P1. Subpunctele a)

```
%ADĂUGAREA UNUI ELEMENT LA FINALUL UNEI LISTE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%add_f(L:lista, E:element, O:lista)
%adaugă la finalul unei liste un element E
%rezultatul se afla în O
%iio iii determinist

add_f([], E, [E]).

add_f([H|T], E, [H|C1]):-
      add_f(T, E, C1).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se scrie un predicat care determină cel mai mic
%multiplu comun al elementelor unei liste formate
%din numere întregi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%cmmmc(A:int, B:int, C:int, O:int)
%cmmmc între A și B, calculat în C
%rezultatul în O
%iiio iiii determinist

cmmmc(A, B, C, C):-
      C mod B =:= 0,
      C mod A =:= 0,
      !.

cmmmc(A, B, C, O):-
      K is C+1,
      cmmmc(A, B, K, O).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%cmmmcF(A:int, B:int, C:int)
%C <- cmmmc(A, B)

cmmmcF(A, B, C):-
      cmmmc(A, B, A, C).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%cmmmcL(L:lista, C:int, O:int)
%calculează în C cmmmc de elemente din lista
%rezultatul în O
%iii, iio determinist


cmmmcL([], C, C).
cmmmcL([H|T], C, O):-
      cmmmcF(H, C, C1),
      cmmmcL(T, C1, O).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%cmmmcFull(L:lista, C:int)
%C <- cmmmc(L)
%ii, io determinist

cmmmcFull([H|T], C):-
      cmmmcL([H|T], H, C).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se scrie un predicat care întoarce diferența a două mulțimi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%apare(L:lista, E:int)
%verifica dacă E apare în lista L
%determinist (ii)

apare([], _):-fail.
apare([E|_], E).
apare([H|T], E):-
      H =\= E,
      apare(T, E).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%mdif(A:lista, B:lista, C:lista, O:lista)
%diferenta A-B colectată în C
%rezultatul în O
%iiio iiii determinist

mdif([], _, C, C).

mdif([H|T], B, C, O):-
      apare(B, H),
      !,
      mdif(T, B, C, O).

mdif([H|T], B, C, O):-
      add_f(C, H, C1),
      mdif(T, B, C1, O).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%mdiff(A:lista, B:lista, C:lista)
%C <- A-B
%determinist iii, iio

mdiff(A, B, C):-
      mdif(A, B, [], C).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se verifice dacă două mulțimi sunt egale (fără să se facă apel
la diferența a două mulțimi)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
in([], _):- fail.
in([E|_], E).
in([_|T], E):- in(T, E).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%inclus(A:list, B:list)
%verifică dacă A este inclus în B
%ii determinist
inclus([], _).
inclus([H|T], B):-
      in(B, H),
      inclus(T, B).
```

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%egale(A:list, B:list)
%verifică dacă A === B
%ii determinist

egale(A, B):-
      inclus(A, B),
      inclus(B, A).
```

```prolog
%predicatul se satisface dacă o listă are un nr par de elemente
%eşuează în caz contrar
%NU SE POT NUMĂRA ELEMENTELE DIN LISTĂ
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%par(L:list)
%determină dacă lista L are nr par de elemente
%i determinist

par([]).
par([_,_|T]):-
      par(T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```prolog
%să se elimine dintr-o lista toate elementele care se repetă
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%aparitii_e(L:lista, E:int, N:int, O:int)
%numără aparițiile elementului E în lista L
%N variabilă de numărare
%O rezultat

aparitii_e([], _, N, N).

aparitii_e([H|T], E, N, O):-
      H =:= E,
      !,
      NPP is N+1,
      aparitii_e(T, E, NPP, O).

aparitii_e([_|T], E, N, O):-
      aparitii_e(T, E, N, O).

%wrapper
aparitii(L, E, N):-aparitii_e(L, E, 0, N).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%add_f(L, E, C)
%adaugă la final

add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%elimina(L:list, LC:list C:list, O:list)
%elimină elementele din L care se repetă
%C colectoare
%LC copie lista L
%O rezultat

elimina([], _, C, C).

elimina([H|T], LC, C, O):-
      aparitii(LC, H, N),
      N > 1,
      !,
      elimina(T, LC, C, O).

elimina([H|T], LC, C, O):-
      add_f(C, H, R),
      elimina(T, LC, R, O).

%wrapper
run(L, C):-
      elimina(L, L, [], C).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se scrie un predicat care testează dacă o listă este o mulțime
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%ap(L:list, E:int, N:int, R:int)
%numără în N numărul de apariții ale lui E în L
%rezultat în R
%iiio iiii determinist

ap([], _, N, N).
ap([H|T], E, N, R):-
      H =:= E,
      !,
      NPP is N+1,
      ap(T, E, NPP, R).
ap([_|T], E, N, R):-
      ap(T, E, N, R).
```

```
%wrapper
%iio iii determinist
aparitii(L, E, N):- ap(L, E, 0, N).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%multime(L:lista, LC:lista)
%verifiva dacă lista L este o mulțime
%LC - copia listei
%ii determinist
multime([], _).
multime([H|_], LC):-
      aparitii(LC, H, N),
      N > 1,
      !,
      fail.

multime([_|T], LC):-
      multime(T, LC).

%wrapper
%i determinist
e_multime(L):-multime(L, L).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se intercaleze un element pe poziția a n-a a unei liste
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%adaugare la final
add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
```

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%inter(L:list, C:list, R:list E:int, P:int, I:int)
%intercalează elementul E pe poziția P în lista L
%C colectoare
%R rezultat
%E elementul de intercalat
%P poziția pe care se intercalează
%I indicele curent

inter([], C, C, _, P, I):-
      P < I.

inter([], C, CPP, E, P, I):-
      P >= I,
      add_f(C, E, CPP).

inter(L, C, R, E, P, I):-
      P =:= I,
      !,
      add_f(C, E, CPP),
      IPP is I+1,
      inter(L, CPP, R, E, P, IPP).

inter([H|T], C, R, E, P, I):-
      add_f(C, H, CPP),
      IPP is I+1,
      inter(T, CPP, R, E, P, IPP).

%wrapper run(L:list, R:list, E:int, P:int)

run(L, R, E, P):- inter(L, [], R, E, P, 1).
```

```prolog
%sa se scrie un predicat care întoarce intersectia a doua multimi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%in(L:list, E:int)
%verifică dacă E apare în lista L
%ii determinist

in([], _):-fail.
in([E|_], E).
in([_|T], E):- in(T, E).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%adaugare final
add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%intersectie(A:list, B:list, C:list, R:list)
%intersecția dintre A și B
%C colectoare
%R rezultat
%iiio iiii determinist

intersectie([], _, C, C).

intersectie([H|T], B, C, R):-
      in(B, H),
      !,
      add_f(C, H, CPP),
      intersectie(T, B, CPP, R).

intersectie([_|T], B, C, R):-
      intersectie(T, B, C, R).

%wrapper run(A:list, B:list, R:list)
% A intersectat cu B
% R rezultat
% iio iii determinist

run(A, B, R):- intersectie(A, B, [], R).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se scrie un predicat care transformă
%o listă într-o mulțime, în ordinea ULTIMEI
%aparițiii
%exemplu [1, 2, 3, 1, 2] -> [3, 1, 2]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%reverse_o(L:list, C:list, R:list)
%inversează lista L
%C colectoare
%rezultat în R
%iio iii determinist
reverse_o([], R, R).
reverse_o([H|T], C, R):- reverse_o(T, [H|C], R).
reverse(L, R):- reverse_o(L, [], R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%add_f(L:lista, E:int, R:lista)
add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%apare(L:lista, E:int)
%verifica dacă E apare în L(true)
%(ii) determinist

apare([], _):-fail.
apare([E|_], E).
apare([H|T], E):-
      H =\= E,
      apare(T, E).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%lm(L:lista, C:lista, O:lista)
%transforma lista L într-o mulțime cu păstrarea pozițiilor (prima
apariție)
%rezultat în O
%iii, iio determinist

lmm([], C, C).

lmm([H|T], C, O):-
      apare(C, H),
      !,
      lmm(T, C, O).

lmm([H|T], C, O):-
      add_f(C, H, R),
      lmm(T, R, O).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%wrapper

ltos(L, R):-
      reverse(L, LR),
      lmm(LR, [], RR),
      reverse(RR, R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se scrie un predicat care transformă
%o listă într-o mulțime, în ordinea primei
%apariții
%exemplu [1, 2, 3, 1, 2] -> [1, 2, 3]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%add_f(L:lista, E:int, R:lista)
add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%apare(L:lista, E:int)
%verifica dacă E apare în L(true)
%(ii) determinist

apare([], _):-fail.
apare([E|_], E).
apare([H|T], E):-
      H =\= E,
      apare(T, E).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%lm(L:lista, C:lista, O:lista)
%transforma lista L într-o mulțime cu păstrarea pozițiilor
%rezultat în O
%iii, iio determinist

lmm([], C, C).

lmm([H|T], C, O):-
      apare(C, H),
      !,
      lmm(T, C, O).

lmm([H|T], C, O):-
      add_f(C, H, R),
      lmm(T, R, O).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%wrapper
ltos(L, R):- lmm(L, [], R).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se scrie un predicat care întoarce reuniunea a două mulțimi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%in(L:list, E:element)
%verifică dacă un element apare în lista

in([], _):-fail.
in([E|_], E).
in([_|T], E):-in(T, E).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%adaugare la final
add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
```

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%reuniune(A:list, B:list, O:list)
%efectuează reuniunea mulțimilor A și B
%rezultatul în O

reuniune(A, [], A).

reuniune(A, [H|T], O):-
      in(A, H),
      !,
      reuniune(A, T, O).

reuniune(A, [H|T], O):-
      add_f(A, H, C),
      reuniune(C, T, O).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```prolog
%să se substituie un element prin altul
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
add_f([], E, [E]).
add_f([H|T], E, [H|R]):-
      add_f(T, E, R).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%substituie(L:list, C:list, R:list, A:int, B:int)
%substituie orice element A din L cu elementul B
%C colectoare
%R rezultat
%iioii iiiii determinist

substituie([], C, C, _, _).

substituie([A|T], C, R, A, B):-
      !,
      add_f(C, B, CPP),
      substituie(T, CPP, R, A, B).

substituie([H|T], C, R, A, B):-
      add_f(C, H, CPP),
      substituie(T, CPP, R, A, B).

%wrapper run(L:list, R:list, A:int, B:int)
%ioii iiii determinist

run(L, R, A, B):- substituie(L, [], R, A, B).
```

```
%să se scrie un predicat care substituie
%într-o listă un element printr-o altă listă.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%add_f(L:lista, E:int, C:lista)
%adaugă la final elementul L
%iii, iio determinist
add_f([], E, [E]).
add_f([H|T], E, [H|C]):-add_f(T, E, C).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%concat(A:lista, B:lista, C:lista, O:lista)
%concatenează A și B
%C colectoare
%O rezultat
%iiio, iiii determinist

concat([], [], C, C).

concat([H|T], B, C, O):-
      add_f(C, H, R),
      concat(T, B, R, O).

concat([], [H|T], C, O):-
      add_f(C, H, R),
      concat([], T, R, O).

%concatenare full
concatenare(A, B, C):-concat(A, B, [], C).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%sub(A:lista, L:lista, C:lista, O:lista, E:int)
%substituie toate elementele E din A cu lista L
%C colectoare
%O rezultat
%iiioi, iiiii  determinist

sub([], _, C, C, _).

sub([H|T], L, C, O, E):-
      H == E,
      !,
      concatenare(C, L, R),
      sub(T, L, R, O, E).

sub([H|T], L, C, O, E):-
      concatenare(C, [H], R),
      sub(T, L, R, O, E).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%substituie(A:lista, L:lista, E:int, C:list)
%wrapper
substituie(A, L, E, C):-
      sub(A, L, [], C, E).

%pretty print
run(A, L, E):-
      substituie(A, L, E, C),
      write(C).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%să se determine dacă o listă are aspect de vale
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%vale(L:lista, F:int)
%verifica dacă lista este vale
%F = 0 dacă se coboară
%F = 1 dacă se urcă
% ii determinist

vale([], 1).
vale([_], 1).

vale([ A,B|T], -1):-
      A > B,
      vale([B|T], 0).

vale([ A,B|T], 0):-
      A > B,
      vale([B|T], 0).

vale([ A,B|T], 0):-
      B >= A,
      vale([B|T], 1).

vale([ A,B|T], 1):-
      A < B,
      vale([B|T], 1).

%wrapper run(L:lista)
%verifica dacă lista L are aspect de vale
%i determinist
run(L) :- vale(L, -1).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```