

# 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 definiție de funcție în LISP

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

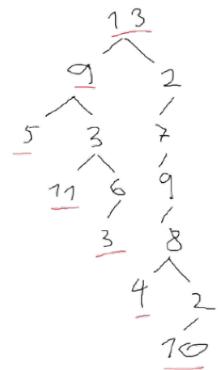
Rescrieți această definiție pentru a evita dublul apel recursiv **(F (CAR L1) L2)**. Nu redefiniți funcția. Nu folosiți SET, SETQ, SETF. Justificați răspunsul.

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

Folosind o funcție lambda ce are ca și parametru un I, putem evita apelul repetat al funcției **(F (CAR L1) L2)** pentru ca acesta să se efectueze o singura dată, în momentul în care este rulată funcția lambda.

- B. Dându-se un arbore binar în care nodurile conțin informații numerice și arborele este reprezentat sub forma unei liste în care fiecare nod este urmat de un număr (0, 1 or 2) care reprezintă numărul de descendenți ai nodului respectiv, se cere un program SWI-Prolog care calculează suma primului element de pe fiecare nivel. De exemplu, pentru lista [13, 2, 9, 2, 5, 0, 3, 2, 11, 0, 6, 1, 3, 0, 2, 1, 7, 1, 9, 1, 8, 2, 4, 0, 2, 1, 10, 0] rezultatul va fi 55.

A fost scos subiectul asta



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]  $\Rightarrow$  [[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 curentă a elementelor din C
%LE - lungimea lui C
%C - colectoare
%O - un aranjament al listei date ce are suma impara și lungimea pară
%model flux: (i,i,i,i,i) - determinist, (i,i,i,i,o) - nedeterminist
%vom folosi varianta nedeterminista (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 element
%E - un element din lista
%model de flux (i,i) - determinist, (i,o) - nedeterminist
%vom folosi modelul (i,o) nedeterminist
```

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

```
%elimina_prim(L:lista,E:element,O:lista)
%elimina prima apariție a lui E din lista L
%L - lista pe care operăm
%E - elementul ce-l stergem
%O - lista rezultată în urma eliminării
%model de flux (i,i,i) - determinist; (i,i,o) - nedeterminist
%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 funcție LISP care să aibă ca rezultat lista inițială în care toate aparitările 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)) și **e1** este B => (1 (2 B (3 B)) (B))  
**b)** dacă lista este (1 (2 (3))) și **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 și următoarea definiție de predicat PROLOG **f(list, integer)**, având modelul de flux (i, o):

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

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

$f([], -1).$   
 $f([H|T], S) :- f(T, S1),$   
 $\quad p\_aux([H|T], S, S1).$   
 $p\_aux([H|_], S, S1) :-$   
 $\quad H > 0,$   
 $\quad S1 < H,$   
 $\quad !,$   
 $\quad S \text{ is } H.$   
 $p\_aux(_, S, S1) :- S \text{ is } S1.$

Definim un predicat auxiliar ce are aceeași parametru ca funcția f plus un parametru ce reprezintă 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 noastră
% 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 multimii  $1..N$ , cu proprietatea că valoarea absolută a diferenței între 2 valori consecutive din permutare este  $\geq 2$ . Se vor scrie modelele matematice și modelele de flux pentru predicatelor folosite.

**Exemplu:** pentru  $N=4 \Rightarrow [[3,1,4,2], [2,4,1,3]]$  (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, I1,...,In) = 1. e + I1I2...In
%           2. I1 + insereaza(e, I2...In)
%
% 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(I1,...,In) = 1. [] daca n = 0
%           2. insereaza(I1, permutari(I2,...,In))
%
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
    permutari(T, L),
    insereaza(E, L, P).

% conditie(I1,...,In) = { adevarat , n < 2
%           { fals , n >= 2 si abs(I1-I2)<2
%           { conditie(I2,...,In), n >= 2 si abs(I1-I2)>=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)) = adevarat
% 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** și **e1=f** rezultă lista (1 f (2 d (f))).

```

; substituie(l, e, e1, niv) = {   , 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)))
      )
    )
    (
      (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)))
      )
    )
  )
; 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 definiție 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)**.

- B.** Un arbore n-ar poate fi memorat ca o listă liniară în care fiecare nod este urmat de numărul de descendenți. Dându-se o listă liniară care reprezintă un arbore n-ar, se cere un program Lisp care determină, sub forma unei liste, al k-lea descendente al rădăcinii din arbore. De exemplu, pentru arborele (A 5 B 2 E 0 F 3 G 0 H 0 I 0 C 1 J 1 K 2 L 0 M 0 D 4 N 0 O 0 P 2 R 0 S 1 T 0 Q 0 U 0 V 1 Z 2 T 0 W 0) și k = 3 rezultatul va fi (D 4 N 0 O 0 P 2 R 0 S 1 T 0 Q 0), iar pentru același arbore și k = 5 rezultatul va fi (V1 Z 2 T 0 W 0).

A fost scos subiectul asta

**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]  $\Rightarrow$  [[2,3],[3,2],[3,4],[4,3]] (nu neapărat în această ordine)

Subiect anterior rezolvat

- 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 contină 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 definiție de funcție 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([ ], 1).
% prodLista([_|T], P) :- prodLista(T, P1), P is P1 * prodLista([ ], P1).

% conditie([ ], v).
% conditie([_|T], v) :- conditie(T, v).

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

% prodLista(L, P) :- prodLista(L, P).

% prodLista([ ], 1).

% prodLista([\_|T], P) :- prodLista(T, P1), P is P1 \* prodLista([ ], P1).

% conditie([ ], v).

% conditie([\_|T], v) :- conditie(T, v).

% comb([ ], K, C).

% comb([H|T], K, C) :- comb(T, K1), K is K1 + 1, comb([H], 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 predicatelor 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 combinariile 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. []
%           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 definiție 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(I1 ... In) = { 0, n = 0
%                               { I1 + sumaLista(I2 ... In), altfel
```

```
% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculata suma
% S - suma listei date
```

PB C

```
% 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(I1,...,In, k) = 1. [I1]           daca k = 1
%                   2. comb(I2,...,In, k)
%                   3. I1 + comb(I2,...,In, 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) - determinant
```

```
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)) - adevarat
```

```
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinariile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinant, (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 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  $[2,3,4] \Rightarrow [[2,3,4]]$

```
% lungime(l) = { 0, daca l = []
% { (1 - l1 \% 2) + lungime(l2 ... ln), altfel
% lungime(L: list, Len:integer)
% L - lista pentru care trebuie aflată 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_liste(l1,...,ln) = { 0, n = 0
% { 1 + lung_liste(l2,...,ln), altfel
% lung_liste(l: list, lung:integer)
% l - lista careia îl determinăm lungimea
% model de flux - (i, o) - determinist - îl folosim
% - (i, i) - determinist
lung_liste([], 0).
lung_liste([_|T], Lung):- lung_liste(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 - listă, lung curentă a submult
% K - integer, lung curentă a submult
% Lung - Lungimea listei l
% O - rezultatul
% model de flux (i,i,i,o) - determinist - îl 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 în care trebuie inserat elementul în 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 și el <= l1
% { l1 (+) insert(l2 ... ln, el) , n > 0 și 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 primită (cea care trebuie sortată)
% O: lista pentru output (lista l sortată)
% 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_liste(l))
% 
% Model de flux (i, i) - determinist, (i, o) - determinist
% Vom folosi (i, o) - determinist
main(L, LC):-
lung_liste(L, Len),
sortare(L, L1),
toateSubm(L1, 2, Len, LC).
```

- 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** și **e1=f** rezultă lista (1 f (2 d (f))).

E facuta deja

# **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 definiție de funcție 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.

Facuta deja

- B. Dându-se o listă eterogenă formată din numere și liste liniare nevide de numere, se cere un program SWI-PROLOG care să construiască o listă cu valorile minime din acele subliste având suma elementelor număr prim. Lista rezultată va conține elementele în ordine inversă față de ordinea în care sublistele apăreau în lista inițială. **De exemplu**, pentru lista [[4, 1, 18], 7, 2, -3, [6, 9, 11, 3], 4, [5, 9, 19]], rezultatul va fi [3, 1]. Nu se da

- 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**  $\Rightarrow$  [[1,2],[2,1],[1,3],[3,1],[2,3],[3,2]] (nu neapărat în această ordine)

e facut

- 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

e facuta

# 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 definiție de funcție 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 heterogenă 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(I1 ... In) = { 0, n = 0
%           { I1 + sumaLista(I2 ... In), 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(I1,...,In, k) = 1. [I1] daca k = 1
%           2. comb(I2,...,In, k)
%           3. I1 + comb(I2,...,In, 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(I, k, S) = 1. comb(I, k),daca conditie(comb(I, k),S)
%                               - adev
%
% combinariConditie(L: List, K: Integer, C: List, S:integer).
% L - lista pe care trebuie sa facem combinariile 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 submulțimilor 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] și **S=10**  $\Rightarrow$  [[1,2,3,4], [4,6]].

```
% lungime(l) = { 0, daca l=[])
% { 1 - 11 % 2 } + lungime(l2 ... ln), altfel
% lungime(L: list, Len:integer)
% L - lista pentru care trebuie aflată 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_listă(l1,...,ln) = { 0, n = 0
% { 1 + lung_listă(l2,...,ln), altfel
% lung_listă(l:list, lung:integer)
% l - lista careia îi determinăm lungimea
% model de flux - (i, o) - determinist - îl folosim
% - (i, i) - determinist
lung_listă([], 0).
lung_listă([_|T], Lung):-
lung_listă(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 - listă, lista de elemente
% K - integer, lungimea curentă a submultimii
% Lung - Lungimea listei L
% O - rezultatul
% model de flux (i,i,o,i) - determinist - îl folosim pe acesta
% (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 în care trebuie inserat elementul în 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 și el <= l1
% { l1 (+) insert(l2 ... ln, el) , n > 0 și 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 primită (cea care trebuie sortată)
% O: lista pentru output (lista L sortată)
% 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_listă(l), S)
% 
% Model de flux (i, i, o) - determinist, (i, i, i) - determinist
% Vom folosi (i, i, o) - determinist
main(L, S, LC):-
lung_listă(L, Len),
sortare(L, L1),
toateSubm(L1, 2, Len, LC, S).
```

- D. Se consideră o listă neliniară. Să se scrie o funcție care să aibă ca rezultat lista inițială în 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 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)))
```

```
    |  
    )
```

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

```
    0  
    )
```

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

```
) )
```

```
; 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 și următoarea definiție 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 (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(I1 ... In) = { 0, n = 0
%                               { I1 + sumaLista(I2 ... In), 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(I) = { 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(I1,...,In, k) = 1. [I1]           daca k = 1
%                      2. comb(I2,...,In, k)
%                      3. I1 + comb(I2,...,In, 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) - determinant

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

% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinariile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinant, (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  $b=4 \Rightarrow [[2,3],[3,4],[2,3,4]]$  (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) - 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).
```

D. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială în care toate aparițiile 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)) și **e1** este B => (1 (2 B (3 B)) (B))
- b)** dacă lista este (1 (2 (3))) și **e** este A => (1 (2 (3)))

E facut deja

# 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 definiție 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 nenumerați, se cere un program LISP care să înlocuiască fiecare atom nenumerație cu numărul de aparții 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([_1,...,_n]) = { adevarat, n = 1
%                                { fals      , n >= 2 si abs(_1-_2) % 2 = 1
%                                { verifProp(_2,...,_n), 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) - determinant

```
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)) - adevarat

```
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinariile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinant, (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 predicatelor folosite.

**Exemplu:** pentru **N=4, k=2**  $\Rightarrow [[1,3],[2,4]]$  (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(I) = { toateAranjamentele(I, 2, lungime(I))  
%  
% 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).
```

- D. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială în 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

Facuta deja

# 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 definiție de funcție 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)))
  )
)
```

## Nu primim asa ceva

- B. Dându-se o listă eterogenă formată din numere și liste liniare de numere, se cere un program SWI-PROLOG care să calculeze diferența dintre cel mai mare număr din subliste și cel mai mic număr de la nivelul superficial al listei. Se va presupune că lista de intrare conține cel puțin o sublistă și cel puțin un număr la nivel superficial, dar nu se cunoaște valoarea minimă/maximă posibilă pentru numerele din listă/subliste. **De exemplu**, pentru lista [[4, 2, 8], 7, 2, -3, [6, 9, 11, 2], 4], rezultatul va fi 14 [11 - [-3]].

**C.** Să se scrie un program PROLOG care generează lista submulțimilor 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] și **S=10**  $\Rightarrow$  [[1,2,3,4], [4,6]].

E facuta deja

- 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 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 definiție de funcție 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-o singură submulțime. De exemplu, pentru mulțimea  $[1,2,3]$  și  $k = 2$ , soluția este (nu neapărat în această ordine):  $\{[[3], [1]], [[2], [3, 1]], [[3], [2, 1]]\}$ .

```
% sumaLista(I1 ... In) = { 0, n = 0
%                                { I1 + sumaLista(I2 ... In), altfel
```

```
% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculată 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(I1,...,In) = { 0 , n = 0
%                                { 1 + nrImpare(I2,...,In) , n > 0 si I1 \% 2 = 1
%                                { nrImpare(I2,...,In) , altfel
% nrImpare(I:list, ct:integer)
% I - lista careia să încaramăm numărul de numere impare
% ct - numărul de numere impare
% model de flux - (i, o) - determinist - îl 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(I) = { fals, sumaLista(L) \% 2 = 0
% { adevarat, altfel (sumaLista(L) \% 2 = 1)
```

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

```
% comb(I1,...,In, k) = 1. [I1]           dacă k = 1
%                      2. comb(I2,...,In, k)
%                      3. I1 + comb(I2,...,In, k - 1) k > 1
```

```
% comb(L: list, K:integer, C:list)
% L - mulțimea de numere
% K - numărul de numere din combinare
% C - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o să-l folosim
% (i, i, i) - determinist
```

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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]  $\Rightarrow [[2,3],[3,4],[2,3,4]]$  (nu neapărat în această ordine)

```

comb([H|T], 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(I, k) = 1, comb(I, k),  

%      daca conditie(comb(I, k)) - adev  

% combinariConditie(L: List, K: Integer).  

% L - lista pe care trebuie să facem combinările cu condiție  

% K - numărul de elemente din combinări  

% 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  

% ( L combinariConditie(L,K) +toateSubm(L,K+1,Lung)  

% toateSubm(L, K, Lung, O)  

% L - list, lista de elemente  

% K - integer, lung curență 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(I) = { 0, daca vidal()  

% { 1 + lungime(I2 ... In), altfel  

%  

% lungime(L: list, Len)  

% L - lista pentru care trebuie afișata 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 în care trebuie inserat elementul în 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(I1 I2 ... In, el) = { [el] , n = 0  

% { el (+) I1 ... In , n > 0 și el <= I1  

% { I1 (+) insert(I2 ... In, el) , n > 0 și el > I1  

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 primită (ceea ce trebuie sortată)  

% O: lista pentru output (lista L sortată)  

% Modelul de flux: (i, i) - determinist sau (i, o) - determinist  

% Folosim (i, o) - determinist.  

%sortare(I1 I2 ... In) = { [] , n = 0  

% { insert(sortare(I2 ... In), I1), altfel (n > 0)  

sortare([], []):-!.  

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

    sortare(T, O1),  

    insert(H, O1, O).  

% main(I) = { toateAranjamentele(I, 2, lungime(I))  

%  

% 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 funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți 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 nu e atom numeric si niv % 2 = 1
;   {   | | , l nu e atom numeric si niv % 2 = 0
;   {   | , 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)
        )
      )
      )
    )
  )
; 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 și următoarea definiție 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 pentru a evita apelul repetat al functiei  $f(T, S)$ .

## Nu se da

- B. Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care verifică dacă media atomilor numerici de pe niveluri pare este egală cu media atomilor numerici de pe niveluri impare. De exemplu, pentru lista (10 A 10 V (10 B (4 H 5)) (3 (A B (J (1) 5 L)))) rezultatul va fi true.

- C. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista submulțimilor cu număr par de elemente. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu-** pentru lista L=[2,3,4]  $\Rightarrow$  [[], [2,3],[2,4],[3,4]] (nu neapărat în această ordine)

```
% comb(I1,...,In, k) = 1. [I1] daca k = 1
% 2. comb(2,...,In)
% 3. I1 + comb(2,...,In, 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|T], 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(I, k, S) = 1. comb(I, k),daca conditie(comb(I, k),S)
% - adev
%
% combinariConditie(L: List, K: Integer, C: List, S:integer).
% L - lista pe care trebuie sa facem combinariile 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
combinariConditie(L, K, C):-
    comb(L, K, C),
    % lung_list(I,...,In) = { 0, n = 0
    % { 1 + lung_list(2,...,In), altfel
    % lung_list(I:list, lung:integer)
    % I - lista careia ii determinam lungimea
    % model de flux - (i, o) - determinist - il folosim
    % - (i, i) - determinist
    lung_list([], 0).
    lung_list([_|T], Lung):-
        lung_list(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, i) - determinist.
    %insert(I1 I2 ... In, el) = { [el] , n = 0
    % { el (+) I1 ... In , n > 0 si el <= I1
    % { I1 (+) insert(I2 ... In, el) , n > 0 si el > I1

    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(I1 I2 ... In) = { [] , n = 0
    % { insert(sortare(I2 ... In), I1), altfel (n > 0)
    sortare([], []):-!.
    sortare([H|T], O):-
        sortare(T, O1),
        insert(O1, H, O).

    % main(I) = { toateSubm(I, 0, lung_list(I))
    %
    % Model de flux (i, o) - determinist, (i, i) - determinist
    % Vom folosi (i, o) - determinist
    main(L, LC):-
        lung_list(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
        )
      )
    )
  )
  (
    (AND (listp a) (equal gasit NIL) (equal e (car a)))
    (+ 1
      (apply #'max(
        mapcar #'(
          lambda (x)
          (
            inaltime_nod x e T
          )
        )
      )
    )
  )
  (
    (+ 1
      (apply #'max(
        mapcar #'(
          lambda (x)
          (
            inaltime_nod x e T
          )
        )
      )
    )
  )
  )
)
```

```
; 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 și următoarea definiție 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 aceeași parametru ca funcția f plus un parametru ce reprezintă valoarea predicatului f(T,S1).

### Nu se da

- B. Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care verifică dacă următoarele trei liste sunt egale: lista tuturor atomilor de pe niveluri multiple de 3 (3, 6, etc.), lista tuturor atomilor de pe niveluri sub forma  $3k+1$  (1, 4, 7, etc.) lista tuturor atomilor de pe niveluri sub forma  $3k+2$  (2, 5, 8, etc.). De exemplu pentru lista (A 1 (A 1(A 1(B 777 (B (B 777 C) 777 C) C) D) D) rezultatul va fi true.

- 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  $b=4 \Rightarrow [[2,3],[3,4],[2,3,4]]$  (nu neapărat în această ordine)

Facuta deja

- 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 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)
   )
  )
 )
```

```
; 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 și următoarea definiție 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([I1 ... In]) = { 0, n = 0
%                               { I1 + sumaLista(I2 ... In), 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([I]) = { fals, sumaLista([I]) \% 2 = 0
% { adevarat, altfel (sumaLista([I]) \% 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([I1,...,In], k) = 1. [I1]           daca k = 1
%                   2. comb([I2,...,In], k)
%                   3. I1 + comb([I2,...,In], 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) - determinant
```

```
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([I], k) = 1. comb([I], k),
%                           daca conditie(comb([I], k)) - adevarat
```

```
% combinariConditie([L: List, K: Integer]).
% L - lista pe care trebuie sa facem combinariile cu conditie
% K - numarul de elemente din combinari
% Model de flux: (i, i, i) - determinant, (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 curent 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 aflată 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 în care trebuie inserat elementul în 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(I1 I2 ... In, el) = { [el] , n = 0  
% { el (+) I1 ... In , n > 0 si el <= I1  
% { I1 (+) insert(I2 ... In, el) , n > 0 si el > I1  
  
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 primită (cea care trebuie sortată)  
% O: lista pentru output (lista L sortată)  
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist  
% Folosim (i, o) - determinist.  
  
%sortare(I1 I2 ... In) = { [] , n = 0  
% { insert(sortare(I2 ... In), I1), 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))) și **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)))
      |
    )
    (
      (AND (atom l) (equal niv k))
      e
    )
    (
      T
      (mapcar #'(lambda (x)
                  (substituie x (+ niv 1) k e))
              )
    )
  )
; 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 rationamentului; (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 definiție 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,Y)** 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, diferențele fiind înlaturarea incremantarii lui K și a apelului recursiv f(K1, Y).

## Nu se da

- B. Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care concatenează fiecare (sub)listă liniară în care numărul atomilor numerici este egal cu numărul atomilor nenumerici cu lista în care se află. Se repetă acest proces până lista nu mai poate fi modificată. De exemplu, pentru lista (A B (4 A 3) 11 (5 (A (B 3) (C 10) 1) (4 A ) X Y Z)5) rezultatul va fi (A B (4 A 3) 11 (5 A B 3 C 10 1 4 A X Y Z) 5).

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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]  $\Rightarrow$  [[2,3,4]]

E facuta deja

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 => ()

E facuta deja

# **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 definiție de funcție 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.

E facut deja

### Nu se cere

- B. Dându-se o listă neliniară formată din numere mai mari sau egale cu 2, se cere un program SWI-PROLOG care să înlocuiască fiecare număr neprim cu suma divizorilor săi proprii. Repetați procesul până când lista rămâne doar cu numere prime. **De exemplu**, pentru lista [10, 20, 30, 40] rezultatul va fi [7, 7, 41, 7] (lista inițială devine la început [7, 21, 41, 49], apoi [7, 10, 41, 7] iar final [7, 7, 41, 7]). Va trebui să returnați doar lista finală.

- C. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista submulțimilor cu număr par de elemente. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu-** pentru lista L=[2,3,4]  $\Rightarrow$  [[], [2,3], [2,4], [3,4]] (nu neapărat în această ordine)

Facuta deja

- 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 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)
    )
  )
)
```

```
)
```

```
; main(l) = nod_pare(l, -1)
```

```
; main(l:list)
```

```
(defun main(l)
```

```
) (nod_pare 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).
  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 și următoarea definiție 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.

Facut deja

## Nu se da

- B. Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care înlocuiește fiecare atom numeric par de pe niveluri impare cu suma cifrelor. Nivelul superficial este impar. De exemplu, pentru lista (A 2 (B 31 F (D 102 5 T (66) E) (D 10 (E R 51)) 99)) rezultatul va fi (A 2 (B 31 F (D 3 5 T (66) E) (D 1 (E R 51)) 99)).

- 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**  $\Rightarrow$  [[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) - determinant
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 combinariile cu conditie
% K - numarul de elemente din combinari
% C - output value
% S - suma pe care o verificam
% Model de flux: (i, i, i) - determinant, (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) - determinant, (i, i, i) - determinant
% Vom folosi (i, i, o) - determinant
main(L, K, LC):-
    findall(O1, combinariConditie(L, K, O1), LC).
```

- D. Se consideră o listă neliniară. Să se scrie o funcție care să aibă ca rezultat lista inițială în 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 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)))
    |
  )
  (T
    (mapcar #'(lambda (x)
      (
        inlocuire x (+ niv 1)
      )
    )
    )
  )
)

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

  (inlocuire 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 următoarea definiție de funcție 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.

Facut

### **Nu se da**

- B. Dându-se 2 liste formate din numere întregi și subliste de numere întregi, se cere un program SWI-Prolog care returnează o listă care conține, pentru fiecare pereche posibilă de subliste (o sublistă din prima listă și una din a doua), produsul elementelor maxime. De exemplu, pentru următoarele 2 subliste [1,2, [4,2], 6, [3,2]] și [1,2,3,[5,6],8, 5,[12,3], 4,1,[3,8]] rezultatul va fi (nu neapărat în această ordine): [24, 48, 32, 18, 36, 24].

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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 predicatelor folosite.

**Exemplu:** pentru **a=2** și **b=4** ⇒ [[2,3,4]]

```
% pare(I) = { 0, daca I=0
%           { (1 - I \% 2) + pare(I2 ... In), altfel
% pare(L: list, Len:integer)
% L - lista pentru care trebuie aflată 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(I) = { 0, daca I=0
%               { (1 \% 2 + impare(I2 ... In), altfel
% impare(L: list, Len:integer)
% L - lista pentru care trebuie aflată 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(I) = { fals, pare(L) \% 2 = 1 sau impare(L) \% 2 = 1
%                 { adevarat, altfel(pare(L) \% 2 = 0 si impare(L) \% 2 = 1)
% conditie(L: list)
% L - lista pentru care trebuie verificată condiția
% Model de flux: (i) - determinist.
conditie(L):-
    pare(L, P),
    P mod 2 =:= 0,
    impare(L, I),
    I mod 2 =:= 1.

% comb(I1,...,In, k) = 1. [|] dacă k = 1
%                   2. comb(I2,...,In, k)
%                   3. I1 + comb(I2,...,In, k - 1) dacă k > 1
% comb(L: list, K:integer, C:list)
% L - mulțimea de numere
% K - numărul de numere din combinare
% C - lista rezultat
% (i, i, o) - nedeterminist - pe acesta o să-l folosim
% (i, i, i) - determinist
comb([_|_], 1, [H]).
comb([_|T], K, C):-
    comb(T, K, C),
    comb([_|T], K1, C).
comb([H|T], K, [H|C]):-
    K > 1,
    K1 is K - 1,
    comb(T, K1, C).

% combinariConditie(I, k) = 1. comb(I, k),
%                           dacă conditie(comb(I, k)) - adevarat
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie să facem combinările cu condiție
% K - numărul 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 - capătul inferior al intervalului
% b - capătul superior al intervalului
% o - intervalul [a,b]
genereazaLista(A, B, []):-!
genereazaLista(A, B, [A]):-
    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 - listă de elemente
% K - integer, lung curență a submulțimii
% 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)
          )
        )
      )
    )
  )
; main(l) = inlocuire(l, 0)
; main(l:list)
(defun main(l)
  (inlocuire 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 și următoarea definiție 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.

Facut

- B.** Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care verifică dacă secvența atomilor numerici de pe toate nivelurile impare formează o secvență zig-zag (elementul 2 e mai mare decât primul element, elementul 3 este mai mic decât elementul 2, elementul 4 este mai mare decât elementul 3, etc.). De exemplu, pentru lista (10 21 (3 A (B (0 77) 1 77)) C (5 (D 54) 11 6) 89 F H) rezultatul va fi true (secvența zig-zag este (10 21 1 77 54 89)).

- C. Dându-se o listă formată din numere întregi, să se genereze lista submulțimilor 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] și k=3 ⇒ [[1,2,3],[1,5,9],[1,3,5]] (nu neapărat în această ordine)

```
% progresie([I1|I2|I3,...,In]) = { adevarat , n = 2
%                                { fals , n > 2 si abs(I1-I2)!=abs(I2-I3)
%                                { progresie(I2,...,In) , altfel
% progresie(I: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(I) = { 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(I1,...,In, k) = 1. [I1] daca k = 1
%           2. comb(I2,...,In, k)
%           3. I1 + comb(I2,...,In, 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 combinariile 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(I1 I2 ... In, el) = { [el] , n = 0
%                            { el (+) I1 ... In , n > 0 si el <= I1
%                            { I1 (+) insert(I2 ... In, el) , n > 0 si el > I1  

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(I1 I2 ... In) = { [] , n = 0
%                        { insert(sortare(I2 ... In), I1), 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 funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți atomii numerici multipli de 3. **Se va folosi o funcție MAP.**

**Exemplu**

- a) dacă lista este  $(1\ (2\ A\ (3\ A))\ (6)) \Rightarrow (1\ (2\ A\ (A))\ NIL)$   
 b) dacă lista este  $(1\ (2\ (C))) \Rightarrow (1\ (2\ (C)))$

```
; 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))
)
```

# **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 definiție de funcție 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.

Facut

## Nu se da

- B. Dându-se o listă liniară formată din numere, se cere un program SWI-PROLOG care să furnizeze lista în care fiecare număr care este mai mic decât succesorul său din listă este înmulțit cu 2. Repetați această operație până când nu mai sunt posibile modificări în listă. **De exemplu**, pentru lista [1, 2, 3] rezultatul va fi [8, 16, 3].

- 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  $\Rightarrow$  [[2,5], [4,3]] (nu neapărat în această ordine)

```
% sumaLista([1 ... ln]) = { 0, n = 0
%                           { 1 + sumaLista([2 ... ln]), altfel

% sumaLista(L: List, S: Integer).
% L - lista pentru care trebuie calculată 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([1...ln]) = { adevarat      , n = 1 și l1 \% 2 = 1
%                   { fals          , n = 2 și l1 \% 2 = 0
%                   { verif([2,...ln]), altfel
% verif([_|list])
% l - lista pe care o verificăm dacă 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 și verif(L)=adev)

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

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

% comb(L: list, K:integer, C:list)
% L - mulțimea 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(combi(l, k)) - adev

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

% inserarea(e, l1,...,ln) = 1. e + l1l2...ln
%                           2. l1 + inserarea(e, l2...ln)

% inserarea(E: element, L:List, LRez:list)
% E - elementul pe care dorim să-l inserăm pe toate pozițiile
% L - lista în care o să fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o sa-l folosim
% (i, i, i) - determinist
inserarea(E, L, [E|L]).
inserarea(E, [H|T], [H|Rez]):-
    inserarea(E, T, Rez).

% perm([1,...,ln]) = 1. []
%                           2. inserarea([l1, permutari([2,...,ln])])
% perm(L:list, LRez:list)
% (i, o) - nedeterminist
% (i, i) - determinist
permutari([_|T], P):-
    permutari([E|T], P),
    permutari(T, L),
    inserarea(E, L, P).

% aranjamente([1,...,ln], k, SUMA) = 1. l, l=permutari(combinari(L,K)),
%                                         daca conditie(l,SUMA) = true
%                                         %
%                                         2. aranjamente([2,...,ln], k, SUMA)
%                                         daca conditie([2,...,ln], SUMA) = true

% aranjamente(L:list, K:element, O:list, SUMA:integer)
% L - mulțimea numerelor
% K - nr de elemente din aranjament
% O - aranjamentul curent
% SUMA - suma cu care comparăm suma permutării
% 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).
```

- D. Se consideră o listă neliniară. Să se scrie o funcție care să aibă ca rezultat lista inițială în 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)))

facuta

# 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 definiție de funcție 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.

### Nu se da

- B. Dându-se o listă liniară de numere, se cere un program SWI-Prolog care aplică o sortare stabilă pe această listă și ordonează elementele crescător pe baza restului împărțirii la 3. De exemplu, pentru lista [10, 5, 6, 12, 7, 3, 20, 30] rezultatul va fi [6, 12, 3, 30, 10, 7, 5, 20]. (Obs: sortare stabilă înseamnă că elementele *egale* vor rămâne în aceeași ordine în care erau în lista inițială, de exemplu 6 și 12).

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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]  $\Rightarrow$  [[], [2], [4], [2, 4]] (nu neapărat în această ordine)

```
% sumaLista([1 ... ln]) = { 0, n = 0
% { l1 + sumaLista([2 ... 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(1,...,ln, k) = 1. [l1] daca k = 1
% 2. comb([2,...,ln, 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([_|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 combinariile 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([2,...,ln]), altfel
% lung_lista(l: list, lung: integer)
% l - lista care trebuie determinata lungimea
% model de flux - (i, o) - determinist - il folosim
% (i, 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 curentă a submultimi
% Lung - lungimea listei L
% O - rezultat
% 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(I1 I2 ... ln, el) = { [el] , n = 0
% { el (+) I1 ... ln , n > 0 si el <= I1
% { el (+) insert(I2 ... ln, el) , n > 0 si el > I1
%insert([], E, [E]):-
%insert([_|T], E, O):-
%    E =< H,
%    !,
%    O = [E, H|T].
%insert([_|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(I1 I2 ... ln) = { [], n = 0
% { insert(sortare(I2 ... ln), I1), altfel (n > 0)
% sortare([], []):-.
sortare([_|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).
```

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

Facut

# 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 definiție de funcție 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.

- B. Dându-se o listă liniară de numere, se cere un program SWI-PROLOG care returnează (într-o listă de perechi) toate posibilele partiții ale listei inițiale în două subliste, astfel încât toate elementele sublistelor să fie prime între ele (toate elementele primei subliste sunt prime între ele și toate elementele celei de-a doua subliste sunt prime între ele). Pentru a evita generarea aceleiași partiționări de două ori (ex: [A, B] și [B, A]), prima sublistă va contine cel mult același număr de elemente ca cea de-a doua sublistă. **De exemplu**, pentru lista [3, 5, 7, 9], rezultatul va fi (nu neapărat în această ordine): [[[5, 3], [9, 7]], [[7, 3], [9, 5]], [[3], [9, 7, 5]], [[9, 5], [7, 3]], [[9, 7], [5, 3]], [[9], [7, 5, 3]]].

nu se da

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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]  $\Rightarrow$  [[2,3],[3,4],[2,3,4]] (nu neapărat în această ordine)

Facut

- 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))).

facut

# **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 definiție de funcție 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.

facut

- B. Dându-se o listă liniară de numere, se cere un program SWI-Prolog care înlocuiește secvențele de numere egale cu suma secvenței. Acest proces trebuie repetat până când nu mai sunt elemente consecutive egale în listă. De exemplu, pentru lista [1, 2 , 1, 1, 4, 5, 6, 7, 7, 3, 3, 3, 3, 3, 3, 10], rezultatul va fi [1, 8, 5, 6, 42, 10].

nu se da

- 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**  $\Rightarrow$  [[6,4],[5,3]] (nu neapărat în această ordine)

facut

- D. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți atomii numerici multipli de 3. **Se va folosi o funcție MAP.**

**Exemplu**

- a) dacă lista este  $(1\ (2\ A\ (3\ A))\ (6)) \Rightarrow (1\ (2\ A\ (A))\ NIL)$   
b) dacă lista este  $(1\ (2\ (C))) \Rightarrow (1\ (2\ (C)))$

facut

# 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 definiție de funcție 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.

facut\

- B. Dându-se o listă liniară de numere pozitive, scrieți un program SWI-PROLOG care returnează (sub forma unei liste de perechi) toate posibilele partiții ale listei inițiale în două subliste, astfel încât suma cifrelor elementelor din cele două subliste este egală (se presupune că există cel puțin o astfel de partitionare a listei inițiale). **De exemplu**, pentru lista [28, 21, 52, 34, 7], rezultatul ar trebui să fie (nu neapărat în această ordine): [[[52, 28], [7, 34, 21]], [[34, 28], [7, 52, 21]], [[7, 28], [34, 52, 21]], [[34, 52, 21], [7, 28]], [[7, 52, 21], [34, 28]], [[7, 34, 21], [52, 28]]].

nu

- C. Dându-se o listă formată din numere întregi, să se genereze lista submulțimilor cu **k** elemente în progresie aritmetică. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu-** pentru lista  $L=[1,5,2,9,3]$  și  $k=3 \Rightarrow [[1,2,3],[1,5,9],[1,3,5]]$  (nu neapărat în această ordine)

facut

- 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)
    |
  )
  (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)
)
```

# **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 definiție de predicat PROLOG **f(integer, integer)**, având modelul de flux (i, o):
- f(1, 1):-!.  
f(K,X):-K1 is K-1, **f(K1,Y)**, Y>1, !, K2 is K1-1, X is K2.  
f(K,X):-K1 is K-1, **f(K1,Y)**, Y>0.5, !, X is Y.  
f(K,X):-K1 is K-1, **f(K1,Y)**, X is Y-1.

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

rez

- B. Dându-se o listă neliniară conținând atât atomi numerici, cât și nenumerici, se cere un program LISP care să returneze lista din care să se eliminate atomii nenumerici din 3 în 3 (numărarea se va face de la stânga spre dreapta, considerând toate elementele). Lista rezultată va păstra structura listei initiale. **De exemplu**, pentru lista (A B 12 (5 D (A F (10 B) D (5 F) 1)) C 9 (F 4 (D) 9 (F (H 7) K) (P 4)) X) rezultatul va fi lista (A B 12 (5 (A F (10) D (5 F) 1)) 9 (F 4 (D) 9 ((H 7) K) (4)) X).

nu

- C. Scrieți un program PROLOG care determină dintr-o listă formată din numere întregi lista subșirurilor cu cel puțin 2 elemente, formate din elemente în ordine strict crescătoare. Se vor scrie modelele matematice și modelele de flux pentru predicatelor folosite.

**Exemplu-** pentru lista [1, 8, 6, 4]  $\Rightarrow$  [[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(I1,...,In, k) = 1. [I1] daca k = 1
% 2. comb([2,...,In, k)
% 3. I1 + comb([2,...,In, 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([T, K, C].
comb([H|T], K, [H|C]):-
K > 1,
K1 is K - 1,
comb(T, K1, C).

% lung_list(I1,...,In) = { 0, n = 0
% { 1 + lung_list(I2,...,In), altfel
% lung_list(I:list, lung:integer)
% I - lista careia ii determinam lungimea
% model de flux - (i, o) - determinist - il folosim
% - (i, i) - determinist
lung_list([], 0).
lung_list([_|T], Lung):-
lung_list(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) - 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(I1 I2 ... In, el) = { [el] , n = 0
% { el (+) I1 ... In , n > 0 si el <= I1
% { I1 (+) insert(I2 ... In, el) , n > 0 si el > I1

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(I1 I2 ... In) = { [] , n = 0
% { insert(sortare(I2 ... In), I1), altfel (n > 0)
sortare([], []):!.

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

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

- D. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți atomii numerici pari situati pe un nivel impar. Nivelul superficial se consideră a fi 1. **Se va folosi o funcție MAP.**

**Exemplu**

- a) dacă lista este  $(1 \ (2 \ A \ (4 \ A)) \ (6)) \Rightarrow (1 \ (2 \ A \ (A)) \ (6))$
- b) dacă lista este  $(1 \ (2 \ (C))) \Rightarrow (1 \ (2 \ (C)))$

```

; 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)
      )
    )
    )
  )

; main(l) = elimina(l,0)[1]
; main(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 următoarea definiție 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.

facut

- B. Dându-se o listă neliniară conținând atât atomi numerici, cât și nenumerici, se cere un program LISP care să calculeze numărul total de atomi nenumerici la nivel superficial din acele subliste (incluzând și lista originală) al căror prim atom numeric (la orice nivel) este număr par. **De exemplu**, pentru lista (A B 12 (5 D (A F (10 B) D (5 F) 1)) C 9) rezultatul va fi 7.

nu

- 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**  $\Rightarrow [[1,3],[2,4]]$  (nu neapărat în această ordine)

facut

- 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  $(1 \ s\ 4 \ (4 \ f\ (7)))$ .

facut

# **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 definiție 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.

facut

- B. Dându-se o listă neliniară conținând atât atomi numerici, cât și nenumerici, se cere un program LISP care să construiască o listă care să conțină pentru nivelurile pare cel mai mare atom numeric iar pentru nivelurile impare cel mai mic atom numeric (se presupune că fiecare nivel al listei conține cel puțin un atom numeric), dar în ordine inversă (astfel minimul de pe nivelul 1 este ultimul element, minimul de pe nivelul 2 este penultimul element, etc). **De exemplu**, pentru lista (A B 12 (5 D (A F (10 B) D (5 F) 1)) C 9 (F 4 (D) 9 (F (H 7) K) (P 4)) X) rezultatul va fi (10 1 9 9). Nu este permisă folosirea funcției *reverse* din Lisp.

nu

- C. Dându-se o listă formată din numere întregi, să se genereze lista submulțimilor cu **k** elemente numere impare, în progresie aritmetică. Se vor scrie modelele matematice și modelele de flux pentru predicale folosite.

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

```
% progresie([1|2|3,...,In]) = { adevarat , n = 2
%                                { fals , n > 2 si abs(1-I2)|=abs(I2-I3)
%                                { progresie([2,...,In]) , altfel
% progresie(I: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([1...n]) = { adevarat , n = 0
%                      { fals , n > 0 si I1 \% 2 = 0
%                      { impare([2...n]) , altfel
% impare(I:list)
% I - lista pe care o verificam
% model de flux - (i)
impare([]):-!.
impare([H|T])-:
    H mod 2 ==:= 1,
    impare(T).

% conditie(I) = { 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(I1,...,In, k) = 1. [I1] daca k = 1
%                      2. comb(I2,...,In, k)
%                      3. I1 + comb([2,...,In, 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|T], K, C):-
    comb(T, K, C).
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(I, k) = 1. comb(I, k),
%                           daca conditie(comb(I, k)) - adev
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie sa facem combinariile 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([I1 I2 ... In, el]) = { [el] , n = 0
%                           { el (+) I1 ... In , n > 0 si el <= I1
%                           { I1 (+) insert([I2 ... In, el]) , n > 0 si el > I1
%
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 (ceea ce trebuie sortata)
% O: lista pentru output (lista L sortata)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.
%sortare([I1 I2 ... In]) = { [] , n = 0
%                           { insert(sortare([I2 ... In], I1)), 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 funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți 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 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
    )
  )
  ))
```

```
) ) }
```

; 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 definiție de funcție 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.

- B. Pentru 2 liste care reprezintă numere cifră cu cifră, definim operația de *rearanjare* în modul următor: comparăm cifrele și formăm 2 liste noi, una conținând cifrele mai mari și una cu cifrele mai mici. De exemplu, rearanjarea listelor [1,4,2,7] și [8,2,9,6,1] va returna [8, 2, 9, 6 ,7] și [1, 4, 2, 1]. Dându-se 2 liste care conțin numere și subliste cu cifre, se cere un program SWI-Prolog care returnează 2 liste cu subliste, ele conținând rezultatul după rearanjarea sublistelor din cele 2 liste. Una dintre liste conține sublistele cu cifrele mai mari și cealaltă conține sublistele cu cifrele mai mici. Cele două liste de intrare au același număr de subliste, dar nu neapărat pe aceeași pozitii. De exemplu, pentru listele [1, 2, [6, 2, 4], 6, [9, 9, 1, 1], 17, 9, [5, 3, 8, 1, 9]] și [1, 2, 3, [1, 5], 7, 11, [8, 3], 7, 5, [9, 4, 2, 5], 77] rezultatul va fi [[6, 2, 5], [9, 9, 8, 3], [5, 9, 8, 2, 9]] și [[1, 4], [1, 1], [3, 4, 1, 5]].

nu

- C. Pentru o valoare **N** dată, să se genereze lista permutărilor cu elementele  $N, N+1, \dots, 2^N-1$  având proprietatea că valoarea absolută a diferenței dintre două valori consecutive din permutare este  $\leq 2$ . 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, I1,...,In) = 1. e + I1I2...In
%           2. I1 + insereaza(e, I2...In)
%
% 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(I1,...,In) = 1. []
%           2. insereaza(I1, permutari(I2,...,In))
%
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([E|T], P) :-
    permutari(T, L),
    insereaza(E, L, P).

% conditie(I1,...,In) = { adevarat , n < 2
%           { fals , n >= 2 si abs(I1-I2)>2
%           { conditie(I2,...,In), n >= 2 si abs(I1-I2)<=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 noastră
% 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. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți 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)))

facut

# **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 definiție de predicat PROLOG **f(integer, integer)**, având modelul de flux (i, o):

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

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

facut

- B. Dându-se o listă neliniară conținând atât atomi numerici cât și nenumerici, se cere un program LISP care construiește o listă cu elementele listei initiale, din k în k (numărarea se face de la stânga la dreapta, considerând toate elementele), în ordine inversă. **De exemplu**, pentru lista (A B 12 (5 D (A F (10 B) D (5 F) 1)) C 9) și k = 3 rezultatul este (9 F B A 12). Nu este permisă utilizarea funcției predefinite *reverse* din Lisp.

nu

- 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  $N=1 \Rightarrow [[3],[2,4],[2,3,4]]$  (nu neapărat în această ordine)

f

- 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))) și **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)))
    )
  (
    (AND (atom l) (equal 1 (mod niv 2)))
    e
    )
  (
    T
    (mapcar #'(lambda (x)
                (substituie x (+ niv 1) e)
                )
    )
  )
)
```

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

```
(defun main(l e)
```

```
  (substituie l -1 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 definiție 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.
```

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

**f**

- B. Dându-se o listă neliniară care conține atomi numerici și nenumeric, se cere un program Lisp, care construiește o listă care are aceeași structură ca lista inițială, dar fiecare (sub)listă conține un singur element pe nivel superficial, și acel element este diferența dintre valoarea maximă și cea minimă de pe nivelul superficial al (sub)listei corespunzătoare în lista inițială. Este garantat că fiecare (sub)listă conține măcar un atom numeric. De exemplu, pentru lista (F A 12 13 (B 11 (A D 15) C 3 C (1 F 6) 1) 18 11 D (A 7 F 9) F) rezultatul va fi (7 (10 (0) (5)) (2)) (7 este diferența dintre 18 și 11, 10 este diferența dintre 11 și 1, 0 este diferența dintre 15 și 15, etc.).

nu

- 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** și **S=9**  $\Rightarrow$  [[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 calculată suma
% S - suma listei date
% Model de flux: (l, 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.
%
% condiție(l,SUMA) = { fals, sumaLista(L) = SUMA
%                       { adevarat, altfel (sumaLista(L)=SUMA)
% condiție(L: list,SUMA:integer).
% L - lista pentru care trebuie verificată condiția
% SUMA - suma cu care comparați 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 - mulțimea de numere
% K - numarul de numere din combinare
% C - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o să-l folosim
% (i, i, i) - determinist
comb([H|L], 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
%                           2. l1 + combinariConditie(l2,...,ln, k)
%
% combinariConditie(L: List, K: Integer).
% L - lista pe care trebuie să facem combinările cu condiție
% K - numarul de elemente din combinări
% Model de flux: (i, i, i) - determinist - îl folosim pe asta
% (i, i, o) - nedeterminist
combinariConditie(L, K, C):-
combinariConditie(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 să-l inserăm pe toate pozițiile
% L - lista în care să o să fie inserat elementul E
% LRez - lista rezultat
%
% (i, i, o) - nedeterminist - pe acesta o să-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. []
%                           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. l1=permutari(combinari(L,K)),
%                                         daca conditie(l, SUMA) = true
%                                         2. l1 + aranjamente(l2,...,ln, k, SUMA)
%
% aranjamente(L:list, K:element, O:list, SUMA:integer)
% L - mulțimea numerelor
% K - nr de elemente din aranjament
% O - aranjamentul curent
% SUMA - suma cu care comparați suma permutării
% 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).
```

- D. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți 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

f

# 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 definiție 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)**.

- B. Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care construiește o listă care conține doar atomii numerici, alternativ un număr par urmat de unul impar. Numerele impare sunt în câte o sublistă. Numerele pare și impare sunt în aceeași ordine relativă ca în lista inițială. Se garantează că lista inițială conține un număr egal de numere impare și pare. De exemplu, pentru lista (A B (4 A 2 ) 11 (5 (A (B 20) C 10) (1(2(3(4)5)6)7 7) X Y Z)) rezultatul va fi (4 (11) 2 (5) 20 (1) 10 (3) 2 (5) 4 (7) 6 (7)).

nu

- C. Să se scrie un program PROLOG care generează lista submulțimilor de sumă pară, cu elementele unei liste. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu**- pentru lista  $L=[2, 3, 4] \Rightarrow [[], [2], [4], [2,4]]$  (nu neapărat în această ordine)

f

- 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)
  |
)
(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 definiție de funcție 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.

- B.** Dându-se o listă eterogenă formată din numere și liste nevide de numere, se cere un program SWI-Prolog care verifică dacă toate numerele (inclusiv cele din subliste) formează o secvență de numere crescătoare. De exemplu, pentru lista, [2,4,6, [10, 12, 19], 30, 201, [1000, 1003, 1006, 2003], 2020] rezultatul va fi adevărat, iar pentru lista [2,4,6, [10, 12, 11], 30, 201, [1000, 1003, 1006, 2003], 2020] rezultatul va fi fals.

nu

- C. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista permutărilor având proprietatea că valoarea absolută a diferenței dintre două valori consecutive din permutare este  $<=3$ . Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu:** pentru lista  $L=[2,7,5] \Rightarrow [[2,5,7], [7,5,2]]$  (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).
```

- D. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminați toți atomii numerici pari situati pe un nivel impar. Nivelul superficial se consideră a fi 1. **Se va folosi o funcție MAP.**

**Exemplu**

- a) dacă lista este  $(1\ (2\ A\ (4\ A))\ (6)) \Rightarrow (1\ (2\ A\ (A))\ (6))$   
b) dacă lista este  $(1\ (2\ (C))) \Rightarrow (1\ (2\ (C)))$

f

# 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 definiție de funcție 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.

- B. Dându-se o listă liniară de numere, scrieți un program SWI-PROLOG care returnează (sub forma unei liste de perechi) toate posibilele partiții ale listei inițiale în două subliste, astfel încât media valorilor elementelor din prima sublistă este mai mică sau egală cu cel mai mare divizor comun al elementelor din a doua sublistă. **De exemplu**, pentru lista [8, 4, 6, 1], rezultatul va fi (nu neapărat în această ordine): [[[1, 4, 8], [6]], [[1, 6, 4], [8]], [[1, 6], [4, 8]], [[1], [6, 4, 8]]].

nu

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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  $b=4 \Rightarrow [[2,3,4]]$

f

- 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))) și **e=h**

- a)  $k=2 \Rightarrow (a (b (h)) (c (h (e)) (h))))$
- b)  $k=4 \Rightarrow (a (b (g)) (c (d (e)) (f)))$

f

# 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 definiție de funcție LISP

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

Am folosit o lambda.

- B.** Dându-se o listă formată doar din subliste care conțin cifre pozitive, se cere un program SWI-Prolog care calculează cel mai mare număr par care poate fi format alegând câte o cifră din fiecare sublistă. Cifrele în numărul rezultat trebuie să fie în aceeași ordine în care erau sublistele de unde provin. Fiecare sublistă va conține minimum o cifră pară. De exemplu, pentru lista [[2,5,1,9], [7,2,1], [9,4,6,5], [2,6,0,7]] rezultatul va fi 9796.

- C. Dându-se o listă formată din numere întregi, să se genereze lista submulțimilor cu **k** elemente numere impare, în progresie aritmetică. Se vor scrie modelele matematice și modelele de flux pentru predicale folosite.

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

```
% progresie([I1|I2|I3,...,In]) = { adevarat , n = 2
%                                { fals , n > 2 si abs(I1-I2)!=abs(I2-I3)
%                                { progresie([I2,...,In]) , altfel
% progresie(I: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([I1...In]) = {   adevarat , n = 0
%                      {   fals , n >= 1 si I1 \% 2 = 0
%                      {   impare([I2...In]) , altfel
% impare(I:list)
% I - lista pe care o verificam
% model de flux - (i)
impare([]):-.
impare([H|T]):-
    H mod 2 =:= 1,
    impare(T).

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

% comb([I1,...,In], K) = 1. [I1] daca k = 1
%                           2. comb([I2,...,In], K)
%                           3. I1 + comb([I2,...,In], 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(I, K) = 1. comb(I, K),
%                           daca conditie(comb(I, K)) - adev
% combinariConditie(L: List, K: Integer).
% L - lista printr care trebuie sa facem combinariile 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, O):-
    comb(I, K, C),
    conditie(C).

% insert(L: list, E: int, O: list)
% L: lista în care trebuie inserat elementul în 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([I1|I2 ... In], el) = { [el] , n = 0
%                           { el (+) I1 ... In , n > 0 si el <= I1
%                           { I1 (+) insert([I2 ... In, el]) , n > 0 si el > I1

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 primă (cea care trebuie sortată)
% O: lista pentru output (lista L sortată)
% Modelul de flux: (i, i) - determinist sau (i, o) - determinist
% Folosim (i, o) - determinist.

%sortare([I1|I2 ... In]) = { [] , n = 0
%                           { insert(sortare([I2 ... In], I1), 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)
    )
  )
)
)
)
)
```

```
; 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 următoarea definiție de funcție 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.

f

- B. Dându-se o listă eterogenă formată din numere și liste liniare nevide de numere, se cere un program SWI-PROLOG care să calculeze diferența dintre cel mai mic maxim din subliste și cel mai mare dintre valorile minime din subliste. Se presupune că lista de intrare conține cel puțin o sublistă. **De exemplu**, pentru lista [[4, 2, 18], 7, 2, -3, [6, 9, 11, 3], 4, [5, 9, 19]] rezultatul va fi 6.

nu

- 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  $\Rightarrow$  [[2,5], [4,3]] (nu neapărat în această ordine)

f

- 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))) și **e=h => (a (h (g)) (h (d (h)) (h)))**

f

# 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 și următoarea definiție 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 pentru a evita apelul repetat al functiei **f(T, S)**.

- B. Dându-se o listă neliniară conținând atât atomi numerici cât și nenumerici, să se scrie un program LISP care calculează cel mai mare divizor comun al acelor numere care se află între 2 atomi nenumerici (vecinii unui număr se consideră la orice nivel). **De exemplu**, pentru lista (A B 12 (5 D (A F (15 B) D (5 F) 4)) C 9) numerele al căror cel mai mare divizor comun trebuie calculat sunt: 15 (între F și B), 5 (între D și F) și 4 (între F și C), iar rezultatul va fi 60. Nu este permisă utilizarea funcției predefinite *gcd* din Lisp.

- C. Scrieți un program PROLOG care determină dintr-o listă formată din numere întregi lista subșirurilor 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]  $\Rightarrow$  [[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)

D. Se consideră o listă neliniară. Să se scrie o funcție care să aibă ca rezultat lista inițială în 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

# **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 definiție de funcție 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.

f

- B.** Dându-se 2 liste formate din numere întregi și subliste de numere întregi, se cere un program SWI-Prolog care returnează o listă care conține toate sublistele care rezultă din concatenarea a câte o sublistă din fiecare listă. De exemplu, pentru următoarele 2 liste [1,2, [4,2], 6, [3,2]] și [1,2,3,[5,6],8, 5,[2,3], 4,1,[3,3]] rezultatul va fi (nu neapărat în această ordine): [[4,2,5,6], [4,2,2,3], [4,2,3,3], [3,2,5,6], [3,2,2,3], [3,2,3,3]].

nu

- 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** și **S=9**  $\Rightarrow$  [[6,3],[3,6],[5,4],[4,5]] (nu neapărat în această ordine)

f

- D. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminate toate aparițiile unui element **e**. **Se va folosi o funcție MAP.**

**Exemplu**

- a) dacă lista este (1 (2 A (3 A)) (A)) și **e** este A => (1 (2 (3)) NIL)
- b) dacă lista este (1 (2 (3))) și **e** este A => (1 (2 (3)))

```
; elimina(l, e) = {           | , 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))
  )
```

# **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 definiție de funcție 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.

f

- B. Dându-se o listă liniară de numere, să se scrie un program SWI-PROLOG care construiește o listă de liste astfel: primul element este lista inițială, iar ulterior, fiecare element este reprezentat de lista precedentă, în care secvențele crescătoare de numere au fost inverseate. Ultimul element al listei va fi lista în care toate elementele sunt ordonate descrescător. **De exemplu**, pentru lista [1,3,6,5,2] rezultatul va fi: [[1, 3, 6, 5, 2], [6, 3, 1, 5, 2], [6, 3, 5, 1, 2], [6, 5, 3, 2, 1]].

nu

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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 predicatelor folosite.

**Exemplu:** pentru lista L=[1, 3, 4, 2] și N=2  $\Rightarrow$  [[1,3], [2,4]]

```
% sumaLista([1 ... 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([_|T], 1, [H]).  

comb([_|T], K, C):-
    comb(T, K1, C),
    comb(T, K1, 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 combinari 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 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([1 l2 ... ln, el]) = { [el] , n = 0
%                           { el (+) l1 ... ln , n > 0 si el <= l1
%                           { l1 (+) insert([2 ... 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([1 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. Se consideră o listă neliniară. Să se scrie o funcție LISP care să aibă ca rezultat lista inițială din care au fost eliminate toate aparițiile unui element **e**. **Se va folosi o funcție MAP.**

**Exemplu**

- a) dacă lista este (1 (2 A (3 A)) (A)) și **e** este A => (1 (2 (3)) NIL)  
b) dacă lista este (1 (2 (3))) și **e** este A => (1 (2 (3)))

f

# **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 și următoarea definiție 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**

- B. Dându-se o listă neliniară conținând atât atomi numerici cât și nenumerici, să se scrie un program LISP care construiește o listă conținând ca subliste atomii nenumerici de pe fiecare nivel al listei inițiale (prima sublistă a rezultatului conține atomii nenumerici de la primul nivel, a doua sublistă atomii nenumerici de la nivelul al doilea etc.). De exemplu, pentru lista (A B 12 (5 D (A F (10 B) D (5 F) 1)) C 9 (F 4 (D) 9 (F (H 7) K) (P 4)) X) rezultatul va fi ((A B C X) (D F) (A F D D F K P) (B F H)).

nu

- C. Să se scrie un program PROLOG care generează lista submulțimilor 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  $L=[1, 3, 4, 2]$  și  $N=2 \Rightarrow [[1,3], [2,4]]$

f

- 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
   )
 )
 )
```

```
; 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 și următoarea definiție 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

- B. Dându-se o listă neliniară conținând atât atomi numerici, cât și nenumerici, se cere un program LISP care să verifice dacă atomii numerici din listă formează o secvență de valori în ordine crescătoare. **De exemplu**, pentru lista I(A B 1 (2 C D) 3 4 (F T 6 10 (A E D) (34) F) 111)) rezultatul este **adevărat** (T), iar pentru lista (A B 1 (2 C D) 3 4 (F T 6 1 (A E D) (34) F) 111)) rezultatul este **fals** (NIL).

nu

- C. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista permutărilor având proprietatea că valoarea absolută a diferenței dintre două valori consecutive din permutare este  $\leq 3$ . Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu-** pentru lista  $L=[2,7,5] \Rightarrow [[2,5,7], [7,5,2]]$  (nu neapărat în această ordine)

f

- 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 ()

f

# **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 definiție de funcție 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.

### **Sistemdeki Tüsparışları (DEFLIN E(N)**

(DEFUN F(N)  
 (cond

(cont)

```

( (= N 1)
  1
)
)

(t
  ((lambda (x)
    (cond
      (
        (> X 2)
        (- N 2)
      )
      (
        (> X 1)
        X
      )
    )
    (t
      (- X 1)
    )
  )
  (F (- N 1)))
)
)

```

) Am folosit o lambda

- B.** Dându-se 2 liste formate din numere întregi și subliste de numere întregi, se cere un program SWI-Prolog care returnează o listă care conține, pentru fiecare pereche posibilă de subliste (o sublistă din prima listă și una din a doua), produsul elementelor maxime. De exemplu, pentru următoarele 2 subliste [1,2, [4,2], 6, [3,2]] și [1,2,3,[5,6],8, 5,[12,3], 4,1,[3,8]] rezultatul va fi (nu neapărat în această ordine): [24, 48, 32, 18, 36, 24].

nu

- C. Să se scrie un program PROLOG care generează lista submulțimilor de sumă pară, cu elementele unei liste. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu**- pentru lista  $L=[2, 3, 4] \Rightarrow [[], [2], [4], [2,4]]$  (nu neapărat în această ordine)

f

- 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))).

f

# 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 definiție 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.

- B. Dându-se o listă neliniară conținând atât atomi numerici cât și nenumerici, se cere un program LISP care construiește o listă cu elementele listei initiale, din k în k (numărarea se face de la stânga la dreapta, considerând toate elementele), în ordine inversă. **De exemplu**, pentru lista (A B 12 (5 D (A F (10 B) D (5 F) 1)) C 9) și k = 3 rezultatul este (9 F B A 12). Nu este permisă utilizarea funcției predefinite *reverse* din Lisp.

nu

- 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** și **S=9**  $\Rightarrow$  [[6,3],[3,6],[5,4],[4,5]] (nu neapărat în această ordine)

f

- 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)

f

# 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 definiție de funcție 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.

- 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-o singură submulțime. De exemplu, pentru mulțimea  $[1,2,3]$  și  $k = 2$ , soluția este (nu neapărat în această ordine):  $\{[[3], [1]], [[2], [3, 1]], [[3], [2, 1]]\}$ .

nu

- C. Dându-se o listă formată din numere întregi, să se genereze în PROLOG lista submulțimilor cu număr par de elemente. Se vor scrie modelele matematice și modelele de flux pentru predicatele folosite.

**Exemplu-** pentru lista L=[2,3,4]  $\Rightarrow$  [[], [2,3], [2,4], [3,4]] (nu neapărat în această ordine)

f

- 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  $(1 \ s\ 5 \ (3 \ f\ (7)))$ .

f

# 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 definiție

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

Rescrieți această definiție pentru a evita apelul repetat **(G L)**. 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.

- 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]]`.

nu

- 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 și **P**=20 ⇒ [[2,10],[10,2],[5,4],[4,5]] (nu neapărat în această ordine)

```
% prodLista([1 ... ln] = { 1, n = 0
%                               { 1 * prodLista([2 ... 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 combinariile 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. []
%                           2. insereaza(l1, permutari([l2,...,ln]))
%
% perm(L:list, LRez:list)
% (i, o) - nedeterminist
% (i, i) - determinist
permutari([], []).
permutari([_|T], P):- permutari(T, L),
insereaza(E, L, P).
%
% aranjamente(l1,...,ln, k, SUMA) = 1. l1=permutari(combinari(L,K)),
%                                         daca conditie(l1,SUMA) = true
%                                         2. insereaza(l1, permutari([l2,...,ln]))
%
% 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).
```

- 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))) și **e=h => (a (h (g)) (h (d (h)) (h)))**

f

# 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 definiție 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.

- B.** Dându-se o listă neliniară care conține atomi numerici și nenumerici, se cere un program Lisp care verifică dacă următoarele trei liste sunt egale: lista tuturor atomilor de pe niveluri multiple de 3 (3, 6, etc.), lista tuturor atomilor de pe niveluri sub forma  $3k+1$  (1, 4, 7, etc.) lista tuturor atomilor de pe niveluri sub forma  $3k+2$  (2, 5, 8, etc.). De exemplu pentru lista (A 1 (A 1(A 1(B 777 (B (B 777 C) 777 C) C) D) D) rezultatul va fi true.

nu

- C. Dându-se o listă formată din numere întregi, să se genereze lista submulțimilor cu **k** elemente numere impare, în progresie aritmetică. Se vor scrie modelele matematice și modelele de flux pentru predicale folosite.

**Exemplu-** pentru lista  $L=[1,5,2,9,3]$  și  $k=3 \Rightarrow [[1,5,9],[1,3,5]]$  (nu neapărat în această ordine)

f

D. Se consideră o listă neliniară. Să se scrie o funcție care să aibă ca rezultat lista inițială în 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)b)** k=1 => (0 (1 (2 b)) (c (d)))    **c)** k=4 => lista nu se modifică

f