Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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)**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

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

Folosind o functie lambda ce are ca si parametru un l, putem evita apelul repetat al functiei (F (CAR L1) L2) pentru ca acesta o sa se efectueze o singura data, in momentul in care este rulata functia lambda.

C. 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)

SAU VEZI FISIER 1C.txt

```
%aranjamente_sp(LI:lista,S:intreg,LE:intreg,C:lista,O:lista)
%LI - lista din care alegem elementele
%S - suma curenta a elementelor din C
%LE - lungimea lui C
%C - colectoare
%O - un aranjament al listei date ce are suma impara si lungimea para
%model flux: (i,i,i,i,i) - determinist, (i,i,i,i,o) - nedeterminist
%vom folosi varianta nedeterminsita (i,i,i,i,o).
aranjamente_sp(_,S,LE,C,O):- S mod 2 =:= 1,
    LE mod 2 =:= 0,
    O=C.

aranjamente_sp(LI,S,LE,C,O):- extragere(LI,EL),
    elimina_prim(LI,EL,AUX),
    S1 is S + EL,
    LE1 is LE + 1,
    aranjamente_sp(AUX,S1,LE1,[EL|C],O).

%extragere(L:lista,E:element)
%L - lista din care extragem un elemnet
%E - un element din lista
%model de flux (i,i) - determinist, (i,o) - nedeterminist
%vom folosi modelul (i,o) nedeterminist

extragere([H|_],H).
extragere([_|T],O):-extragere(T,O).



%elimina_prim(L:lista,E:element,O:lista)
%elimina prima aparitie a lui E din lista L
%L - lista pe care operam
%E - elementul ce-l stergem
%O - lista rezultata in urma elimintarii
%model de flux (i,i,i) - determinist; (i,i,o) - determinist
%vom folosi modelul (i,i,o)
elimina_prim([],_,[]):-!.
elimina_prim([H|T],E,[H|O]):-H\=E,
    !,
    elimina_prim(T,E,O).
elimina_prim([_|T],_,T).
```

D. 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)))

```
; (load "./pregatire/1D.lisp")
; model matematic
; inlocuire(l, e, subs) = {    l                          , l e atom si l != e
;                         {    subs                       , l e atom si l = e
;                         { inlocuire(l1) U ... U inlocuire(l1) , altfel
;
; inlocuire(l:list, e:element, subs:element)
(defun inlocuire(l e subs)

   (cond

      (
         (AND (atom l) (not(equal l e)))

            l
      )

      (
         (AND (atom l) (equal l e))

            subs
      )

      (t
         (mapcar #'(lambda (x) (

                  inlocuire x e subs
               )
               )
               l
         )
      )
   )
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
4. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

```
f([],-1).

f([H|T],S):-f(T,S1),
   p_aux([H|T],S,S1).

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

p_aux(_,S,S1):-S is S1.
```

Definim un predicat auxiliar ce are aceeasi parametrii ca functia f plus un parametru ce reprezinta valoarea predicatului f(T,S1).

B. Dându-se o listă neliniară conținând atât atomi numerici, cât și nenumerici, se cere un program LISP care să construiască o listă liniară formată doar din acei atomi nenumerici care apar de un număr par de ori în lista inițială. Rezultatul va conține fiecare element o singură dată, în ordine inversă față de ordinea în care elementele apar în lista inițială. **De exemplu**, pentru lista (F A 2 3 (B 1 (A D 5) C C (F)) 8 11 D (A F) F), rezultatul va fi (C D F). NU se pot folosi funcțiile predefinite *reverse* sau *member* din Lisp.

Nu se mai da

de la punctul C - continuare

```
% main(n) = U(permConditie(genereazaLista(1,n)))
%
% main(N:intreg, O:List)
% N - elementul maxim din lista noastra
% O - lista rezultat
% Model de flux (i, o) - determinist - folosim
%              (i, i) - determinist
main(N, O):-
        genereazaLista(1, N, L),
        findall(O1, permConditie(L, O1), O).
```

C. 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)

```
% genereazaLista(i, n) = { [n]                        , i = n
%                        { i + genereazaLista(i+1, n), altfel (i < n)
%
% genereazaLista(i:intreg, n:intreg, r:list)
% i - numarul pe care l adaugam in lista
% n - val maxima din lista
% r - lista rezultat
%
% model de flux (i, i, o) - determinist - cel folosit de noi
%              (i, i, i) - determinist
genereazaLista(N, N, [N]):-!.
genereazaLista(I, N, [I|R]):-
                        I < N,
                        I1 is I + 1,
                        genereazaLista(I1, N, R).

% insereaza(e, l1,...,ln) = 1. e + l1l2...ln
%                           2. l1 + insereaza(e, l2...ln)
%
% insereaza(E: element, L:List, LRez:list)
% E - elementul pe care dorim sa-l inseram pe toate pozitiile
% L - lista in care o sa fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                        insereaza(E, T, Rez).

% perm(l1,...,ln) = 1.      []                          daca n = 0
%                   2.      insereaza(l1, permutari(l2,...,ln))
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
                    permutari(T, L),
                    insereaza(E, L, P).

% conditie(l1,...,ln) = {   adevarat       , n < 2
%                       {   fals           , n >= 2 si abs(l1-l2)<2
%                       {   conditie(l2,...,ln), n >= 2 si abs(l1-l2)>=2
% conditie(L:list)
% L - lista pe care o verificam daca respecta conditia din enunt sau nu
% model de flux (i) - determinist
conditie([_]):-!.
conditie([L1,L2|T]):-
                    D is abs(L1-L2),
                    D >= 2,
                    conditie([L2|T]).

% permConditie(l, k) = 1. permutari(l), conditie(permutari(l)) = adev
% permConditie(L: List, O:List).
% L - lista pe care trebuie sa facem permutarile cu conditie
% O - permutarea ce respecta conditia
% Model de flux: (i, i) - determinist,
%                (i, o) - nedeterminist - cel folosit
permConditie(L, O):-
                permutari(L, O),
                conditie(O).
```

D. 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))).

```
; substituie(l, e, e1, niv) = {   l                                                    , l e atom si l != e
;                            {   e                                                    , l e atom si l = e si niv % 2 = 0
;                            {   e1                                                   , l e atom si l = e si niv % 2 = 1
;                            {   substituie(l1, e, e1, niv + 1) U ... U substituie(ln, e, e1, niv + 1) , altfel
;
; substituie(l:list, e:element, e1:element, niv:intreg)

(defun substituie(l e e1 niv)

   (cond

      (
         (AND (atom l) (not (equal l e)))

            l
      )

      (
         (AND (atom l)  (equal l e) (equal (mod niv 2) 0))

            e
      )

      (
         (AND (atom l)  (equal l e) (equal (mod niv 2) 1))

            e1
      )

      (
         T
            (mapcar #'(lambda (x)
                              (substituie x e e1 (+ niv 1))
                     )
                     l
            )
      )

   )
)

; main(l, e, e1) = substituie(l,e,e1,0)
; aceasta este functia main
; main(l:list, e:element, e1:element)

(defun main(l e e1)

   (substituie l e e1 0)

)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

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

Am definit un nou predicat pentru a evita apelul repetat al functiei f(J, V).

D. 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

```
; sau(l1...,ln) = {    fals                  , n = 0
;                 {    l1 OR sau(l2,...,ln)  , altfel
;
; sau(l:list)
; l - o sa contina doar T sau NIL

(defun sau(l)

   (cond

      (
         (null l)
            NIL
      )

      (
         T
            (OR (CAR l) (sau (cdr l)))
      )
   )
)

; nivel(l, niv, e) = {        adevarat                                  , l e atom si l = e si niv % 2 = 0
;                    {        fals                                      , l e atom si l = e si niv % 2 = 1
;                    {        fals                                      , l e atom si l != e
;                    { nivel(l1,niv+1,e) OR nivel (l2,niv+1,e) OR ... OR nivel(ln,niv+1,e)  , altfel
;
; nivel(l:list, niv:intreg, e:element)

(defun nivel(l niv e)

   (cond

      (
         (AND (atom l) (equal l e) (equal (mod niv 2) 0))

            T
      )

      (
         (AND (atom l) (equal l e) (equal (mod niv 2) 1))

            NIL
      )

      (
         (AND (atom l) (not (equal l e)))

            NIL
      )

      (
         T
            (FUNCALL #'sau(mapcar #'(lambda (l)
                                       (nivel l (+ niv 1) e)
                                    )
                              l
                             )
            )

      )
   )
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN F(N)
        (COND
                ((= N 0) 0)
                (T ((lambda (X)
                        (cond
                                ((> X 1) (- N 2))
                                (T (+ X 1))
                        )
                 ) (F (- N 1)))
                )
        )
)
```

Am folosit o functie lambda

B. Dându-se o listă formată din numere întregi și subliste de numere întregi, se cere un program SWI-Prolog care verifică dacă toate elementele listei (inclusiv și cele din subliste) formează o secvență simetrică. De exemplu, pentru lista [1, 5, [2,4], 7, 11, 25, [11, 7, 4], 2, 5, 1] rezultatul va fi **true**.

Nu se mai cere AICI INCEPE PUNCTUL C

```
% prodLista(l1 ... ln) = { 1, n = 0
%                        { l1 * prodLista(l2 ... ln), altfel

% prodLista(L: List, P: Integer).
% L - lista pentru care trebuie calculata suma
% P - produsul listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
prodLista([], 1).
prodLista([H|T], P):-
 prodLista(T, Rest),
 P is H * Rest.

% conditie(l, v) = { fals, prodLista(L) >= v
%  { adevarat, altfel (prodLista(L) < v)

% conditie(L: list, V:integer).
% L - lista pentru care trebuie verificata conditia
% V - valoarea cu care comparam produsul listei L
% Model de flux: (i, i) - determinist.
conditie(L, V):-
 prodLista(L, P),
 P < V.

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

C. 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)

```
% combinariConditie(l, k) = 1. comb(l, k),
%         daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i, i) - determinist, (i, i, o, i) -
% nedeterminist Folosim (i, i, o, i) - nedeterminist
combinariConditie(L, K, C, V):-
 comb(L, K, C),
 conditie(C, V).

% insereaza(e, l1,...,ln) = 1. e + l1l2...ln
%                           2. l1 + insereaza(e, l2...ln)
%
% insereaza(E: element, L:List, LRez:list)
% E - elementul pe care dorim sa-l inseram pe toate pozitiile
% L - lista in care o sa fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                           insereaza(E, T, Rez).

% perm(l1,...,ln) = 1.      []                          daca n = 0
%                   2.      insereaza(l1, permutari(l2,...,ln))
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
                   permutari(T, L),
                   insereaza(E, L, P).

% aranjamente(l1,...,ln, k) = 1.   permutari(combinari(L, K))
%
% aranjamente(L:list, K:element, O:list)
% L - multimea numerelor
% K - nr de elemente din aranjament
% O - aranjamentul curent
% model de flux (i, i, o, i) - nedeterminist
%              (i, i, i, i) - determinist
aranjamente(L, K, O, V):-
 combinariConditie(L, K, O1, V),
 permutari(O1, O).

% main(L, K) = U(aranjamente(L, K))
%
% main(L:list, K:integer, O:list)
% L - lista de elemente
% K - nr de elemente din aranjament
% O - lista rezultat
%
% model de flux (i, i, i, o) - determinist - pe asta l folosim
%              (i, i, i, i) - determinist

main(L, K, V, O):-
            findall(O1, aranjamente(L, K, O1, V), O).
```

D. 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 => ()

```
; cauta(l1,...,ln, e) = {      fals                          , n = 0
;                       {      adevarat                      , l1 e atom si l1 = e
;                       {      cauta(l1, e) sau cauta(l2,...,ln, e)  , l1 e lista
;                       {      cauta(l2,...,ln, e)           , altfel
;
; cauta(l:list, e:element)

(defun cauta(l e)

   (cond

      (
         (null l)
            NIL
      )

      (
         (AND (atom (car l)) (equal (car l) e))
            T
      )

      (
         (listp (car l))
            (OR (cauta (car l) e) (cauta (cdr l) e))
      )

      (T
         (cauta (cdr l) e)
      )
   )
)

; drum(l, e) = {          []                          ,   l atom si l != e
;              {          e                           ,   l atom si l = e
;              {          l1 + (drum(l2,e) U ... U (drum(ln,e))) ,   l e lista cauta(l, e) = adevarat
;              {          [] + (drum(l2,e) U ... U (drum(ln,e))) ,   altfel (cauta(l1...,ln, e) = false)
;
; drum(l:list, e:element)

(defun drum(l e)

   (cond

      (
         (AND (atom l) (not (equal l e)))
            NIL
      )

      (
         (AND (atom l) (equal l e))
            e
      )

      (
         (AND (listp l) (cauta l e))

            (append (list(car l)) (mapcan #'(lambda (x)
                                             (drum x e)
                                          )
                                          l)))
                  )
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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

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

Noi credem ca in loc de Rez era Y

Cred ca e ceva gresit in enunt

Rescrieți această definiție pentru a evita apelul recursiv **f(T,S)** în clauze. Nu redefiniți predicatul. Justificați răspunsul.

```
f([], -1):-!.
f([H|T], Y):-
        f(T,S),
        aux([H|T], Y, S).

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

Definim un predicat auxiliar pentru a evita apelul repetat.

B. Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care numără pentru câte subliste (considerând și lista inițială) numărul total de atomi numerici pe nivelurile impare este egal cu numărul total de atomi nenumerici pe nivelurile impare. Nivelul superficial este impar. De exemplu, pentru lista (A B 12 (5 D (A F (B) D (5 F) 1) 5) C 9 (F 4 (D) 9 (F (H 7) K) (P 4)) X) rezultatul va fi 4 (listele numărate fiind (5 F) (H 7) (P 4) (5 D (A F (B) D (5 F) 1) 5)).

Nu se mai da

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                        { l1 + sumaLista(l2 ... ln), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
 sumaLista(T, Rest),
 S is H + Rest.

% conditie(l) = { fals, sumaLista(L) % 2 = 0
%  { adevarat, altfel (sumaLista(L) % 2 = 1)

% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
 sumaLista(L, S),
 S mod 2 =:= 1,
      lungime(L, Lung),
      Lung > 0,
      Lung mod 2 =:= 0.

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%          daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C),
 conditie(C).
```

PB C

C. 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]]

```
% lungime(l) = { 0, daca vida(l)
% { (1 - l1 % 2) + lungime(l2 ... ln), altfel
% lungime(L: list, Len:integer)
% L - lista pentru care trebuie aflata lungimea
% Len - lungimea listei date
%
% Model de flux: (i, i) - determinist, (i, o) - determinist
% Folosim (i, o) - determinist.
lungime([], 0).
lungime([H|T], Len):-
lungime(T, Len1),
Ct is H mod 2,
Ctt is 1 - Ct,
Ct1 is Ctt,
Len is Len1 + Ct1.
% lung_lista(l1,...,ln) = { 0, n = 0
% { 1 + lung_lista(l2,..,ln), altfel
% lung_lista(l:list, lung:integer)
% l - lista careia ii determinam lungimea
% model de flux - (i, o) - determinist - il folosim
% - (i, i) - determinist
lung_lista([], 0).
lung_lista([_|T], Lung):-
lung_lista(T, Lung1),
Lung is Lung1 + 1.
% toateSubm(L, K, Lung) = { [] , K > Lung
% { U combinariConditie(L,K) + toateSubm(L, K+1,
% Lung)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
K > Lung,
!.
toateSubm(L, K, Lung, [R|O]):-
findall(O1,combinariConditie(L, K, O1),R),
K1 is K + 1,
toateSubm(L, K1, Lung, O).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
       !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).

% main(l) = { toateSubm(sortare(l), 2, lung_lista(l))
%
% Model de flux (i, i) - determinist, (i, o) - determinist
% Vom folosi (i, o) - determinist
main(L, LC):-
lung_lista(L, Len),
sortare(L, L1),
toateSubm(L1, 2, Len, LC).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN F(L)
        (COND
                ((NULL L) 0)
                (T((lambda(X)
                        (cond
                                ((> (F X) 2) (+ (F (CDR L)) X ))
                                (T (+ X 1))
                        )
                  ) (F(CAR L)))
                 )
        )
)
```

Am folosit o lambda

Nu se da

B. Dându-se o listă eterogenă formată din numere și liste liniare nevide de numere, se cere un program SWI-PROLOG care inversează acele subliste în care cel mai mic multiplu comun al elementelor este mai mare decât pătratul primului element al sublistei. **De exemplu**, pentru lista [[4, 1, 18], 7, 2, -3, [6, 9, 11, 3], 4, [9, 4, 3]], rezultatul corect este: [[18, 1, 4], 7, 2, -3, [3, 11, 9, 6], 4, [9, 4, 3]].

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                         { l1 + sumaLista(l2 ... ln), altfel
% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
                    sumaLista(T, Rest),
                    S is H + Rest.
% conditie(l) = { fals, sumaLista(L) % 2 = 0
%               { adevarat, altfel (sumaLista(L) % 2 = 1)
% conditie(L: list, Suma: integer).
% L - lista pentru care trebuie verificata conditia
% Suma - suma cu care comparam suma listei
% Model de flux: (i,i) - determinist.
conditie(L, Suma):-
                   sumaLista(L, S),
                   S =:= Suma,
                   lungime(L, Lung),
                   Lung mod 2 =:= 0.

% comb(l1,...,ln, k) = 1. [l1] daca k = 1
%                      2. comb(l2,...,ln, k)
%                      3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
                   comb(T, K, C).
comb([H|T], K, [H|C]):-
                      K > 1,
                      K1 is K - 1,
                      comb(T, K1, C).
% combinariConditie(l, k, S) = 1. comb(l, k),daca conditie(comb(l, k),S)
%                                                  - adev
%
% combinariConditie(L: List, K: Integer, C: List, S:integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% C - output value
% S - suma pe care o verificam
% Model de flux: (i, i, i, i) - determinist, (i, i, o, i) -
% nedeterminist Folosim (i, i, o, i) - nedeterminist
combinariConditie(L, K, C, S):-
comb(L, K, C),
conditie(C, S).
```

C. 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]].

```
% lungime(l) = { 0, daca vida(l)
% { (1 - l1 % 2) + lungime(l2 ... ln), altfel
% lungime(L: list, Len:integer)
% L - lista pentru care trebuie aflata lungimea
% Len - lungimea listei date
%
% Model de flux: (i, i) - determinist, (i, o) - determinist
% Folosim (i, o) - determinist.
lungime([], 0).
lungime([H|T], Len):-
lungime(T, Len1),
Ct is H mod 2,
Ctt is 1 - Ct,
Ct1 is Ctt,
Len is Len1 + Ct1.
% lung_lista(l1,...,ln) = { 0, n = 0
% { 1 + lung_lista(l2,..,ln), altfel
% lung_lista(l:list, lung:integer)
% l - lista careia ii determinam lungimea
% model de flux - (i, o) - determinist - il folosim
% - (i, i) - determinist
lung_lista([], 0).
lung_lista([_|T], Lung):-
lung_lista(T, Lung1),
Lung is Lung1 + 1.
% toateSubm(L, K, Lung, S) = { [] , K > Lung
% { U combinariConditie(L,K) +toateSubm(L,K+1,Lung,S)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o,i) - determinist - il folosim pe acesta
% (i,i,i,i,i) - determinist
toateSubm(_, K, Lung, [],_):-
K > Lung,
!.
toateSubm(L, K, Lung, [R|O], S):-
findall(O1,combinariConditie(L, K, O1,S),R),
K1 is K + 1,
toateSubm(L, K1, Lung, O, S).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
       !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(l) = { toateSubm(l, 2, lung_lista(l), S)
%
% Model de flux (i, i, o) - determinist, (i, i, i) - determinist
% Vom folosi (i, i, o) - determinist
main(L, S, LC):-
lung_lista(L, Len),
                sortare(L, L1),
toateSubm(L1, 2, Len, LC, S).
```

D. Se consideră o listă neliniară. Să se scrie o funcţie care să aibă ca rezultat lista iniţială in care atomii de pe nivelul **k** 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)))

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

```
; substituie(l, niv, k) = {   l                                          , l e atom si niv != k
;                        {   0                                        , l e atom si niv = k
;                        {   substituie(l2, niv k) U ... U substituie(ln, niv k) , altfel
;
; substituie(l:list, niv:intreg, k:intreg)

(defun substituie(l niv k)

   (cond

      (
         (AND (atom l) (not (equal niv k)))

            l
      )

      (
         (AND (atom l) (equal niv k))

            0
      )

      (
         T
            (mapcar #'(lambda (x)
                              (substituie x (+ niv 1) k)
                     )
                     l
            )
      )

   )
)

; main(l, k) = substituie(l,0,k)
; aceasta este functia main
; main(l:list, k:integer)

(defun main(l k)

   (substituie l 0 k)

)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

```
f([], 0).

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

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

Am definit un predicat auxiliar pentru a evita apelul repetat al functiei f(T,S).

Nu se mai da

B. Dându-se o listă neliniară conținând atât atomi numerici, cât și nenumerici, se cere un program LISP care să calculeze cel mai mare divizor comun al numerelor impare de la nivelurile pare ale listei. Nivelul superficial al listei se consideră 1. **De exemplu**, pentru lista (A B 12 (9 D (A F (75 B) D (45 F) 1) 15) C 9), rezultatul va fi 3. Se presupune că există cel puțin un număr impar la un nivel par al listei. Nu se va folosi funcția predefinită *gcd* din Lisp.

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                       { l1 + sumaLista(l2 ... ln), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
 sumaLista(T, Rest),
 S is H + Rest.

% conditie(l) = { fals, sumaLista(L) % 2 = 0
%  { adevarat, altfel (sumaLista(L) % 2 = 1)

% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
 sumaLista(L, S),
 S mod 2 =:= 1.

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%          daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C),
 conditie(C).
```

C. Să se scrie un program PROLOG care generează lista submulțimilor 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)

```
% genereazaLista(a, b) = { []      , a > b
%                        { a + genereazaLista(a+1,b), altfel
%
% genereazaLista(a:integer,b:integer,o:list)
% a - capatul inferior al intervalului
% b - capatul superior al intervalului
% o - intervalul [a,b]
genereazaLista(A, B, []):-
                        A > B,
                        !.
genereazaLista(A, B, [A|O]):-
                          A1 is A + 1,
                          genereazaLista(A1, B, O).

% toateSubm(L, K, Lung) = { [] , K > Lung
%              { U combinariConditie(L,K) +toateSubm(L,K+1,Lung)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
                          K > Lung,
                          !.
toateSubm(L, K, Lung, [R|O]):-
                           findall(O1,combinariConditie(L, K, O1),R),
                           K1 is K + 1,
                           toateSubm(L, K1, Lung, O).

% main(l) = { toateAranjamentele(l, 2, lungime(l))
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(A, B, LC):-
       genereazaLista(A, B, L),
 Len is B - A + 1,
 toateSubm(L, 2, Len, LC).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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.
```

20:40

Rescrieți această definiție pentru a evita apelul recursiv **f(J,V)** în clauze. Nu redefiniți predicatul. Justificați răspunsul.

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

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

Am definit un predicat auxiliar pentru a evita apelul repetat al functiei f(K1, Y).

Nu se mai da

B. Dându-se o listă neliniară conținând atât atomi numerici, cât și nenumerici, se cere un program LISP care să înlocuiască fiecare atom nenumeric cu numărul de apariții ale atomului la nivelul pe care se află. **De exemplu**, pentru lista (F A 12 13 (B 11 (A D 15) C C (F)) 18 11 D (A F) F), rezultatul va fi (2 1 12 13 (1 11 (1 1 15) 2 2 (1)) 18 11 1 (1 1) 2).

```
% verifProp(l1,..,.ln) = { adevarat, n = 1
%                       { fals    , n >= 2 si abs(l1-l2) % 2 = 1
%                       { verifProp(l2,...,ln), altfel
% verifProp(l:list)
% l - lista pe care o testam daca respecta sau nu o anumita proprietate
% model de flux - (i) - determinist

verifProp([_]):-!.
verifProp([L1,L2|T]):-
                DIF is abs(L1-L2),
                R is DIF mod 2,
                R =:= 0,
                verifProp([L2|T]).

% conditie(l) = verifProp(L)

% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
        verifProp(L).

% comb(l1,...,ln, k) =     1. [l1]                  daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%         daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C),
 conditie(C).
```

C. 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)

```
% genereazaLista(a, b) = { []      , a > b
%                       { a + genereazaLista(a+1,b), altfel
%
% genereazaLista(a:integer,b:integer,o:list)
% a - capatul inferior al intervalului
% b - capatul superior al intervalului
% o - intervalul [a,b]
genereazaLista(A, B, []):-
                        A > B,
                        !.
genereazaLista(A, B, [A|O]):-
                          A1 is A + 1,
                          genereazaLista(A1, B, O).

% main(l) = { toateAranjamentele(l, 2, lungime(l))
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(N, K, O):-
      genereazaLista(1, N, L),
      findall(O1, combinariConditie(L, K, O1), O).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN Fct(F L)
        (COND
                ((NULL L) NIL)
                (T((lambda (X)
                        (cond
                                (X (CONS X (Fct F (CDR L))))
                                (T NIL)
                        )
                  )(FUNCALL F(CAR L)))
                 )
        )
)
```

D. 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))).

```
; inlocuire(l, e) =  {      e                                          , l e atom si numar
;                    {     l                                        , l e atom, dar nu e numar
;                    { inlocuire(l1,e) U inlocuire(l2,e) U ... U inlocuire(ln,e) , altfel
;
; inlocuire(l:list, e:element)

(defun inlocuire(l e)

   (cond

      (
         (AND (atom l) (numberp l))

            e
      )

      (
         (AND (atom l) (not (numberp l)))

            l
      )

      (T
         (mapcar #'(lambda (x) (

                  inlocuire x e
                )
                )
                l
         )
      )
   )
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

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

Am folosit o lambda

Nu se da

B. Dându-se o listă care reprezintă o mulțime, se cere un program SWI-Prolog, care returnează toate posibilitățile de a împărți mulțimea în *k* submulțimi. Cele *k* submulțimi trebuie să fie disjuncte și fiecare element din mulțimea inițială trebuie să apară într-una dintre submulțimi. De exemplu, pentru mulțimea [1,2,3] și k = 2, soluția este (nu neapărat în această ordine): [[[3, 2], [1]], [[2], [3, 1]], [[3], [2,1]]].

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                        { l1 + sumaLista(l2 ... ln), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
 sumaLista(T, Rest),
 S is H + Rest.

% nrImpare(l1,...,ln) = { 0                        , n = 0
%                       { 1 + nrImpare(l2,...,ln)  , n > 0 si l1 % 2 = 1
%                       { nrImpare(l2,...,ln)      , altfel
% nrImpare(l:list, ct:integer)
% l - lista careia o sa i numaram numarul de numere impare
% ct - numarul de numere impare
% model de flux - (i, o) - determinist - il folosim
%               - (i, i) - determinist
nrImpare([], 0):-!.
nrImpare([H|T], O):-
                nrImpare(T, O1),
                R is H mod 2,
                R =:= 1,
                O is O1 + 1.
nrImpare([_|T], O):-
                nrImpare(T, O).

% conditie(l) = { fals, sumaLista(L) % 2 = 0
%  { adevarat, altfel (sumaLista(L) % 2 = 1)

% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
 sumaLista(L, S),
 S mod 2 =:= 1,
       nrImpare(L, CT),
       CT mod 2 =:= 1.

% comb(l1,...,ln, k) =     1. [l1]                  daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
```

C. 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)

```
comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%          daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C),
 conditie(C).

% toateSubm(L, K, Lung) = { [] , K > Lung
% { U combinariConditie(L,K) +toateSubm(L,K+1,Lung)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
                        K > Lung,
                        !.
toateSubm(L, K, Lung, [R|O]):-
                          findall(O1,combinariConditie(L, K, O1),R),
                          K1 is K + 1,
                          toateSubm(L, K1, Lung, O).

% lungime(l) = { 0, daca vida(l)
%        { 1 + lungime(l2 ... ln), altfel
%
% lungime(L: list, Len)
% L - lista pentru care trebuie aflata lungimea
% Len - lungimea listei date
%
% Model de flux: (i, i) - determinist, (i, o) - determinist
% Folosim (i, o) - determinist.

lungime([], 0).
lungime([_|T], Len):-
 lungime(T, Len1),
 Len is Len1 + 1.

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                          { el (+) l1 ... ln   , n > 0 si el <= l1
%                          { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
       !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(l) = { toateAranjamentele(l, 2, lungime(l))
%
% Model de flux (i, i) - determinist, (i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(L, LC):-
 lungime(L, Len),
 sortare(L, L1),
 toateSubm(L1, 2, Len, LC).
```

D. 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)))

```
; elimina(l, niv) = {        l                                    , l nu e atom numeric si niv % 2 = 1
;                   {        []                                   , l nu e atom numeric si niv % 2 = 0
;                   {        l                                    , l e atom numeric
;                   { elimina(l1, niv) U elimina(l2,niv) U ... U elimina(ln,niv)  , altfel
; elimina(l:list, niv:intreg)

(defun elimina(l niv)

   (cond

      (
         (AND (atom l) (not (numberp l)) (equal (mod niv 2) 1))
            (list l)
      )

      (
         (AND (atom l) (not (numberp l)) (equal (mod niv 2) 0))
            NIL
      )

      (
         (AND (atom l) (numberp l))
            (list l)
      )

      (T
         (list (mapcan #'(lambda (x)
                           (
                             elimina x (+ niv 1)
                           )
                     )
                     l
         )
         )
      )
   )
)

; main(l) = elimina(l, 0)
; l - list
(defun main(l)

   (car (elimina l 0))
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

```
f([],0).

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

aux([H|T], S, S1):-
                S1>=2,
                !,
                S is S1 + H.

aux([_|T], S, S1):-
                S is S1+1.
```

Am definit un predicat auxiliar pentur a evita apelul repetat al functiei f(T, S).

C. 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)

```
% comb(l1,...,ln, k) = 1. [l1] daca k = 1
% 2. comb(l2,...,ln, k)
% 3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
comb(T, K, C).
comb([H|T], K, [H|C]):-
K > 1,
K1 is K - 1,
comb(T, K1, C).
% combinariConditie(l, k, S) = 1. comb(l, k),daca conditie(comb(l, k),S)
% - adev
%
% combinariConditie(L: List, K: Integer, C: List, S:integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% C - output value
% S - suma pe care o verificam
% Model de flux: (i, i, i) - determinist, (i, i, o) -
% nedeterminist Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
comb(L, K, C).
% lung_lista(l1,...,ln) = { 0, n = 0
% { 1 + lung_lista(l2,..,ln), altfel
% lung_lista(l:list, lung:integer)
% l - lista careia ii determinam lungimea
% model de flux - (i, o) - determinist - il folosim
% - (i, i) - determinist
lung_lista([], 0).
lung_lista([_|T], Lung):-
lung_lista(T, Lung1),
Lung is Lung1 + 1.
% toateSubm(L, K, Lung) = { [] , K > Lung
% { U combinariConditie(L,K) +toateSubm(L,K+1,Lung)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
K > Lung,
!.
toateSubm(L, K, Lung, [R|O]):-
findall(O1,combinariConditie(L, K, O1),R),
K1 is K + 2,
toateSubm(L, K1, Lung, O).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
    !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(l) = { toateSubm(l, 0, lung_lista(l))
%
% Model de flux (i, o) - determinist, (i, i) - determinist
% Vom folosi (i, o) - determinist
main(L, LC):-
lung_lista(L, Len),
   sortare(L, L1),
toateSubm(L1, 0, Len, LC).
```

D. 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

```
;inaltime_nod(a,e,gasit) = {          -1          , a atom
;                          {       max(inaltime_nod(a1,e,fals),...,inaltime_nod(an,e,fals))    , a lista si gasit=fals si a1 !=e
;                          {       1+max(inaltime_nod(a1,e,true),...,inaltime_nod(an,e,true))  , a lista si gasit=fals si a1=e
;                          {       1+max(inaltime_nod(a1,e,true),...,inaltime_nod(an,e,true))  , altfel (a lista si gasit=adevarat)
; inaltime_nod(a:lista,e:element,gasit:T/NIL)

(defun inaltime_nod(a e gasit)

   (cond

      (
         (atom a)
            -1
      )

      (
         (AND (listp a) (equal gasit NIL) (not (equal e (car a))))
            (apply #'max(
                     mapcar #'(
                              lambda (x)
                                    (
                                       inaltime_nod x e NIL
                                    )
                              )
                              a
                     )
            )
      )

      (
         (AND (listp a) (equal gasit NIL) (equal e (car a)))
            (+ 1
               (apply #'max(
                     mapcar #'(
                              lambda (x)
                                    (
                                       inaltime_nod x e T
                                    )
                              )
                              a
                     )
               ))
      )

      (T
         (+ 1
               (apply #'max(
                     mapcar #'(
                              lambda (x)
                                    (
                                       inaltime_nod x e T
                                    )
                              )
                              a
                     )
               ))
      )
   )
)

; main(a, e) = inaltime_nod(a, e, NIL)
; main(a:list, e:element)

(defun main(l e)

   (inaltime_nod l e NIL)
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

Definim un predicat auxiliar ce are aceeasi parametrii ca functia f plus un parametru ce reprezinta valoarea predicatului f(T,S1).

D. 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)) (f))) => (a g d f)

```
; elim(l niv) = {      [l]                          , l e atom si niv % 2 = 0
;               {      []                         , l e atom si niv % 2 = 1
;               { elim(l2, niv+1) U ... U main(l2,niv+1) , altfel
; elim(l:list, niv:intreg)

(defun elim (l niv)

   (cond

      (
         (AND (atom l) (equal (mod niv 2) 0))
            (list l)
      )

      (
         (AND (atom l) (equal (mod niv 2) 1))
            NIL
      )

      (T
         (mapcan #'(lambda (x)
                        (
                           elim x (+ niv 1)
                        )
                  )
                  l
         )
      )
   )
)

; main(l) = elim(l,-1)
; main(l:list)

(defun main(l)

   (elim l -1)
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
4. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

O sa definim un predicat auxiliar.

Nu se mai da

B. Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care construiește o listă care are câte un nivel pentru fiecare nivel existent în lista inițială și pe fiecare nivel are 3 elemente: numărul atomilor numerici de pe acest nivel din lista inițială, o sublistă care conține aceste informații pentru restul nivelurilor și numărul atomilor nenumerici de pe acest nivel din lista inițială. De exemplu, pentru lista (A B (4 A 3) 11 (5 (A (B) C 10) (1(2(3(4)5)6)7) X Y Z) rezultatul va fi (1 (3 (3 (2 (2 (1 0) 0) 1) 2) 4) 2).

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                        { l1 + sumaLista(l2 ... ln), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
 sumaLista(T, Rest),
 S is H + Rest.

% conditie(l) = { fals, sumaLista(L) % 2 = 0
%  { adevarat, altfel (sumaLista(L) % 2 = 1)

% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
 sumaLista(L, S),
 S mod 3 =:= 0.

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%          daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C),
 conditie(C).
```

C. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista submulțimilor cu cel puțin **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)

```
% toateSubm(L, K, Lung) = { [] , K > Lung
% { U combinariConditie(L,K) +toateSubm(L,K+1,Lung)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
                        K > Lung,
                        !.
toateSubm(L, K, Lung, [R|O]):-
                          findall(O1,combinariConditie(L, K, O1),R),
                          K1 is K + 1,
                          toateSubm(L, K1, Lung, O).

% lungime(l) = { 0, daca vida(l)
%       { 1 + lungime(l2 ... ln), altfel
%
% lungime(L: list, Len)
% L - lista pentru care trebuie aflata lungimea
% Len - lungimea listei date
%
% Model de flux: (i, i) - determinist, (i, o) - determinist
% Folosim (i, o) - determinist.

lungime([], 0).
lungime([_|T], Len):-
 lungime(T, Len1),
 Len is Len1 + 1.

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
       !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(l) = { toateAranjamentele(l, 2, lungime(l))
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(L, N, LC):-
 lungime(L, Len),
 sortare(L, L1),
 toateSubm(L1, N, Len, LC).
```

D. 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)))

```
; substituie(l, niv, k, e) = {   l                                   , l e atom si niv != k
;                           {   e                                   , l e atom si niv = k
;                           {   substituie(l2,niv,k,e) U ... U substituie(ln,niv,k,e) , altfel
;
; substituie(l:list, niv:intreg, k:intreg, e:element)

(defun substituie(l niv k e)

   (cond

      (
         (AND (atom l) (not (equal niv k)))

            l
      )

      (
         (AND (atom l) (equal niv k))

            e
      )

      (
         T
            (mapcar #'(lambda (x)
                              (substituie x (+ niv 1) k e)
                     )
                     l
            )
      )

   )
)

; main(l, k) = substituie(l,0,k)
; aceasta este functia main
; main(l:list, k:integer)

(defun main(l k e)

   (substituie l -1 k e)

)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

Am definit un predicat auxiliar, aux, ce are 3 parametrii:
K1 - care este egal cu K + 1
X - care este variabila de output
Y - rezultatul apelului predicatului f(K1, Y)
Acest predicat are 3 ramuri, acestea fiind asemanatoare cu ramurile vechi ale predicatului f, diferentele fiind inlaturarea incremantarii lui K si a apelului recursiv f(K1, Y).

D. 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)) (f))) => (a g d f)

```
; nod_pare(l, niv) = {        l                                , l e atom si niv % 2 = 0
;                    {        NIL                              , l e atom si niv % 2 = 1
;                    { nod_pare(l1,niv+1) U ... U nod_pare(ln,niv+1)    , altfel (l e lista)
; nod_pare(l:list, niv:intreg)

(defun nod_pare(l niv)

   (cond

      (
         (AND (atom l) (equal 0 (mod niv 2)))
            (list l)
      )

      (
         (AND (atom l) (equal 1 (mod niv 2)))
            NIL
      )

      (T
         (mapcan #'(lambda (x)
                        (
                           nod_pare x (+ niv 1)
                        )
                  )
                  l
         )
      )
   )
)

; main(l) = nod_pare(l, -1)
; main(l:list)
(defun main(l)

   (nod_pare l -1)
)
```

C. 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)

```
%sumaLista(l1,...,ln) = {    0                    , n = 0
%                        { l1 + sumaLista(l2,...,ln) , altfel
%sumaLista(l:list, s:integer)
% modele de flux - (i, o) - determinist - il folosim
%                 (i, i) - determinist
sumaLista([], 0):-!.
sumaLista([H|T], S):-
                  sumaLista(T, S1),
                  S is S1 + H.

% conditie(l) = {    adevarat      , sumaLista(l) % 2 = 0
%               {    fals          , altfel (sumaLista(l) % 2 = 1)
% conditie(l:list)
% model de flux - (i)
conditie(L):-
          sumaLista(L, S),
          S mod 2 =:= 0.

% comb(l1,...,ln, k) = 1. [l1] daca k = 1
%                      2. comb(l2,...,ln, k)
%                      3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
                comb(T, K, C).
comb([H|T], K, [H|C]):-
                    K > 1,
                    K1 is K - 1,
                    comb(T, K1, C).
% combinariConditie(l, k, S) = 1. comb(l, k),daca conditie(comb(l, k),S)
%                                                  - adev
%
% combinariConditie(L: List, K: Integer, C: List, S:integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% C - output value
% S - suma pe care o verificam
% Model de flux: (i, i, i) - determinist, (i, i, o) -
% nedeterminist Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
                        comb(L, K, C),
                        conditie(C).

% main(l) = { toateSubm(l, 0, lung_lista(l))
%
% Model de flux (i, i, o) - determinist, (i, i, i) - determinist
% Vom folosi (i, i, o) - determinist
main(L, K, LC):-
             findall(O1, combinariConditie(L, K, O1), LC).
```

D. 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)))

```
; inlocuire(l, niv) = {        0                                  , l e atom si niv % 2 = 0
;                     {        l                                  , l e atom si niv % 2 = 1
;                     { inlocuire(l1,niv+1) U ... U inlocuire(ln,niv+1) , altfel (l e lista)
; inlocuire(l:list, niv:intreg)
(defun inlocuire(l niv)

    (cond

        (
            (AND (atom l) (equal 0 (mod niv 2)))
               0
        )

        (
            (AND (atom l) (equal 1 (mod niv 2)))
               l
        )

        (T
            (mapcar #'(lambda (x)
                              (
                                 inlocuire x (+ niv 1)
                              )
                      )
                      l
            )
        )
    )
)

; main(l) = inlocuire(l, 0)
; main(l:list)
(defun main(l)

    (inlocuire l 0)
)
```

C. 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]]

```
% pare(l) = { 0, daca vida(l)
%          { (1 - l1 % 2) + pare(l2 ... ln), altfel
% pare(L: list, Len:integer)
% L - lista pentru care trebuie aflata nr de nr pare
% Len - nr de nr pare
%
% Model de flux: (i, i) - determinist, (i, o) - determinist
% Folosim (i, o) - determinist.
pare([], 0).
pare([H|T], Len):-
                pare(T, Len1),
                Ct is H mod 2,
                Ctt is 1 - Ct,
                Ct1 is Ctt,
                Len is Len1 + Ct1.

% impare(l) = { 0, daca vida(l)
%           { l1 % 2 + impare(l2 ... ln), altfel
% impare(L: list, Len:integer)
% L - lista pentru care trebuie aflata nr de nr impare
% Len - nr de nr impare
%
% Model de flux: (i, i) - determinist, (i, o) - determinist
% Folosim (i, o) - determinist.
impare([], 0).
impare([H|T], Len):-
                  impare(T, Len1),
                  Ct is H mod 2,
                  Len is Len1 + Ct.


% conditie(l) = { fals, pare(L) % 2 = 1 sau impare(L) % 2 = 0
%             { adevarat, altfel (pare(L) % 2 = 0 si impare(L) % 2 =1)
% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
          pare(L, P),
          P mod 2 =:= 0,
          impare(L, I),
          I mod 2 =:= 1.

% comb(l1,...,ln, k) = 1. [l1] daca k = 1
%                    2. comb(l2,...,ln, k)
%                    3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
               comb(T, K, C).
comb([H|T], K, [H|C]):-
                  K > 1,
                  K1 is K - 1,
                  comb(T, K1, C).

% combinariConditie(l, k) = 1. comb(l, k),
%                           daca conditie(comb(l, k)) - adev
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
                     comb(L, K, C),
                     conditie(C).

% genereaza Lista(a, b) = { [] , a > b
% { a + genereazaLista(a+1,b), altfel
%
% genereazaLista(a:integer,b:integer,o:list)
% a - capatul inferior al intervalului
% b - capatul superior al intervalului
% o - intervalul [a,b]
genereazaLista(A, B, []):-
                     A > B,
                     !.
genereazaLista(A, B, [A|O]):-
                       A1 is A + 1,
                       genereazaLista(A1, B, O).

% toateSubm(L, K, Lung) = { [] , K > Lung
%                         { U combinariConditie(L,K) +toateSubm(L,K+1,Lung)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
                     K > Lung,
                     !.
toateSubm(L, K, Lung, [R|O]):-
                       findall(O1,combinariConditie(L, K, O1),R),
                       K1 is K + 1,
                       toateSubm(L, K1, Lung, O).

% main(A, B) = toateSubm(genereazaLista(A,B), 2, B-A+1)
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(A, B, LC):-
            genereazaLista(A, B, L),
            Len is B - A + 1,
            toateSubm(L, 1, Len, LC).
```

D. 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))).

```
; inlocuire(l, niv) = {     l + 1                                , l e atom, l e numar, l % 2 = 1, niv % 2 = 0
;                     {     l                                    , l e atom, l e numar, niv % 2 = 1
;                     {     l                                    , l e atom, l nu e numar
;                     { inlocuire(l1,niv+1) U ... U inlocuire(ln,niv+1) , altfel
; inlocuire(l:list, niv:intreg)

(defun inlocuire(l niv)

   (cond

      (
         (AND (atom l) (numberp l) (equal 1 (mod l 2)) (equal 0 (mod niv 2)))
            (+ l 1)
      )

      (
         (AND (atom l) (numberp l) (equal 1 (mod niv 2)))
            l
      )

      (
         (AND (atom l) (not (numberp l)))
            l
      )

      (T
         (mapcar #'(lambda(x)
                        (
                           inlocuire x (+ niv 1)
                        )
                  )
                  l
         )
      )
   )
)

; main(l) = inlocuire(l, 0)
; main(l:list)
(defun main(l)

   (inlocuire l 0)
)
```

C. 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)

```
% progresie(l1l2l3,...,ln) = { adevarat            , n = 2
%                           { fals , n > 2 si abs(l1-l2)!=abs(l2-l3)
%                           { progresie(l2,...,ln) , altfel
% progresie(l:list)
% model de flux - (i)
progresie([_,_]):-!.
progresie([L1,L2,L3|T]):-
                    D1 is abs(L1-L2),
                    D2 is abs(L2-L3),
                    D1 =:= D2,
                    progresie([L2,L3|T]).

% conditie(l) = { fals, pare(L) % 2 = 1 sau impare(L) % 2 = 0
%               { adevarat, altfel (pare(L) % 2 = 0 si impare(L) % 2 =1)
% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
        progresie(L).

% comb(l1,...,ln, k) = 1. [l1] daca k = 1
%                    2. comb(l2,...,ln, k)
%                    3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
                comb(T, K, C).
comb([H|T], K, [H|C]):-
                  K > 1,
                  K1 is K - 1,
                  comb(T, K1, C).

% combinariConditie(l, k) = 1. comb(l, k),
%                           daca conditie(comb(l, k)) - adev
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
                      comb(L, K, C),
                      conditie(C).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
      !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(L, K) = U combinariConditie(L, K)
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(L, K, LC):-
            sortare(L, L1),
            findall(O1, combinariConditie(L1,K,O1), LC).
```

D. 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)))

```
; elimina(l) = {    NIL                    , l e atom numeric si l % 3 = 0
;              {     l                     , l e atom si l % 3 != 1 si l % 3 != 0
;              { elimina(l1) U ... U elimina(ln)  , altfel
; elimina(l:list)
(defun elimina(l)

   (cond

      (
         (AND (atom l) (numberp l) (equal 0 (mod l 3)))
            NIL
      )

      (
         (AND (atom l))
            (list l)
      )

      (T
         (list(mapcan #'elimina l))
      )
   )
)

; main(l) = elimina(l)[1]
; main(l:list)
(defun main(l)

   (car (elimina l))
)
```

C. 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)

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                        { l1 + sumaLista(l2 ... ln), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
 sumaLista(T, Rest),
 S is H + Rest.

% verif(l1...,ln) = { adevarat        , n = 1 si l1 % 2 = 1
%                   { fals            , n = 2 si l1 % 2 = 0
%                   { verif(l2,...,ln), altfel
% verif(l:list)
% l - lista pe care o verificam daca are ultimul element impar
% model de flux (i)
verif([H]):-
        H mod 2 =:= 1,
        !.
verif([_|T]):-
          verif(T).

% conditie(l) = { fals, sumaLista(L) % 2 = 0 sau verif(L) = fals
%  { adevarat, altfel (sumaLista(L)%2=1 si verif(L)=adev)

% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L, SUMA):-
 sumaLista(L, S),
 S =:= SUMA,
      verif(L).

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%                           daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist - il folosim pe asta
%                (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C).

% insereaza(e, l1,...,ln) = 1. e + l1l2...ln
%                           2. l1 + insereaza(e, l2...ln)
%
% insereaza(E: element, L:List, LRez:list)
% E - elementul pe care dorim sa-l inseram pe toate pozitiile
% L - lista in care o sa fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                           insereaza(E, T, Rez).

% perm(l1,...,ln) = 1.      []                              daca n = 0
%                   2.      insereaza(l1, permutari(l2,...,ln))
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
                   permutari(T, L),
                   insereaza(E, L, P).

% aranjamente(l1,...,ln, k, SUMA) = 1. l ,l=permutari(combinari(L,K)),
%                                      daca conditie(l,SUMA) = true
%
% aranjamente(L:list, K:element, O:list, SUMA:integer)
% L - multimea numerelor
% K - nr de elemente din aranjament
% O - aranjamentul curent
% SUMA - suma cu care comparam suma permutarii
% model de flux (i, i, o, i) - nedeterminist
%               (i, i, i, i) - determinist
aranjamente(L, K, O, SUMA):-
 combinariConditie(L, K, O1),
 permutari(O1, O),
       conditie(O, SUMA).

% main(l) = { toateAranjamentele(l, 2, lungime(l))
%
% Model de flux (i, i, i, i) - determinist, (i, i, i, o) - determinist
% Vom folosi (i, i, i, o) - determinist
main(L, N, S, LC):-
 findall(O1, aranjamente(L, N, O1, S), LC).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. 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))
                  )
                )
        )
)
```

Am folosit o lambda.

C. 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)

```
% sumaLista(l1 ... ln) = { 0, n = 0
% { l1 + sumaLista(l2 ... ln), altfel
% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
sumaLista(T, Rest),
S is H + Rest.
% conditie(l) = { fals, sumaLista(L) % 2 = 1
% { adevarat, altfel (sumaLista(L) % 2 = 0)
% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
sumaLista(L, S),
S mod 2 =:= 0.
% comb(l1,...,ln, k) = 1. [l1] daca k = 1
% 2. comb(l2,...,ln, k)
% 3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
comb(T, K, C).
comb([H|T], K, [H|C]):-
K > 1,
K1 is K - 1,
comb(T, K1, C).
% combinariConditie(l, k) = 1. comb(l, k),
% daca conditie(comb(l, k)) - adev
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
comb(L, K, C),
conditie(C).

% lung_lista(l1,...,ln) = { 0, n = 0
% { 1 + lung_lista(l2,..,ln), altfel
% lung_lista(l:list, lung:integer)
% l - lista careia ii determinam lungimea
% model de flux - (i, o) - determinist - il folosim
% - (i, i) - determinist
lung_lista([], 0).
lung_lista([_|T], Lung):-
lung_lista(T, Lung1),
Lung is Lung1 + 1.
% toateSubm(L, K, Lung) = { [] , K > Lung
% { U combinariConditie(L,K) + toateSubm(L, K+1,
% Lung)
% toateSubm(L, K, Lung, O)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
K > Lung,
!.
toateSubm(L, K, Lung, [R|O]):-
findall(O1,combinariConditie(L, K, O1),R),
K1 is K + 1,
toateSubm(L, K1, Lung, O).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
       !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).

% main(l) = { toateSubm(sortare(l), 2, lung_lista(l))
%
% Model de flux (i, i) - determinist, (i, o) - determinist
% Vom folosi (i, o) - determinist
main(L, LC):-
lung_lista(L, Len),
sortare(L, L1),
toateSubm(L1, 0, Len, LC).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN F(L)
        (COND
                ((NULL L) 0)
                (T ((lambda (X)
                        (cond
                                ((> X 2) (+ X (CAR L)))
                                (T (+ X 1))
                        )
                    )(F (CDR L))
                  )
                )
        )
)
```

Am folosit o lambda.

D. 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)))

```
; subs(l, k, niv) = {       l - 1                    ,  l e atom numeric si l > k si niv % 2 = 1
;                   {       l                        , l e atom
;                   { subs(l1,k,niv+1) U ... U subs(ln,k,niv+1) , altfel
; subs(l:list, k:integer, niv:integer)
(defun subs(l k niv)

   (cond

      (
         (AND (atom l) (numberp l) (> l k) (equal 1 (mod niv 2)))
            (- l 1)
      )

      (
         (atom l)
            l
      )

      (T
         (mapcar #'(lambda (x)
                        (
                           subs x k (+ niv 1)
                        )
                  )
                  l
         )
      )
   )
)

; main(l, k) = subs(l, k, 0)
; main(l:list, k:integer)
(defun main(l k)

   (subs l k 0)
)
```

C. Scrieţi un program PROLOG care determină dintr-o listă formată din numere întregi lista subşirurilor cu cel puțin 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)

```
% comb(l1,...,ln, k) = 1. [l1] daca k = 1
% 2. comb(l2,...,ln, k)
% 3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
comb(T, K, C).
comb([H|T], K, [H|C]):-
K > 1,
K1 is K - 1,
comb(T, K1, C).

% lung_lista(l1,...,ln) = { 0, n = 0
% { 1 + lung_lista(l2,..,ln), altfel
% lung_lista(l:list, lung:integer)
% l - lista careia ii determinam lungimea
% model de flux - (i, o) - determinist - il folosim
% - (i, i) - determinist
lung_lista([], 0).
lung_lista([_|T], Lung):-
lung_lista(T, Lung1),
Lung is Lung1 + 1.
% toateSubm(L, K, Lung) = { [] , K > Lung
%                         { U comb(L,K) + toateSubm(L, K+1,Lung)
% toateSubm(L:list, K:integer, Lung:integer, O:list)
% L - list, lista de elemente
% K - integer, lung curenta a submult
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,i,o) - determinist - il folosim pe acesta
% (i,i,i,i) - determinist
toateSubm(_, K, Lung, []):-
K > Lung,
!.
toateSubm(L, K, Lung, [R|O]):-
findall(O1,comb(L, K, O1),R),
K1 is K + 1,
toateSubm(L, K1, Lung, O).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln    , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
       !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).

% main(l) = { toateSubm(sortare(l), 2, lung_lista(l))
%
% Model de flux (i, i) - determinist, (i, o) - determinist
% Vom folosi (i, o) - determinist
main(L, LC):-
lung_lista(L, Len),
sortare(L, L1),
toateSubm(L1, 2, Len, LC).
```

D. 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)))

```
; elimina(l, niv) = {     NIL                    , l e atom numeric si l % 2 = 0 si niv % 2 = 1
;                   {     [l]                    , l e atom
;                   { elimina(l1,niv+1) U ... U elimina(ln,niv+1)  , altfel
; elimina(l:list, niv:intreg)
(defun elimina(l niv)

   (cond

      (
         (AND (atom l) (numberp l) (equal 0 (mod l 2)) (equal 1 (mod niv 2)) )
            NIL
      )

      (
         (atom l)
            (list l)
      )

      (T
         (list(mapcan #'(lambda (x)
                              (
                                 elimina x (+ niv 1)
                              )
                        )
                        l
               )
         )
      )
   )
)

; main(l) = elimina(l,0)[1]
; main(l:list)
(defun main(l)

   (car (elimina l 0))
)
```

C. 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)

```
% progresie(l1l2l3,...,ln) = { adevarat            , n = 2
%                           { fals , n > 2 si abs(l1-l2)!=abs(l2-l3)
%                           { progresie(l2,...,ln) , altfel
% progresie(l:list)
% model de flux - (i)
progresie([_,_]):-!.
progresie([L1,L2,L3|T]):-
                    D1 is abs(L1-L2),
                    D2 is abs(L2-L3),
                    D1 =:= D2,
                    progresie([L2,L3|T]).

% impare(l1...ln) = {  adevarat       , n = 0
%                   {  fals           , n > 0 si l1 % 2 = 0
%                   {  impare(l2...ln) , altfel
% impare(l:list)
% l - lista pe care o verificam
% model de flux - (i)
impare([]):-!.
impare([H|T]):-
            H mod 2 =:= 1,
            impare(T).

% conditie(l) = { fals, pare(L) % 2 = 1 sau impare(L) % 2 = 0
%               { adevarat, altfel (pare(L) % 2 = 0 si impare(L) % 2 =1)
% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
           impare(L),
           progresie(L).

% comb(l1,...,ln, k) = 1. [l1] daca k = 1
%                      2. comb(l2,...,ln, k)
%                      3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
              comb(T, K, C).
comb([H|T], K, [H|C]):-
                    K > 1,
                    K1 is K - 1,
                    comb(T, K1, C).

% combinariConditie(l, k) = 1. comb(l, k),
%                           daca conditie(comb(l, k)) - adev
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
                        comb(L, K, C),
                        conditie(C).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
     !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(L, K) = U combinariConditie(L, K)
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(L, K, LC):-
               sortare(L, L1),
               findall(O1, combinariConditie(L1,K,O1), LC).
```

D. 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 funcție 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

```
; elimina(l, niv, k) = {          l                                          , l atom si niv != k
;                     {          []                                       , l atom si niv = k
;                     { elimina(l1, niv+1, k) U elimina(l2,niv,k+1) U ... U elimina(ln,niv,k+1)  , altfel
; elimina(l:list, niv:intreg, k:intreg)

(defun elimina(l niv k)

    (cond

        (
            (AND (atom l) (not(equal niv k)))
                (list l)
        )

        (
            (AND (atom l) (equal niv k))
                NIL
        )

        (T
            (list (mapcan #'(lambda (x)
                              (
                                elimina x (+ niv 1) k
                              )
                          )
                          l
            )
            )
        )
    )
)

; main(l, k) = elimina(l, 0, k)
; l - list
; k - intreg
(defun main(l k)

    (car (elimina l 0 k))
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN F(L)
        (COND
                ((NULL L) 0)
                (T ((lambda (X)
                        (cond
                                ((> X 1) (F (CDR L)))
                                (T (+ X (F (CDR L))))
                        )
                    )(F (CAR L))
                  )
                )
        )
)
```

Am folosit o lambda.

C. 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.

```
% genereazaLista(i, n) = { [n]                    , i = n
%                        { i + genereazaLista(i+1, n), altfel (i < n)
%
% genereazaLista(i:intreg, n:intreg, r:list)
% i - numarul pe care l adaugam in lista
% n - val maxima din lista
% r - lista rezultat
%
% model de flux (i, i, o) - determinist - cel folosit de noi
%             (i, i, i) - determinist
genereazaLista(N, N, [N]):-!.
genereazaLista(I, N, [I|R]):-
                        I < N,
                        I1 is I + 1,
                        genereazaLista(I1, N, R).

% insereaza(e, l1,...,ln) = 1. e + l1l2...ln
%                           2. l1 + insereaza(e, l2...ln)
%
% insereaza(E: element, L:List, LRez:list)
% E - elementul pe care dorim sa-l inseram pe toate pozitiile
% L - lista in care o sa fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                        insereaza(E, T, Rez).

% perm(l1,...,ln) = 1.     []                          daca n = 0
%                   2.     insereaza(l1, permutari(l2,...,ln))
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
                    permutari(T, L),
                    insereaza(E, L, P).

% conditie(l1,...,ln) = {   adevarat      , n < 2
%                       {   fals          , n >= 2 si abs(l1-l2)>2
%                       {   conditie(l2,...,ln), n >= 2 si abs(l1-l2)<=2
% conditie(L:list)
% L - lista pe care o verificam daca respecta conditia din enunt sau nu
% model de flux (i) - determinist
conditie([_]):-!.
conditie([L1,L2|T]):-
                    D is abs(L1-L2),
                    D =< 2,
                    conditie([L2|T]).

% permConditie(l, k) = 1. permutari(l), conditie(permutari(l)) = adev
% permConditie(L: List, O:List).
% L - lista pe care trebuie sa facem permutarile cu conditie
% O - permutarea ce respecta conditia
% Model de flux: (i, i) - determinist,
%                (i, o) - nedeterminist - cel folosit
permConditie(L, O):-
                permutari(L, O),
                conditie(O).

% main(n) = U(permConditie(genereazaLista(1,n)))
%
% main(N:intreg, O:List)
% N - elementul maxim din lista noastra
% O - lista rezultat
% Model de flux (i, o) - determinist - folosim
%              (i, i) - determinist
main(N, O):-
        N2 is 2 * N,
        N1 is N2 - 1,
        genereazaLista(N, N1, L),
        findall(O1, permConditie(L, O1), O).
```

D. 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)))

```
; substituie(l, niv, e) = {   l                                   , l e atom si niv % 2 = 0
;                        {   e                                   , l e atom si niv % 2 = 1
;                        {   substituie(l2,niv,e) U ... U substituie(ln,niv,e) , altfel
;
; substituie(l:list, niv:intreg, e:element)

(defun substituie(l niv e)

   (cond

      (
         (AND (atom l) (equal 0 (mod niv 2)))

            l
      )

      (
         (AND (atom l) (equal 1 (mod niv 2)))

            e
      )

      (
         T
            (mapcar #'(lambda (x)
                              (substituie x (+ niv 1) e)
                     )
                     l
            )
      )

   )
)

; main(l, e) = substituie(l,-1,e)
; aceasta este functia main
; main(l:list, e:element)

(defun main(l e)

   (substituie l -1 e)

)
```

C. 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)

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                       { l1 + sumaLista(l2 ... ln), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
 sumaLista(T, Rest),
 S is H + Rest.

% conditie(l,SUMA) = { fals, sumaLista(L) = SUMA
%                    { adevarat, altfel (sumaLista(L)=SUMA
% conditie(L: list,SUMA:integer).
% L - lista pentru care trebuie verificata conditia
% SUMA - suma cu care comparam suma listei
% Model de flux: (i) - determinist.
conditie(L, SUMA):-
 sumaLista(L, S),
 S =:= SUMA.

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%                           daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist - il folosim pe asta
%                (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C).

% insereaza(e, l1,...,ln) = 1. e + l1l2...ln
%                           2. l1 + insereaza(e, l2...ln)
%
% insereaza(E: element, L:List, LRez:list)
% E - elementul pe care dorim sa-l inseram pe toate pozitiile
% L - lista in care o sa fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                           insereaza(E, T, Rez).

% perm(l1,...,ln) = 1.      []                              daca n = 0
%                   2.      insereaza(l1, permutari(l2,...,ln))
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
                   permutari(T, L),
                   insereaza(E, L, P).

% aranjamente(l1,...,ln, k, SUMA) = 1. l ,l=permutari(combinari(L,K)),
%                                      daca conditie(l,SUMA) = true
%
% aranjamente(L:list, K:element, O:list, SUMA:integer)
% L - multimea numerelor
% K - nr de elemente din aranjament
% O - aranjamentul curent
% SUMA - suma cu care comparam suma permutarii
% model de flux (i, i, o, i) - nedeterminist
%               (i, i, i, i) - determinist
aranjamente(L, K, O, SUMA):-
 combinariConditie(L, K, O1),
 permutari(O1, O),
       conditie(O, SUMA).

% main(l) = { toateAranjamentele(l, 2, lungime(l))
%
% Model de flux (i, i, i, i) - determinist, (i, i, i, o) - determinist
% Vom folosi (i, i, i, o) - determinist
main(L, N, S, LC):-
 findall(O1, aranjamente(L, N, O1, S), LC).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

Am definit un nou predicat pentru a evita apelul repetat al functiei f(J, S).

D. 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))).

```
; inlocuire(l) = {     l + 1                        , l e atom, l e numar, l % 2 = 0
;                {     l                          , l e atom
;                { inlocuire(l1) U ... U inlocuire(ln) , altfel
; inlocuire(l:list, niv:intreg)

(defun inlocuire(l)

   (cond

      (
         (AND (atom l) (numberp l) (equal 0 (mod l 2)))
            (+ l 1)
      )

      (
         (atom l)
            l
      )

      (T
         (mapcar #'inlocuire l)
      )
   )
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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)**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

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

Am folosit o lambda.

C. 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)

```
% insereaza(e, l1,...,ln) = 1. e + l1l2...ln
%                          2. l1 + insereaza(e, l2...ln)
%
% insereaza(E: element, L:List, LRez:list)
% E - elementul pe care dorim sa-l inseram pe toate pozitiile
% L - lista in care o sa fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                          insereaza(E, T, Rez).

% perm(l1,...,ln) = 1.      []                          daca n = 0
%                   2.      insereaza(l1, permutari(l2,...,ln))
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
                  permutari(T, L),
                  insereaza(E, L, P).

% conditie(l1,...,ln) = {   adevarat       , n < 2
%                       {   fals           , n >= 2 si abs(l1-l2)>3
%                       {   conditie(l2,...,ln), n >= 2 si abs(l1-l2)<=3
% conditie(L:list)
% L - lista pe care o verificam daca respecta conditia din enunt sau nu
% model de flux (i) - determinist
conditie([_]):-!.
conditie([L1,L2|T]):-
                  D is abs(L1-L2),
                  D =< 3,
                  conditie([L2|T]).

% permConditie(l, k) = 1. permutari(l), conditie(permutari(l)) = adev
% permConditie(L: List, O:List).
% L - lista pe care trebuie sa facem permutarile cu conditie
% O - permutarea ce respecta conditia
% Model de flux: (i, i) - determinist,
%                (i, o) - nedeterminist - cel folosit
permConditie(L, O):-
                permutari(L, O),
                conditie(O).

% main(L) = U(permConditie(L))
%
% main(L:list, O:List)
% L - lista initiala
% O - lista rezultat
% Model de flux (i, o) - determinist - folosim
%                (i, i) - determinist
main(L, O):-
           findall(O1, permConditie(L, O1), O).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN F(L)
        (COND
                ((ATOM L) -1)
                (T ((lambda(X)
                        (cond
                                ((> X 0) (+ (CAR L) X (F (CDR L))))
                                (T (F (CDR L)))
                        )
                    )(F (CAR L))
                   )
                )
        )
)
```

Am folosit o lambda.

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN F(L)
        (COND
                ((NULL L) 0)
                (T ((lambda (X)
                        (cond
                                ((> (CAR L) 0)
                                        (COND
                                            ((> (CAR L) X) (CAR L))
                                            (T X)
                                        )
                                )
                                (T X)
                        )
                    ) (F (CDR L))
                )
                )
        )
)
```

Am folosit o lambda.

C. 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)

```
% progresie(l1l2l3,...,ln) = { adevarat              , n = 2
%                           { fals , n > 2 si abs(l1-l2)!=abs(l2-l3)
%                           { progresie(l2,...,ln) , altfel
% progresie(l:list)
% model de flux - (i)
progresie([_,_]):-!.
progresie([L1,L2,L3|T]):-
                    D1 is abs(L1-L2),
                    D2 is abs(L2-L3),
                    D1 =:= D2,
                    progresie([L2,L3|T]).

% impare(l1...ln) = {    adevarat        ,  n = 0
%                   {    fals            ,  n >= 1 si l1 % 2 = 0
%                   {    impare(l2...ln) , altfel
% impare(l:list)
% l - lista pe care o verificam
% model de flux - (i)
impare([]):-!.
impare([H|T]):-
            H mod 2 =:= 1,
            impare(T).

% conditie(l) = progresie(l)
% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
          progresie(L),
          impare(L).

% comb(l1,...,ln, k) = 1. [l1] daca k = 1
%                      2. comb(l2,...,ln, k)
%                      3. l1 + comb(l2,...,ln, k - 1) k > 1
% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
comb([H|_], 1, [H]).
comb([_|T], K, C):-
               comb(T, K, C).
comb([H|T], K, [H|C]):-
                    K > 1,
                    K1 is K - 1,
                    comb(T, K1, C).

% combinariConditie(l, k) = 1. comb(l, k),
%                           daca conditie(comb(l, k)) - adev
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
                        comb(L, K, C),
                        conditie(C).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
     !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(L, K) = U combinariConditie(L, K)
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(L, K, LC):-
              sortare(L, L1),
              findall(O1, combinariConditie(L1,K,O1), LC).
```

D. 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 ()

```
; contor(l, k, niv) = {        1                              ,    l e atom si niv = k
;                     {        0                              ,    l e atom si niv != k
;                     { contor(l1,k,niv+1) + ... + contor(ln,k,niv+1)  , altfel
; contor(l:list, k:intreg, niv:intreg)
(defun contor (l k niv)

   (cond

      (
         (AND (atom l) (equal niv k))
            1
      )

      (
         (AND (atom l) (not(equal niv k)))
            0
      )

      (T
         (
            apply #'+(mapcar( lambda (x)
                              (
                                 contor x k (+ niv 1)
                              )
                           )
                           l
                     )
         )
      )
   )
)

; main(l,k) =  contor(l,k,-1)
; main(l:list, k:integer)
(defun main(l k)

   (contor l k -1)
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

Am definit un predicat auxiliar pentur a evita apelul repetat al functiei f(T, S).

D. 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 funcţie 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)))

```
; elimina(l, e) = {          l                                   , l atom si l != e
;                 {          []                                  , l atom si l = e
;                 { elimina(l1, e) U elimina(l2, e) U ... U elimina(ln, e)  , altfel
; elimina(l:list, e:element)

(defun elimina(l e)

   (cond

      (
         (AND (atom l) (not(equal l e)))
            (list l)
      )

      (
         (AND (atom l) (equal l e))
            NIL
      )

      (T
         (list (mapcan #'(lambda (x)
                           (
                             elimina x e
                           )
                     )
                     l
         )
         )
      )
   )
)

; main(l, e) = elimina(l, e)
; l - list
; e - element
(defun main(l e)

   (car (elimina l e))
)
```

C. 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]]

```
% sumaLista(l1 ... ln) = { 0, n = 0
%                        { l1 + sumaLista(l2 ... ln), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
sumaLista([], 0).
sumaLista([H|T], S):-
 sumaLista(T, Rest),
 S is H + Rest.

% conditie(l) = { fals, sumaLista(L) % 2 = 1
%  { adevarat, altfel (sumaLista(L) % 2 = 0)
% conditie(L: list).
% L - lista pentru care trebuie verificata conditia
% Model de flux: (i) - determinist.
conditie(L):-
 sumaLista(L, S),
 S mod 2 =:= 0.

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%          daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist, (i, i, o) - nedeterminist
% Folosim (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C),
 conditie(C).

% insert(L: list, E: int, O: list)
% L: lista in care trebuie inserat elementul in ordine
% E: elementul ce trebuie inserat
% O: lista cu elementul inserat
% Modelul de flux: (i, i, i) - determinist sau (i, i, o) - determinist
% Folosim (i, i, o) - determinist.
%insert(l1 l2 ... ln, el) = { [el] , n = 0
%                           { el (+) l1 ... ln   , n > 0 si el <= l1
%                           { l1 (+) insert(l2 ... ln, el) , n > 0 si el > l1

insert([], E, [E]):-!.

insert([H|T], E, O):-
 E =< H,
      !,
 O = [E, H|T].

insert([H|T], E, O):-
 E > H,
 O = [H|O1],
 insert(T, E, O1).


% sortare(L: list, O: list)
% L: lista primita (cea care trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare(l1 l2 ... ln) = { [] , n = 0
%                        { insert(sortare(l2 ... ln), l1), altfel (n > 0)
sortare([], []):-!.

sortare([H|T], O):-
 sortare(T, O1),
 insert(O1, H, O).


% main(l) = { toateAranjamentele(l, 2, lungime(l))
%
% Model de flux (i, i, i) - determinist, (i, i, o) - determinist
% Vom folosi (i, i, o) - determinist
main(L, N, LC):-
 sortare(L, L1),
 findall(O1, combinariConditie(L1, N, O1), LC).
```

D. 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) **b)** k=5 => ()

```
; elim(l niv k) = {       [l]                          , l e atom si niv = k
;                 {       []                         , l e atom si niv != k
;                 { elim(l2,niv+1,k) U ... U main(l2,niv+1,k) , altfel
; elim(l:list, niv:intreg, k:intreg)

(defun elim (l niv k)

   (cond

      (
         (AND (atom l) (equal niv k))
            (list l)
      )

      (
         (AND (atom l) (not (equal niv k)))
            NIL
      )

      (T
         (mapcan #'(lambda (x)
                        (
                           elim x (+ niv 1) k
                        )
                  )
                  l
         )
      )
   )
)

; main(l,k) = elim(l,-1,k)
; main(l:list,k:integer)

(defun main(l k)

   (elim l -1 k)
)
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

Am definit un predicat auxiliar

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

```
(DEFUN F(N)
   (cond

      (
         (= N 1)
            1
      )

      (t
         ((lambda (x)
               (cond
                  (
                     (> X 2)
                        (- N 2)
                  )

                  (
                     (> X 1)
                        X
                  )

                  (t
                     (- X 1)
                  )
               )
         )
         (F (- N 1))
         )
      )
   )
)
```

Am folosit o lambda

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

Am definit un predicat auxiliar.

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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))**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

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

Am folosit o lambda.

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

A. Fie **G** o funcție LISP și fie următoarea definiţie

```
(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)**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

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

Am folosit o lambda.

C. 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)

```
% prodLista(l1 ... ln) = { 1, n = 0
%                        { l1 * prodLista(l2 ... ln), altfel

% prodLista(L: List, P: Integer).
% L - lista pentru care trebuie calculata suma
% P - prod listei date
% Model de flux: (i, i) - determinist, (i, o) - determinist.
% Folosim modelul de flux (i, o) - determinist.
prodLista([], 1).
prodLista([H|T], S):-
 prodLista(T, Rest),
 S is H * Rest.

% conditie(l,Prod) = { adevarat, prodLista(L) = Prod
%                    { false, altfel (prodLista(L) != Prod)
% conditie(L: list,Prod:integer).
% L - lista pentru care trebuie verificata conditia
% Prod - prod cu care comparam prod listei
% Model de flux: (i, i) - determinist.
conditie(L, Prod):-
 prodLista(L, P),
 P =:= Prod.

% comb(l1,...,ln, k) =     1. [l1]                    daca k = 1
%                          2. comb(l2,...,ln, k)
%                          3. l1 + comb(l2,...,ln, k - 1) k > 1

% comb(L: list, K:integer, C:list)
% L - multimea de numere
% K - numarul de numere din combinare
% C - lista rezultat

% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist

comb([H|_], 1, [H]).
comb([_|T], K, C):-
 comb(T, K, C).

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

% combinariConditie(l, k) = 1. comb(l, k),
%                           daca conditie(comb(l, k)) - adev

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinarile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinist - il folosim pe asta
%                (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
 comb(L, K, C).

% insereaza(e, l1,...,ln) = 1. e + l1l2...ln
%                           2. l1 + insereaza(e, l2...ln)
%
% insereaza(E: element, L:List, LRez:list)
% E - elementul pe care dorim sa-l inseram pe toate pozitiile
% L - lista in care o sa fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                           insereaza(E, T, Rez).

% perm(l1,...,ln) = 1.     []                              daca n = 0
%                   2.     insereaza(l1, permutari(l2,...,ln))
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
                   permutari(T, L),
                   insereaza(E, L, P).

% aranjamente(l1,...,ln, k, SUMA) = 1. l ,l=permutari(combinari(L,K)),
%                                      daca conditie(l,SUMA) = true
%
% aranjamente(L:list, K:element, O:list, SUMA:integer)
% L - multimea numerelor
% K - nr de elemente din aranjament
% O - aranjamentul curent
% Prod - prod cu care comparam prod permutarii
% model de flux (i, i, o, i) - nedeterminist
%               (i, i, i, i) - determinist
aranjamente(L, K, O, Prod):-
 combinariConditie(L, K, O1),
 permutari(O1, O),
       conditie(O, Prod).

% main(l,n,p) = U aranjamente(l, N, p)
%
% Model de flux (i, i, i, i) - determinist, (i, i, i, o) - determinist
% Vom folosi (i, i, i, o) - determinist
main(L, N, P, LC):-
 findall(O1, aranjamente(L, N, O1, P), LC).
```

Programare logică și funcțională
- examen scris -

Notă

1. Subiectele se notează astfel: of - 1p; A – 1.5p; B - 2.5p; C - 2.5p; D - 2.5p.
2. Problemele Prolog vor fi rezolvate în SWI Prolog. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare predicat folosit; (3) specificarea fiecărui predicat (semnificația parametrilor, model de flux, tipul predicatului - determinist/nedeterminist).
3. Problemele Lisp vor fi rezolvate în Common Lisp. Se cere: (1) explicarea codului și a raționamentului; (2) modelul recursiv de rezolvare, pentru fiecare funcție folosită; (3) specificarea fiecărei funcții (semnificația parametrilor).

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. Nu redefiniți predicatul. Justificați răspunsul.

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

Am folosit un predicat auxiliar.