)))


; removeElem(l, elem) =
; = nil, if l is number and l % 3 == 0
; = list(l), if l is an atom
; = removeElem(l1, elem) U ... U removeElem(ln, elem), otherwise (l = l1l2...ln)

(defun removeElem(l)
  (cond
    ((and (numberp l) (equal (mod l 3) 0)) nil)
    ((atom l) (list l))
    (t (list (mapcan #'removeElem l)))
  )
)


(defun main(l)
  (car (removeElem l))
)
% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer), având modelul de flux (i, o):

f(20, -1):-!.
f(I,Y):-J is I+1, f(J,V), V>0, !, K is J, Y is K.
f(I,Y):-J is I+1, f(J,V), Y is V-1.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în ambele clauze,
% fără a redefini logica clauzelor. Justificați răspunsul.


f1(20, -1):-!.
f1(I,Y):-
    J is I+1,
    f1(J,V),
    aux(V,J,Y).

aux(V,J,Y):-
    V > 0,
    !,
    Y is J.
aux(V,_,Y):-
    Y is V - 1.

% B. Să se scrie un program PROLOG care generează lista submulţimilor cu valori din intervalul
%    [a, b], având număr par de elemente pare și număr impar de elemente impare.
```

```
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru a=2 și b=4 ⇒ [[2,3,4]]



% subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)

countEven([],0).
countEven([H|T],R1):-
    H mod 2 =:= 0,
    !,
    countEven(T,R),
    R1 is R + 1.
countEven([_|T],R):-
    countEven(T,R).

% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countOdd(l2...ln), if l1 % 2 == 1
% = countOdd(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)

countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).


% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
```

```
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,R),
    R=:=0,
    !,
    false.
checkEven(L):-
    countEven(L,R),
    R mod 2 =:= 0.


% checkOdd(l1l2...ln) =
% = true, if countOdd(l1l2...ln) % 2 = 1
% = false, otherwise

% checkOdd(L:list)
% (i)

checkOdd(L):-
    countOdd(L,R),
    R mod 2 =:= 1.

% createList(a, b) =
% = [], if a = b + 1
% = {a} U createList(a + 1, b), otherwise

% createList(A:number, B:number, R:list)
% (i,i,o)

createList(A,B,[]):-
    A =:= B + 1.
createList(A,B,[A|R]):-
    A1 is A + 1,
    createList(A1,B,R).


% oneSol(l1l2...ln, r1r2...rn) =
% = subsets(l1l2...ln), if checkOdd(subsets(l1l2...ln)) = true and
checkEven(subsets(l1l2...ln)) = true

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subsets(L,R),
    checkOdd(R),
    checkEven(R).


allSols(A,B,R):-
    createList(A,B,L),
    findall(RPartial, oneSol(L, RPartial), R).
```

```
; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....)
; Se cere să se înlocuiască nodurile de pe nivelul k din arbore cu o
valoare e dată. Nivelul rădăcinii se consideră a fi 0.
; Se va folosi o funcţie MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f))) şi e=h
;  a) k=2 => (a (b (h)) (c (h (e)) (h)))
;  b) k=4 => (a (b (g)) (c (d (e)) (f)))


; replaceElems(l, level, k, elem) =
; = elem, if l is an atom and level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k, elem) U ... U replaceElems(ln, level +
1, k, elem) , otherwise (l = l1l2...ln)

(defun replaceElems(l level k elem)
  (cond
    ((and (atom l) (equal level k)) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k elem)) l))
  )
)

(defun main(l k elem)
  (replaceElems l -1 k elem)
)
; A. Fie următoarea definiţie de funcţie LISP


(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (CAR L) 0)
      (COND
        ((> (CAR L) (F (CDR L))) (CAR L))
        (T (F (CDR L)))
      )
    )
    (T (F (CDR L)))
  )
)



; Rescrieţi această definiţie pentru a evita apelul recursiv repetat (F
(CDR L)), fără a redefini logica clauzelor şi fără a folosi o
; funcţie auxiliară. Nu folosiţi SET, SETQ, SETF. Justificaţi răspunsul.

(defun f1(l)
  (cond
    ((null l) 0)
    (((lambda (x)
      (cond
        ((> (car l) 0)
          (cond
            ((> (car l) x) (car l))
            (t x)
```

```
          )
        )
        (t x)
       )
      )(f1 (cdr l))
     )
    )
  )
)
% B. Să se scrie un program PROLOG care generează lista aranjamentelor de
k elemente dintr-o listă
%    de numere întregi, având produs P dat. Se vor scrie modelele
matematice şi modelele de flux
%    pentru predicatele folosite.

% Exemplu- pentru lista [2, 5, 3, 4, 10], k=2 şi P=20 ⇒
[[2,10],[10,2],[5,4],[4,5]] (nu neapărat în această ordine)


% insert(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
```

```
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) = v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP = V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)

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

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat lista
; iniţială din care au fost eliminate toate apariţiile unui element e. Se
va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (A)) şi e este A => (1 (2
(3)) NIL)
;         b) dacă lista este (1 (2 (3))) şi e este A => (1 (2 (3)))


; removeElem(l, elem) =
; = nil, if l is an atom and l = elem
; = list(l), if l is an atom
; = removeElem(l1, elem) U ... U removeElem(ln, elem), otherwise (l =
l1l2...ln)

(defun removeElem(l elem)
  (cond
    ((and (atom l) (equal l elem)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElem a elem)) l)))
  )
)


(defun main(l elem)
  (car (removeElem l elem))
)

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista submulţimilor
%    cu număr par de elemente. Se vor scrie modelele matematice și
modelele de flux pentru
%    predicatele folosite.
```

```
%    Exemplu- pentru lista L=[2,3,4] ⇒ [[],[2,3],[2,4],[3,4]] (nu
neapărat în această ordine)

% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1

% subset(L:list, R:result list)
% (i,o)

subset([],[]).
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
    subset(T,R).

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.


% checkEven(l1l2...ln) =
% = true, if myLength(l1l2...ln) % 2  == 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    myLength(L,N),
    N mod 2 =:= 0.

% oneSol(l1l2...ln) =
% subset(l1l2...ln), if checkEven(l1l2...ln) = true

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkEven(R).

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;  lista iniţială din care au fost eliminați toți atomii numerici pari
situați pe un nivel impar.
```

```
;  Nivelul superficial se consideră a fi 1. Se va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (4 A)) (6)) => (1 (2 A (A)) (6))
;         b) dacă lista este (1 (2 (C))) => (1 (2 (C)))


; removeElems(l,level) =
; = nil, if l is a number and l % 2 == 0 and level % 2 == 1
; = list(l), if l is an atom
; = removeElems(l1, level + 1) U ... removeElems(ln, level + 1),
otherwise (l = l1l2...ln)

(defun removeElems(l level)
  (cond
    ((and (and (numberp l) (equal (mod l 2) 0)) (equal (mod level 2) 1))
nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElems a (+ 1 level))) l)))
  )
)

(defun main(l)
  (car (removeElems l 0))
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
f(list, integer), având modelul
%  de flux (i, o):

f([], 0).
f([H|T],S):-f(T,S1),S1<H,!,S is H.
f([_|T],S):-f(T,S1),S is S1.

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
ambele clauze, fără a redefini
% logica clauzelor. Justificați răspunsul.

f1([], 0).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,H):-
    S1 < H,
    !.
aux(S1,_,S1).

% B. Să se scrie un program PROLOG care generează lista aranjamentelor de
k elemente
%    dintr-o listă de numere întregi, pentru care produsul elementelor e
mai mic decât
%    o valoare V dată. Se vor scrie modelele matematice și modelele de
flux pentru predicatele folosite.
%
% Exemplu- pentru lista [1, 2, 3], k=2 și V=7 ⇒
[[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]] (nu neapărat în această ordine)


% insert(l1l2...ln, elem) =
```

```
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) < v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP < V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)
```

```
allSols(L,K,V,R):-
    findall(RPartial,oneSol(L,K,V,RPartial),R).

; C. Un arbore n-ar se reprezintă în LISP astfel (nod subarbore1
subarbore2 .....).
;    Se cere să se verifice dacă un nod x apare pe un nivel par în
arbore.
;    Nivelul rădăcinii se consideră a fi 0. Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) x=g => T
; b) x=h => NIL


; exists(l1l2...ln) =
; = false, if n = 0
; = true, if l1 is true
; = exists(l2...ln), otherwise

(defun exists(l)
  (cond
    ((null l) nil)
    ((eq (car l) t) t)
    ((listp (car l)) (or (exists (car l)) (exists (cdr l))))
    (t (exists (cdr l)))
  )
)

; checkExistence(l, node, level) =
; = true, if l is an atom and l = node and level % 2 == 0
; = false, if l is an atom
; = checkExistence(l1, node, level + 1) U ... U checkExistence(ln, node,
level + 1), otherwise (l = l1l2...ln)



(defun checkExistence(l node level)
  (cond
    ((and (and (atom l) (eq l node)) (eq (mod level 2) 0)) (list T))
    ((atom l) (list nil))
    (t (apply #' exists (list (mapcar #' (lambda (a) (checkExistence a
node (+ 1 level))) l))))
  )
)

(defun main(l node)
  (checkExistence l node -1)
)
; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CAR L)) 2) (+ (F (CDR L)) (F(CAR L))))
    (T (+ (F (CAR L)) 1))
  )
)
```

; Rescrieți această definiție pentru a evita apelul recursiv repetat (F (CAR L)), fără a redefini logica clauzelor și fără a folosi
; o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
         (cond
           ((> x 2) (+ (f1 (cdr l)) x))
           (t (+ x 1))
         )
       )(f1 (car l))
      )
    )
  )
)
```

```
% B. Să se scrie un program PROLOG care generează lista submulţimilor formate cu elemente unei liste
%   listă de numere întregi, având număr suma elementelor număr impar și număr par nenul de elemente pare.
%   Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
% Exemplu- pentru lista [2,3,4] ⇒ [[2,3,4]]


% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insertFirst(l2...ln, elem)

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
```

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

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)

countEven([],0).
countEven([H|T],R1):-
    H mod 2 =:= 0,
    !,
    countEven(T,R),
    R1 is R + 1.
countEven([_|T],R):-
    countEven(T,R).

% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,RL),
    RL =:= 0,
    !,
```

```
    false.
checkEven(L):-
    countEven(L,RL),
    RL mod 2 =:= 0.

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 = 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 1.

% oneSol(L:list, S:number, R:list)
% (i,i,o)

oneSol(L,R):-
    subsets(L,R),
    checkEven(R),
    checkSum(R).

%allSols(L:list, S:number, R:list)
% (i,i,o)

allSols(L,R):-
    sortare(L,LS),
    findall(RPartial, oneSol(LS,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;  lista iniţială din care au fost eliminaţi toţi atomii numerici pari
situaţi pe un nivel impar.
;  Nivelul superficial se consideră a fi 1. Se va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (4 A)) (6)) => (1 (2 A (A)) (6))
;         b) dacă lista este (1 (2 (C))) => (1 (2 (C)))


; removeElems(l,level) =
; = nil, if l is a number and l % 2 == 0 and level % 2 == 1
; = list(l), if l is an atom
; = removeElems(l1, level + 1) U ... removeElems(ln, level + 1),
otherwise (l = l1l2...ln)
```

```
(defun removeElems(l level)
  (cond
    ((and (and (numberp l) (equal (mod l 2) 0)) (equal (mod level 2) 1))
nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElems a (+ 1 level))) l)))
  )
)

(defun main(l)
  (car (removeElems l 0))
)
% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(0, -1):-!.
f(I,Y):-J is I-1, f(J,V), V>0, !, K is J, Y is K+V.
f(I,Y):-J is I-1, f(J,V), Y is V+I.

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
ambele clauze,
% fără a redefini logica clauzelor. Justificați răspunsul.


f1(0, -1):-!.
f1(I,Y):-
    J is I-1,
    f1(J,V),
    aux(V,J,I,Y).

aux(V,J,_,Y):-
    V > 0,
    !,
    Y is J + V.
aux(V,_,I,Y):-
    Y is I + V.

% B. Să se scrie un program PROLOG care generează lista submulţimilor
formate cu elemente unei liste
%   listă de numere întregi, având număr suma elementelor număr impar și
număr par nenul de elemente pare.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru lista [2,3,4] ⇒ [[2,3,4]]


% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insertFirst(l2...ln, elem)

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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

```
% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countEven(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 == 0
% = countEven(l2...ln), otherwise

% countEven(L:list, R:number)
% (i,o)

countEven([],0).
countEven([H|T],R1):-
```

```
    H mod 2 =:= 0,
    !,
    countEven(T,R),
    R1 is R + 1.
countEven([_|T],R):-
    countEven(T,R).

% checkEven(l1l2...ln) =
% = true, if countEven(l1l2...ln) % 2 = 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    countEven(L,RL),
    RL =:= 0,
    !,
    false.
checkEven(L):-
    countEven(L,RL),
    RL mod 2 =:= 0.

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 = 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 1.

% oneSol(L:list, S:number, R:list)
% (i,i,o)

oneSol(L,R):-
    subsets(L,R),
    checkEven(R),
    checkSum(R).

%allSols(L:list, S:number, R:list)
% (i,i,o)

allSols(L,R):-
```

```
    sortare(L,LS),
    findall(RPartial, oneSol(LS,RPartial),R).
```

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să aibă ca rezultat
;   lista iniţială in care atomii de pe nivelul k au fost inlocuiti cu 0
;   (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.
; Exemplu pentru lista (a (1 (2 b)) (c (d)))
; a) k=2 => (a (0 (2 b)) (0 (d))) b) k=1 => (0 (1 (2 b)) (c (d))) c) k=4 =>lista nu se modifică

```
; replaceElems(l, level, k) =
; = 0, if l is an atom and if level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k) U ... U replaceElems(ln, level + 1, k), otherwise (l = l1l2...ln)

(defun replaceElems(l level k)
  (cond
    ((and (atom l) (equal level k)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k)) l))
  )
)


(defun main(l k)
  (replaceElems l 0 k)
)
```

; A. Fie următoarea definiţie de funcţie LISP

```
(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CAR L)) 2) (+ (CAR L) (F (CDR L))))
    (T (F (CAR L)))
  )
)
```

; Rescrieți această definiție pentru a evita dublul apel recursiv (F (CAR L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
          (cond
            ((> x 2) (+ (car l) (f1 (cdr l))))
            (t x)
          )
        )(f1 (car l))
      )
    )
  )
```

```
)% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista submulţimilor
%    cu număr par de elemente. Se vor scrie modelele matematice și
modelele de flux pentru
%    predicatele folosite.
%    Exemplu- pentru lista L=[2,3,4] ⇒ [[],[2,3],[2,4],[3,4]] (nu
neapărat în această ordine)

% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1

% subset(L:list, R:result list)
% (i,o)

subset([],[]).
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
    subset(T,R).

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.


% checkEven(l1l2...ln) =
% = true, if myLength(l1l2...ln) % 2  == 0
% = false, otherwise

% checkEven(L:list)
% (i)

checkEven(L):-
    myLength(L,N),
    N mod 2 =:= 0.

% oneSol(l1l2...ln) =
% subset(l1l2...ln), if checkEven(l1l2...ln) = true

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkEven(R).

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).
```

```
; C. Un arbore n-ar se reprezintă în LISP astfel (nod subarbore1
subarbore2 .....).
;    Se cere să se verifice dacă un nod x apare pe un nivel par în
arbore.
;    Nivelul rădăcinii se consideră a fi 0. Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) x=g => T
; b) x=h => NIL


; exists(l1l2...ln) =
; = false, if n = 0
; = true, if l1 is true
; = exists(l2...ln), otherwise

(defun exists(l)
  (cond
    ((null l) nil)
    ((eq (car l) t) t)
    ((listp (car l)) (or (exists (car l)) (exists (cdr l))))
    (t (exists (cdr l)))
  )
)

; checkExistence(l, node, level) =
; = true, if l is an atom and l = node and level % 2 == 0
; = false, if l is an atom
; = checkExistence(l1, node, level + 1) U ... U checkExistence(ln, node,
level + 1), otherwise (l = l1l2...ln)



(defun checkExistence(l node level)
  (cond
    ((and (and (atom l) (eq l node)) (eq (mod level 2) 0)) (list T))
    ((atom l) (list nil))
    (t (apply #' exists (list (mapcar #' (lambda (a) (checkExistence a
node (+ 1 level))) l))))
  )
)

(defun main(l node)
  (checkExistence l node -1)
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
f(list, integer), având modelul de flux (i, o):

f([], 0).
f([H|T],S):-f(T,S1),H<S1,!,S is H+S1.
f([_|T],S):-f(T,S1), S is S1+2.

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
ambele clauze, fără a redefini logica clauzelor.
%  Justificați răspunsul.

f1([], 0).
```

```
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
    H < S1,
    !,
    S is H + S1.
aux(S1,_,S):-
    S is S1 + 2.

% B. Să se scrie un program PROLOG care generează lista permutărilor
mulţimii 1..N,
%    cu proprietatea că valoarea absolută a diferenţei între 2 valori
consecutive
%    din permutare este >=2.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru N=4 ⇒ [[3,1,4,2], [2,4,1,3]] (nu neapărat în această
ordine)


% createList(n,i) =
% = [], if i = n + 1
% = {i} U createList(n, i + 1), otherwise

% createList(N:number, I:number, R:list)
% (i,i,o)

createList(N,I,[]):-
    I =:= N + 1.
createList(N,I,[I|R]):-
    I1 is I + 1,
    createList(N,I1,R).

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(elem,l2...ln)

% insert(E:element, L:list, R:list)
% (i,i,o)

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

% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1,perm(l2...ln)), otherwise

% perm(L:list, R:list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).
```

```
% absDiff(a,b) =
% = a - b, if a > b
% = b - a, otherwise

% absDiff(A:number, B:number, R:number)
% (i,i,o)

absDiff(A,B,R):-
    A > B,
    !,
    R is A - B.
absDiff(A,B,R):-
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if absDiff(l1,l2) >= 2
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) >= 2
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R >= 2.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R >= 2,
    !,
    checkAbsDiff([H2|T]).

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

allSols(N,R):-
    createList(N,1,L),
    findall(RPartial, oneSol(L,RPartial),R).
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să aibă ca rezultat
;    lista iniţială in care atomii de pe nivelul k au fost inlocuiti cu 0
;    (nivelul superficial se consideră 1). Se va folosi o funcţie MAP.
; Exemplu pentru lista (a (1 (2 b)) (c (d)))
; a) k=2 => (a (0 (2 b)) (0 (d))) b) k=1 => (0 (1 (2 b)) (c (d))) c) k=4 =>lista nu se modifică


; replaceElems(l, level, k) =
; = 0, if l is an atom and if level = k
; = l, if l is an atom
; = replaceElems(l1, level + 1, k) U ... U replaceElems(ln, level + 1, k), otherwise (l = l1l2...ln)

(defun replaceElems(l level k)
```

```
  (cond
    ((and (atom l) (equal level k)) 0)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level) k)) l))
  )
)


(defun main(l k)
  (replaceElems l 0 k)
)
```

% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG având modelul de flux (i, o):

```
f([],0).
f([H|T],S):-f(T,S1),S1>=2,!,S is S1+H.
f([_|T],S):-f(T,S1),S is S1+1.
```

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în ambele clauze, fără a redefini
% logica clauzelor. Justificați răspunsul.

```
f1([],0).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
    S1 >= 2,
    !,
    S is S1 + H.
aux(S1,_,S):-
    S is S1 + 1.
```

% B. Să se scrie un program PROLOG care generează lista aranjamentelor de k elemente
%   dintr-o listă de numere întregi, pentru care produsul elementelor e mai mic decât
%   o valoare V dată. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.
%
% Exemplu- pentru lista [1, 2, 3], k=2 și V=7 ⇒ [[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]] (nu neapărat în această ordine)

```
% insert(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

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

```
% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) < v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP < V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)

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

; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....).
;    Se cere să se determine calea de la radăcină către un nod dat.
;    Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) nod=e => (a c d e) b) nod=v => ()
```

```lisp
; myAppend(l1l2...ln, p1p2...pm) =
; = p1p2...pm, if n = 0
; = {l1} U myAppend(l2...ln, p1p2...pm), otherwise

(defun myAppend(l p)
  (cond
    ((null l) p)
    (t (cons (car l) (myAppend (cdr l) p)))
  )
)

; reverseSuperficial(l1l2...ln) =
; = [], if n = 0
; = myAppend(reverseSuperficial(l2...ln), list(l1)), otherwise

(defun reverseSuperficial(l)
  (cond
    ((null l) nil)
    (t (myAppend (reverseSuperficial (cdr l) ) (list (car l))))
  )
)


; linearize(l) =
; = [], if l = []
; = list(l), if l is an atom
; = linearize(l1) U ... U linearize(ln), otherwise (l = l1l2...ln)

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


; path(l, node, collector) =
; = collector, if l is an atom and l equal node
; = nil, if l is an atom
; = path(l1, node, {l1,1} U collector) U ... U path(ln, node, {ln,1} U
collector), otherwise (where l = l1l2...ln)


(defun path (l node collector)
    (cond
        ((and (atom l) (eq l node)) collector)
        ((atom l) nil)
        (t (apply #'linearize (list (mapcar #'(lambda (a) (path a node
(cons (car l) collector))) l))))
    )
)

(defun pathMain (tree node)
    (reverse (path tree node nil))
)
```

; A. Fie următoarea definiţie de funcţie LISP

```
(DEFUN Fct(F L)
  (COND
    ((NULL L) NIL)
    ((FUNCALL F (CAR L)) (CONS (FUNCALL F (CAR L)) (Fct F (CDR L))))
    (T NIL)
  )
)
```

; Rescrieți această definiție pentru a evita dublul apel recursiv (FUNCALL F (CAR L)), fără a redefini logica clauzelor și fără a
; folosi o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(defun f1(f l)
  (cond
    ((null l) nil)
    (t ((lambda (x)
         (cond
           (x (cons x (f1 f (cdr l))))
           (t nil)
         )
        )(FUNCALL f (car l))
       )
    )
  )
)
```

% B. Să se scrie un program PROLOG care generează lista aranjamentelor de k elemente
%   dintr-o listă de numere întregi, având o sumă S dată.
%   Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

% Exemplu- pentru lista [6, 5, 3, 4], k=2 şi S=9⇒ [[6,3],[3,6],[5,4],[4,5]] (nu neapărat în această ordine)

```
% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1
```

```
% arr(L:list, K:number, R:list)
% (i,i,o)

arr([H|_],1,[H]).
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).

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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


% oneSol(L:list, K:number, S:number, R:list)
% (i,i,i,o)

oneSol(L,K,S,R):-
    arr(L,K,R),
    computeSum(R,RS),
    RS =:= S.

allSols(L,K,S,R):-
    findall(RP, oneSol(L,K,S,RP),R).
; C. Un arbore n-ar se reprezintă în LISP astfel (nod subarbore1
subarbore2 .....).
;    Se cere să se determine numărul de noduri de
;    pe nivelul k. Nivelul rădăcinii se consideră 0.
;    Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) k=2 => nr=3 (g d f) b) k=4 => nr=0 ()


; computeSum(l1l2...ln) =
; = 0, if n = 0
; = computeSum(l1) + computeSum(l2...ln), if l1 is a list
; = l1 + computeSum(l2...ln), otherwise

(defun computeSum (l)
  (cond
    ((null l) 0)
    ((listp (car l)) (+ (computeSum (car l)) (computeSum (cdr l))))
    (t (+ (car l) (computeSum (cdr l))))
  )
)
```

```
;  nrNodes(l k level)
; = list(1), if l is an atom and k = level
; = list(0), if l is an atom
; = nrNodes(l1, k, level + 1) + ... + nrNodes(ln, k, level + 1),
otherwise (l = l1l2...ln)

(defun nrNodes(l k level)
  (cond
    ((and (atom l) (eq k level)) 1)
    ((atom l) 0)
    (t (apply #'computeSum (list (mapcar #' (lambda (a) (nrNodes a k (+ 1
level))) l))))
  )
)

(defun main(l k)
  (nrNodes l k -1)
)
% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
având modelul de flux (i, o):

f([],-1).
f([H|T],S):-f(T,S1), S1<1, S is S1-H, !.
f([_|T],S):-f(T,S).

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
ambele clauze, fără a redefini logica clauzelor.
% Justificați răspunsul.

f1([],-1).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
    S1 < 1,
    S is S1 - H.
aux(_,S,S).

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista permutărilor având proprietatea
%   că valoarea absolută a diferenţei dintre două valori consecutive din
permutare este <=3.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.
% Exemplu- pentru lista L=[2,7,5] ⇒ [[2,5,7], [7,5,2]] (nu neapărat în
această ordine)

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E: element, R: result list)
% (i,i,o)


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

```
    insert(E,T,R).


% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1, perm(l2...ln)), otherwise

% perm(L:list, R: result list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a >= b
% = b - a, otherwise

absDiff(A,B,R):-
    A >= B,
    R is A - B.
absDiff(A,B,R):-
    A < B,
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) <= 3
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) <= 3
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R =< 3.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R =< 3,
    checkAbsDiff([H2|T]).


% oneSol(l1l2...ln, r1r2...rm) =
% = perm(l1l2...ln, r1r2...rm), if checkAbsDiff(l1l2...ln) = true

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

%allSols(N:number, R:result list)
% (i,o)

allSols(L,R):-
    findall(RPartial,oneSol(L,RPartial),R).
; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice
pare cu numărul natural succesor.
;    Se va folosi o funcție MAP.
```

```
; Exemplu pentru lista (1 s 4 (2 f (7))) va rezulta (1 s 5 (3 f (7))).


; replaceNumbers(l) =
; = l + 1, if l is a number and l % 2 == 0
; = l, if l is an atom
; = replaceNumbers(l1) U ... U replaceNumbers(ln), otherwise (l =
l1l2...ln)

(defun replaceNumbers(l)
  (cond
    ((and (numberp l) (eq (mod l 2) 0)) (+ 1 l))
    ((atom l) l)
    (t (mapcar #' replaceNumbers l))
  )
)

% A. Fie următoarea definiţie de predicat PROLOG f(list, integer), având
modelul de flux (i, o):

f([], -1):-!.
f([_|T], Y):- f(T,S), S<1, !, Y is S+2.
f([H|T], Y):- f(T,S), S<0, !, Y is S+H.
f([_|T], Y):- f(T,S), Y is S.

%Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
clauze, fără a redefini logica clauzelor. Justificați răspunsul.

f1([], -1):-!.
f1([_|T], Y):-
    f1(T,S),
    aux(S,_,Y).


aux(S,_,Y):-
    S < 1,
    !,
    Y is S + 2.
aux(S,H,Y):-
    S < 0,
    !,
    Y is S + H.
aux(S,_,S).

% B. Dându-se o listă formată din numere întregi, să se genereze în
PROLOG lista submulţimilor
%    cu cel puţin N elemente având suma divizibilă cu 3.
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[2,3,4] și N=1 ⇒ [[3],[2,4],[2,3,4]] (nu
neapărat în această ordine)


% subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
```

```
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 3 == 0
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 3 =:= 0.

% myLength(l1l2...ln) =
% = 0, if n = 0
% = 1 + myLength(l2...ln), otherwise

% myLength(L:list, R:number)
% (i,o)

myLength([],0).
myLength([_|T],R1):-
    myLength(T,R),
    R1 is R + 1.

% checkLength(l1l2...lm, n) =
% = true, if myLength(l1l2...lm) >= n
% = false, otherwise

% checkLength(L:list, N:number)
% (i,i)

checkLength(L,N):-
    myLength(L,RL),
    RL >= N.
```

```
; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;    lista iniţială în care toate apariţiile unui element e au fost
înlocuite cu o valoare e1.
;    Se va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (A)) e este A şi e1 este B =>
(1 (2 B (3 B)) (B))
;         b) dacă lista este (1 (2 (3))) şi e este A => (1 (2 (3)))


; replaceElems(l elem newElem)
; = newElem, if l is an atom and l = elem
; = l, if l is an atom
; = replaceElems(l1, elem, newElem) U ... U replaceElems(ln, elem,
newElem), otherwise

(defun replaceElems(l elem newElem)
  (cond
    ((and (atom l) (eq elem l)) newElem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a elem newElem)) l))
  )
)

% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(1, 1):-!.
f(K,X):-K1 is K-1, f(K1,Y), Y>1, !, K2 is K1-1, X is K2.
f(K,X):-K1 is K-1, f(K1,Y), Y>0.5, !, X is Y.
f(K,X):-K1 is K-1, f(K1,Y), X is Y-1.

% Rescrieți această definiţie pentru a evita apelul recursiv f(J,V) în
clauze,
% fără a redefini logica clauzelor. Justificaţi răspunsul.

f1(1, 1):-!.
f1(K,X):-
    K1 is K-1,
    f1(K1,Y),
    aux(Y,K1,X).

aux(Y,K1,X):-
    Y > 1,
    !,
    X is K1 - 1.
aux(Y,_,Y):-
    Y > 0.5,
    !.
aux(Y,_,X):-
    X is Y - 1.

% B. Să se scrie un program PROLOG care generează lista permutărilor
mulţimii 1..N,
%    cu proprietatea că valoarea absolută a diferenţei între 2 valori
consecutive
%    din permutare este >=2.
```

```
%    Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru N=4 ⇒ [[3,1,4,2], [2,4,1,3]] (nu neapărat în această
ordine)


% createList(n,i) =
% = [], if i = n + 1
% = {i} U createList(n, i + 1), otherwise

% createList(N:number, I:number, R:list)
% (i,i,o)

createList(N,I,[]):-
    I =:= N + 1.
createList(N,I,[I|R]):-
    I1 is I + 1,
    createList(N,I1,R).

% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(elem,l2...ln)

% insert(E:element, L:list, R:list)
% (i,i,o)

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

% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1,perm(l2...ln)), otherwise

% perm(L:list, R:list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a > b
% = b - a, otherwise

% absDiff(A:number, B:number, R:number)
% (i,i,o)

absDiff(A,B,R):-
    A > B,
    !,
    R is A - B.
absDiff(A,B,R):-
    R is B - A.

% checkAbsDiff(l1l2...ln) =
```

```
% = true, if absDiff(l1,l2) >= 2
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) >= 2
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R >= 2.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R >= 2,
    !,
    checkAbsDiff([H2|T]).

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

allSols(N,R):-
    createList(N,1,L),
    findall(RPartial, oneSol(L,RPartial),R).

; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice
care sunt mai mari
;     decât o valoare k dată și sunt situate pe un nivel impar, cu
numărul natural predecesor.
;     Nivelul superficial se consideră 1. Se va folosi o funcție MAP.

; Exemplu pentru lista (1 s 4 (3 f (7))) și
; a) k=0 va rezulta (0 s 3 (3 f (6))) b) k=8 va rezulta (1 s 4 (3 f (7)))


; replaceNumbers(l, k, level) =
; = l - 1, if l is a number and l > k and level % 2 == 1
; = l, if l is an atom
; = replaceNumbers(l1, k, level + 1) U ... U replaceNumbers(ln, k, level
+ 1), otherwise (l = l1l2...ln)

(defun replaceNumbers(l k level)
  (cond
    ((and (and (numberp l) (> l k)) (equal (mod level 2) 1)) (- l 1))
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNumbers a k (+ 1 level))) l))
  )
)

(defun main(l k)
  (replaceNumbers l k 0)
)

% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
f(list, integer),
%    având modelul de flux (i, o):
```

```
f([], 0).
f([H|T],S):-f(T,S1),S1<H,!,S is H.
f([_|T],S):-f(T,S1),S is S1.

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
ambele clauze, fără a redefini logica clauzelor. Justificați
% răspunsul.


f1([],0).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,H):-
    S1 < H.
aux(S1,_,S1).
% B. Dându-se o listă formată din numere întregi, să se genereze lista
submulţimilor
%   cu k elemente numere impare, în progresie aritmetică.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[1,5,2,9,3] şi k=3 ⇒ [[1,5,9],[1,3,5]]
%  (nu neapărat în această ordine)

% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insertFirst(l2...ln, elem)

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).
```

```
% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 ==
% = countEven(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)

countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).

% checkOdd(l1l2...ln, n) =
% = true, if countOdd(l1l2...ln) = n
% = false, otherwise

% checkOdd(L:list, N:number)
% (i)

checkOdd(L, N):-
    countOdd(L,R),
    R =:= N.

% progression(l1l2...ln) =
% = true, if n = 3 and l2 = (l1 + l2)/2
```

```
% = progression(l2...ln), if l2 = (l1 + l2)/2
% = false, otherwise

% progression(L:list)
% (i)

progression([H1,H2,H3]):- H2 =:= (H1 + H3) /2.
progression([H1,H2,H3|T]):-
    H2 =:= (H1 + H3) /2,
    progression([H2,H3|T]).

% oneSol(l1l2...ln, k) =
% = subsets(l1l2...ln), if checkOdd(subsets(l1l2...ln), K) = true and
% progression(subsets(l1l2...ln)) = true

% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    subsets(L,R),
    checkOdd(R,K),
    progression(R).

% allSols(L:list, K:number, R:result list)
% (i,i,o)

allSols(L,K,R):-
    sortare(L,RL),
    findall(RPartial, oneSol(RL,K,RPartial),R).

; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca
;    rezultat lista iniţială din care au fost eliminaţi toţi
;    atomii nenumerici de pe nivelurile pare (nivelul superficial se
consideră 1).
;    Se va folosi o funcție MAP.

; Exemplu pentru lista (a (1 (2 b)) (c (d))) rezultă (a (1 (2 b)) ((d)))


; removeElems(l level)
; = list(l), if l is a number
; = [], if l is an atom and level % 2 == 0
; = list(l), if l is an atom
; = removeElems(l1, level + 1) U ... U removeElems(l2, level + 1),
otherwise

(defun removeElems(l level)
  (cond
    ((numberp l) (list l))
    ((and (atom l) (eq (mod level 2) 0)) nil)
    ((atom l) (list l))
    (t (list (mapcan #' (lambda (a) (removeElems a (+ 1 level))) l)))
  )
)

(defun main(l)
  (car (removeElems l 0))
```

)

```
% A. Fie următoarea definiţie de predicat PROLOG f(integer, integer),
având modelul de flux (i, o):

f(50, 1):-!.
f(I,Y):-J is I+1, f(J,S), S<1, !, K is I-2, Y is K.
f(I,Y):-J is I+1, f(J,Y).

% Rescrieți această definiție pentru a evita apelul recursiv f(J,V) în
ambele clauze,
% fără a redefini logica clauzelor. Justificați răspunsul.

f2(50, 1):-!.
f2(I,Y):-
    J is I + 1,
    f2(J,S),
    aux(I,S,Y).

aux(I,S,Y):-
    S < 1,
    !,
    Y is I - 2.
aux(_,S,S).

% B. Să se scrie un program PROLOG care generează lista submulţimilor cu
N elemente,
%    cu elementele unei liste, astfel încât suma elementelor dintr-o
submulțime să
%    fie număr par. Se vor scrie modelele matematice și modelele de flux
pentru predicatele folosite.
% Exemplu- pentru lista L=[1, 3, 4, 2] şi N=2 ⇒ [[1,3], [2,4]]


% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln

% insert(E:element, L:list, R:list)
% (i,i,o)

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

% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k-1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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(l1l2...ln)
% = true, if n = 2 and l1 < l2
% = checkIncreasing(l2...ln), if l1 < l2
% = false, otherwise

% checkIncreasing(L:list)
% (i)

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

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 == 0
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,RS),
    RS mod 2 =:= 0.


% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    arr(L,K,R),
    checkIncreasing(R),
    checkSum(R).

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


; C. Se consideră o listă neliniară. Să se scrie o funcţie LISP care să
aibă ca rezultat
;  lista iniţială din care au fost eliminați toți atomii numerici
multipli de 3.
;  Se va folosi o funcție MAP.

; Exemplu a) dacă lista este (1 (2 A (3 A)) (6)) => (1 (2 A (A)) NIL)
```

```
;         b) dacă lista este (1 (2 (C))) => (1 (2 (C)))


; removeElem(l, elem) =
; = nil, if l is number and l % 3 == 0
; = list(l), if l is an atom
; = removeElem(l1, elem) U ... U removeElem(ln, elem), otherwise (l =
l1l2...ln)

(defun removeElem(l)
  (cond
    ((and (numberp l) (equal (mod l 3) 0)) nil)
    ((atom l) (list l))
    (t (list (mapcan #'removeElem l)))
  )
)


(defun main(l)
  (car (removeElem l))
)
 ; A. Fie următoarea definiţie de funcţie LISP


(DEFUN Fct(F L)
  (COND
    ((NULL L) NIL)
    ((FUNCALL F (CAR L)) (CONS (FUNCALL F (CAR L)) (Fct F (CDR L))))
    (T NIL)
  )
)

; Rescrieți această definiție pentru a evita dublul apel recursiv
(FUNCALL F (CAR L)), fără a redefini logica clauzelor și fără a
; folosi o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați
răspunsul.

(defun f1(f l)
  (cond
    ((null l) nil)
    (t ((lambda (x)
         (cond
           (x (cons x (fct f (cdr l))))
           (t nil)
         )
       )(funcall f (car l))
      )
    )
  )
)

% B. Să se scrie un program PROLOG care generează lista aranjamentelor de
k elemente dintr-o listă
%    de numere întregi, având produs P dat. Se vor scrie modelele
matematice și modelele de flux
%    pentru predicatele folosite.
```

```
% Exemplu- pentru lista [2, 5, 3, 4, 10], k=2 şi P=20 ⇒
[[2,10],[10,2],[5,4],[4,5]] (nu neapărat în această ordine)


% insert(l1l2...ln, elem) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

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


% arr(l1l2...ln, k) =
% = l1, if k = 1
% = arr(l1l2...ln, k), if k >= 1
% = insert(l1, arr(l2...ln, k - 1)), if k > 1

% arr(L:list, K:number, R:list)
% (i,i,o)

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

% productElems(l1l2...ln) =
% = 1, if n = 0
% = l1 * productElems(l2...ln), otherwise

% productElems(L:list, R:number)
% (i,o)

productElems([],1).
productElems([H|T],R1):-
    productElems(T,R),
    R1 is H*R.

% checkProduct(l1l2...ln, v) =
% = true, if productElems(l1l2...ln) = v
% = false, otherwise

checkProduct(L,V):-
    productElems(L,RP),
    RP = V.

% oneSol(L:list, K:number, V:number, R:list)
% (i,i,o)

oneSol(L,K,V,R):-
    arr(L,K,R),
```

```
    checkProduct(R,V).

% allSols(L:list, K:number, V:number, R:result list)
% (i,i,i,o)

allSols(L,K,V,R):-
    findall(RPartial,oneSol(L,K,V,RPartial),R).
; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....).
;    Se cere să se determine lista nodurilor de pe nivelurile pare din
arbore
;    (în ordinea nivelurilor 0, 2, …). Nivelul rădăcinii se consideră 0.
;    Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e (h))) (f))) => (a g d f h)



; myAppend(l1l2...ln, p1p2...pm) =
; = p1p2...pm, if n = 0
; = {l1} U myAppend(l2...ln, p1p2...pm), otherwise


(defun myAppend(l p)
  (cond
    ((null l) p)
    (t (cons (car l) (myAppend (cdr l) p)))
  )
)

; getNodes(l k level)
; = [l], if l is an atom an level = k
; = [], if l is an atom
; = getNodes(l1,k,level + 1) U ... U getNodes(ln, k, level + 1),
otherwise (l = l1l2....ln)

(defun getNodes(l k level)
  (cond
    ((and (atom l) (eq k level)) (list l))
    ((atom l) nil)
    (t (mapcan #' (lambda (a) (getNodes a k (+ 1 level))) l))
  )
)


; getMain(l1l2...ln, level) =
; = [], if getNodes(l1l2...ln, level, - 1) is []
; = myAppend(getNodes(l1l2...ln, level, -1), getMain(l1l2...ln, level +
2)), otherwise



(defun getMain(l level)
  (cond
    ((null (getNodes l level -1)) nil)
    (t (myAppend (getNodes l level -1) (getMain l (+ 2 level))))
  )
)
```

```
(defun main(l)
  (getMain l 0)
)

% A. Fie L o listă numerică şi următoarea definiţie de predicat PROLOG
având modelul de flux (i, o):

f([],-1).
f([H|T],S):-f(T,S1),S1>0,!,S is S1+H.
f([_|T],S):-f(T,S1),S is S1.

% Rescrieți această definiție pentru a evita apelul recursiv f(T,S) în
ambele clauze, fără a redefini logica clauzelor.
%  Justificați răspunsul.

f1([],-1).
f1([H|T],S):-
    f1(T,S1),
    aux(S1,H,S).

aux(S1,H,S):-
    S1 > 0,
    !,
    S is S1 + H.
aux(S1,_,S1).

% B. Dându-se o listă formată din numere întregi, să se genereze lista
submulţimilor
%   cu k elemente numere impare, în progresie aritmetică.
%   Se vor scrie modelele matematice și modelele de flux pentru
predicatele folosite.

% Exemplu- pentru lista L=[1,5,2,9,3] şi k=3 ⇒ [[1,5,9],[1,3,5]]
%  (nu neapărat în această ordine)

% insertFirst(l1l2...ln, elem) =
% = {elem} U l1l2...ln

% insertFirst(L:list, E:element, R:list)
% (i,i,o)

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


% insert(l1l2...ln, elem) =
% = list(elem) , if n = 0
% = l1l2...ln , if l1 = elem
% = {elem} U l1l2...ln, if elem < l1
% = {l1} U insert(l2...ln, elem)

% insert(L:list, E:element, R:list)
% (i,i,o)

insert([],E,[E]).
insert([H|_],E,[H|_]):-
    H=:=E,
```

```
    !.
insert([H|T],E,R1):-
    E < H,
    !,
    insertFirst([H|T],E,R1).
insert([H|T],E,[H|R]):-
    insert(T,E,R).

% sortare(l1l2...ln) =
% = nil , if n = 0
% = insert(sortare(l2...ln), l1) , otherwise

% sortare(L:list, R:result)
% (i,o)

sortare([],[]).
sortare([H|T],R1):-
    sortare(T,R),
    insert(R,H,R1).

%subsets(l1l2...ln) =
% = [], if n = 0
% = {l1} U subsets(l2...ln), if n >= 1
% = subsets(l2...ln), if n >= 1

% subsets(L:list, R:result list)
% (i,o)

subsets([],[]).
subsets([H|T],[H|R]):-
    subsets(T,R).
subsets([_|T],R):-
    subsets(T,R).


% countOdd(l1l2...ln) =
% = 0, if n = 0
% = 1 + countEven(l2...ln), if l1 % 2 ==
% = countEven(l2...ln), otherwise

% countOdd(L:list, R:number)
% (i,o)

countOdd([],0).
countOdd([H|T],R1):-
    H mod 2 =:= 1,
    !,
    countOdd(T,R),
    R1 is R + 1.
countOdd([_|T],R):-
    countOdd(T,R).

% checkOdd(l1l2...ln, n) =
% = true, if countOdd(l1l2...ln) = n
% = false, otherwise

% checkOdd(L:list, N:number)
% (i)
```

```
checkOdd(L, N):-
    countOdd(L,R),
    R =:= N.

% progression(l1l2...ln) =
% = true, if n = 3 and l2 = (l1 + l2)/2
% = progression(l2...ln), if l2 = (l1 + l2)/2
% = false, otherwise

% progression(L:list)
% (i)

progression([H1,H2,H3]):- H2 =:= (H1 + H3) /2.
progression([H1,H2,H3|T]):-
    H2 =:= (H1 + H3) /2,
    progression([H2,H3|T]).

% oneSol(l1l2...ln, k) =
% = subsets(l1l2...ln), if checkOdd(subsets(l1l2...ln), K) = true and
% progression(subsets(l1l2...ln)) = true

% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    subsets(L,R),
    checkOdd(R,K),
    progression(R).

% allSols(L:list, K:number, R:result list)
% (i,i,o)

allSols(L,K,R):-
    sortare(L,RL),
    findall(RPartial, oneSol(RL,K,RPartial),R).

; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice
impare situate pe un nivel par,
;    cu numărul natural succesor.
;    Nivelul superficial se consideră 1. Se va folosi o funcție MAP.

; Exemplu pentru lista (1 s 4 (3 f (7))) va rezulta (1 s 4 (4 f (7))).


; replaceElems(l level) =
; = l + 1, if l is a number and l % 2 == 1 and level % 2 == 0
; = l, if l is an atom
; = replaceElems(l1, level + 1) U ... U replaceElems(ln, level + 1),
otherwise (l = l1l2...ln)

(defun replaceElems(l level)
  (cond
    ((and (and (numberp l) (eq (mod l 2) 1)) (eq (mod level 2) 0)) (+ 1
l))
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceElems a (+ 1 level))) l))
  )
```

```
)

(defun main(l)
  (replaceElems l 0)
)

; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(L)
  (COND
    ((NULL L) 0)
    ((> (F (CDR L)) 2) (+ (F (CDR L)) (CAR L)))
    (T (+ (F (CDR L)) 1))
  )
)

; Rescrieți această definiție pentru a evita apelul recursiv repetat (F
(CDR L)), fără a redefini logica clauzelor și fără a folosi o
; funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.



(defun f1(l)
  (cond
    ((null l) 0)
    (t ((lambda (x)
          (cond
            ((> x 2) (+ x (car l)))
            (t (+ x 1))
          )
        ) (f1 (cdr l))
       )
    )
  )
)

% B. Să se scrie un program PROLOG care generează lista submulţimilor cu
suma număr impar,
%  cu valori din intervalul [a, b]. Se vor scrie modelele matematice și
modelele de flux pentru
%   predicatele folosite.

% Exemplu- pentru a=2 și b=4 ⇒ [[2,3],[3,4],[2,3,4]] (nu neapărat în
această ordine)


% subset(l1l2...ln) =
% = [], if n = 0
% = {l1} U subset(l2...ln), if n >= 1
% = subset(l2...ln), if n >= 1

% subset(L:list, R:result list)
% (i,o)

subset([],[]).
subset([H|T],[H|R]):-
    subset(T,R).
subset([_|T],R):-
```

```
    subset(T,R).

% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

% checkSum(l1l2...ln) =
% = true, if computeSum(l1l2...ln) % 2 == 1
% = false, otherwise

% checkSum(L:list)
% (i)

checkSum(L):-
    computeSum(L,S),
    S mod 2 =:= 1.


% createList(a, b) =
% = [], if a = b + 1
% = {a} U createList(a + 1, b), otherwise

% createList(A:number, B:number, R:list)
% (i,i,o)

createList(A,B,[]):-
    A =:= B + 1.
createList(A,B,[A|R]):-
    A1 is A + 1,
    createList(A1,B,R).

% oneSol(L:list, R:list)
% (i,o)

oneSol(L,R):-
    subset(L,R),
    checkSum(R).

allSols(A,B,R):-
    createList(A,B,L),
    findall(RPartial,oneSol(L,RPartial),R).

; C. Să se substituie valorile numerice cu o valoare e dată, la orice
nivel al unei liste neliniare.
;    Se va folosi o funcție MAP.
; Exemplu, pentru lista (1 d (2 f (3))), e=0 rezultă lista (0 d (0 f
(0))).
```

```
; replaceNumbers(l, elem) =
; = elem, if l is a number
; = l, if l is an atom
; = replaceNumbers(l1, elem) U ... U replaceNumbers(ln, elem), otherwise
(l = l1l2...ln)


(defun replaceNumbers(l elem)
  (cond
    ((numberp l) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNumbers a elem)) l))
  )
)

; A. Fie următoarea definiţie de funcţie LISP

(DEFUN F(N)
  (COND
    ((= N 1) 1)
    ((> (F (- N 1)) 2) (- N 2))
    ((> (F (- N 1)) 1) (F (- N 1)))
    (T (- (F (- N 1)) 1))
  )
)

; Rescrieți această definiție pentru a evita apelul repetat (F (- N 1)),
fără a redefini logica clauzelor și fără a folosi o funcție
; auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.


(defun f1(n)
  (cond
    ((= n 1) 1)
    (t ((lambda (x)
         (cond
           ((> x 2) (- n 2))
           ((> x 1) x)
           (t (- x 1))
         )
       )(f1 (- n 1))
      )
    )
  )
)

% B. Pentru o valoare N dată, să se genereze lista permutărilor cu
elementele N, N+1,….,2*N-1
%   având proprietatea că valoarea absolută a diferenţei dintre două
valori consecutive
%   din permutare este <=2. Se vor scrie modelele matematice și modelele
de
%   flux pentru predicatele folosite.


% insert(elem, l1l2...ln) =
% = {elem} U l1l2...ln
% = {l1} U insert(l2...ln, elem)
```

```
% insert(L:list, E: element, R: result list)
% (i,i,o)


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


% perm(l1l2...ln) =
% = [], if n = 0
% = insert(l1, perm(l2...ln)), otherwise

% perm(L:list, R: result list)
% (i,o)

perm([],[]).
perm([H|T],R1):-
    perm(T,R),
    insert(H,R,R1).

% absDiff(a,b) =
% = a - b, if a >= b
% = b - a, otherwise

absDiff(A,B,R):-
    A >= B,
    R is A - B.
absDiff(A,B,R):-
    A < B,
    R is B - A.

% checkAbsDiff(l1l2...ln) =
% = true, if n = 2 and absDiff(l1,l2) <= 2
% = checkAbsDiff(l2...ln), if absDiff(l1,l2) <= 2
% = false, otherwise

% checkAbsDiff(L:list)
% (i)

checkAbsDiff([H1,H2]):-
    absDiff(H1,H2,R),
    R =< 2.
checkAbsDiff([H1,H2|T]):-
    absDiff(H1,H2,R),
    R =< 2,
    checkAbsDiff([H2|T]).

% createList(n,m) =
% = [], if n = m + 1
% = {n} U createList(n+1, m), otherwise

% createList(N:number, M:number, R:result list)
% (i,i,o)

createList(N,M,[]):- N =:= M + 1.
createList(N,M,[N|R]):-
```

```
    N1 is N + 1,
    createList(N1,M, R).

% oneSol(l1l2...ln, r1r2...rm) =
% = perm(l1l2...ln, r1r2...rm), if checkAbsDiff(l1l2...ln) = true

oneSol(L,R):-
    perm(L,R),
    checkAbsDiff(R).

%allSols(N:number, R:result list)
% (i,o)

allSols(N,R):-
    M is 2 * N - 1,
    createList(N,M,RL),
    findall(RPartial,oneSol(RL,RPartial),R).
; C. Se dă o listă neliniară și se cere înlocuirea valorilor numerice
care sunt mai mari
;     decât o valoare k dată și sunt situate pe un nivel impar, cu
numărul natural predecesor.
;     Nivelul superficial se consideră 1. Se va folosi o funcție MAP.

; Exemplu pentru lista (1 s 4 (3 f (7))) și
; a) k=0 va rezulta (0 s 3 (3 f (6))) b) k=8 va rezulta (1 s 4 (3 f (7)))


; replaceNumbers(l, k, level) =
; = l - 1, if l is a number and l > k and level % 2 == 1
; = l, if l is an atom
; = replaceNumbers(l1, k, level + 1) U ... U replaceNumbers(ln, k, level
+ 1), otherwise (l = l1l2...ln)

(defun replaceNumbers(l k level)
  (cond
    ((and (and (numberp l) (> l k)) (equal (mod level 2) 1)) (- l 1))
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNumbers a k (+ 1 level))) l))
  )
)

(defun main(l k)
  (replaceNumbers l k 0)
)

; A. Fie următoarea definiţie de funcţie LISP


(DEFUN F(G L)
  (COND
    ((NULL L) NIL)
    ((> (FUNCALL G L) 0) (CONS (FUNCALL G L) (F (CDR L))))
    (T (FUNCALL G L))
  )
)
```

; Rescrieți această definiție pentru a evita apelul repetat (FUNCALL G L), fără a redefini logica clauzelor și fără a folosi o funcție auxiliară. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(defun f1(g l)
  (cond
    ((null l) nil)
    (t ((lambda (x)
         (cond
           (( > x 0) (cons x (f1 (cdr l))))
           (t x)
         )
        )(FUNCALL G L)
       )
    )
  )
)
```

% B. Să se scrie un program PROLOG care generează lista combinărilor de k elemente
%   dintr-o listă de numere întregi, având suma număr par.
%   Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

% Exemplu- pentru lista [6, 5, 3, 4], k=2 ⇒ [[6,4],[5,3]] (nu neapărat în această ordine)

```
% comb(l1l2...ln, k) =
% = l1, if k = 1 and n >= 1
% = comb(l2...ln, k), if k >= 1
% = {l1} U comb(l2...ln, k - 1), if k > 1

% comb(L:list, K:number, R:list)
% (i,i,o)

comb([E|_],1,[E]).
comb([_|T],K,R):-
    comb(T,K,R).
comb([H|T],K,[H|R]):-
    K > 1,
    K1 is K - 1,
    comb(T, K1, R).


% computeSum(l1l2...ln) =
% = 0, if n = 0
% = l1 + computeSum(l2...ln), otherwise

% computeSum(L:list, R:number)
% (i,o)

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

```
% oneSol(L:list, K:number, R:list)
% (i,i,o)

oneSol(L,K,R):-
    comb(L,K,R),
    computeSum(R,S),
    S mod 2 =:= 0.

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

; C. Să se substituie valorile numerice cu o valoare e dată, la orice
nivel
;    al unei liste neliniare. Se va folosi o funcție MAP.

; Exemplu, pentru lista (1 d (2 f (3))), e=0 rezultă lista (0 d (0 f
(0))).


; replaceNumbers(l, elem) =
; = elem, if l is a number
; = l, if l is an atom
; = replaceNumbers(l1, elem) U ... U replaceNumbers(ln, elem), otherwise
(l = l1l2...ln)

(defun replaceNumbers(l elem)
  (cond
    ((numberp l) elem)
    ((atom l) l)
    (t (mapcar #' (lambda (a) (replaceNumbers a elem)) l))
  )
)

; C. Un arbore n-ar se reprezintă în LISP astfel ( nod subarbore1
subarbore2 .....).
;    Se cere să se determine înălțimea unui nod în arbore.
;    Se va folosi o funcție MAP.

; Exemplu pentru arborele (a (b (g)) (c (d (e)) (f)))
; a) nod=e => înălțimea e 0
; b) nod=v => înălțimea e -1
; c) nod=c => înălțimea e 2


; myMax(a b)
; = a, if a >= b
; = b, otherwise

(defun myMax(a b)
  (cond
    ((>= a b) a)
    (t b)
  )
)

; maxList(l1l2...ln) =
; = -1, if n = 0
; = myMax(maxList(l1), maxList(l2...ln)), if l1 is a list
```

```
; = myMax(l1, maxList(l2...ln)), otherwise


(defun maxList(l)
  (cond
    ((null l) -1)
    ((listp (car l)) (myMax (maxList (car l)) (maxList (cdr l))))
    (t (myMax (car l) (maxList (cdr l))))
  )
)


; heightNode(l, elem, found)
; = -1, if l is an atom
; = heightNode(l1, elem, false) U ... U heightNode(ln, elem, false), if l
is a list and found is false and elem != l1 (l = l1l2...ln)
; = 1 + maxList(heightNode(l1, elem, true) U ... U heightNode(ln, elem,
true)), if l is a list and found is false and elem == l1
; = 1 + maxList(heightNode(l1, elem, true) U ... U heightNode(ln, elem,
true)), otherwise



(defun heightNode(l elem found)
  (cond
    ((atom l) -1)
    ((and (listp l) (equal found NIL) (not (eq (car l) elem))) (apply
#'maxList (list (mapcar #' (lambda(x) (heightNode x elem NIL)) l))))
    ((and (listp l) (equal found NIL) (eq (car l) elem)) (+ 1 (apply
#'maxList (list (mapcar #' (lambda (x) (heightNode x elem T)) l)))))
    (t (+ 1 (apply #'maxList (list (mapcar #' (lambda (x) (heightNode x
elem T)) l)))))
  )
)


(defun main(l elem)
  (heightNode l elem NIL)
)
```