CURS 5

Nedeterminism. Arbori. Evitare apeluri recursive repetate. Recursivitate de coadă

Cuprins

1. Evitare apeluri recursive repetate

EXEMPLU 1.1 Să se calculeze minimul unei liste de numere întregi.

```
% minim(L: list of integer, M:integer)
% (i, o) - determinist
minim([A], A).
minim([H|T], Rez) :-
        minim(T, M),
        H =< M, !, Rez=H.
minim([_|T], M) :-
        minim(T, M).
```

Soluția 1

```
minim([A], A).
minim([H|T], M) :-
        minim(T, M),
        H > M, !.
minim([H|_], H).
```

Soluția 2. Se folosește un predicat auxiliar, pentru a evita apelul (recursiv) repetat din clauzele 2 și 3.

```
minim([A], A).
minim([H|T], Rez) :-
        minim(T, M), aux(H, M, Rez).
```

```
% aux(H: integer, M:integer, Rez:integer)
% (i, i, o) - determinist
aux(H, M, Rez) :-
        H =< M, !, Rez=H.
aux(_, M, M).
```

EXEMPLU 1.2 Se dă o listă numerică. Să se dea o soluție pentru evitarea apelului recursiv repetat. *Nu se vor redefini clauzele.*

```
% f(L:list of numbers, E: number)
% (i,o) – determinist
f([E],E).
f([H|T],Y):- f(T,X),
             H=<X,
             !,
             Y=H.
f([_|T],X):- f(T,X).
```

Soluție. Se folosește un predicat auxiliar. Soluție nu presupune înțelegerea semanticii.

```
f([E],E).
f([H|T],Y):- f(T,X),  aux(H, X, Y).
% aux(H: integer, X:integer, Y:integer)
% (i, i, o) - determinist
aux(H, X, Y) :-
             H=<X,
             !,
             Y=H.
aux(_, X, X).
```

EXEMPLU 1.3 Să se dea o soluție pentru evitarea apelului recursiv repetat.

```
% f(K:number, X:number)
% (i,o) – determinist
f(1,1):-!.
f(2,2):-!.
f(K,X):- K1 is K-1,
         f(K1, Y),
         Y>1,
          !,
          K2 is K-2,
          X=K2.
f(K,X):- K1 is K-1,
         f(K1, X).
```

Soluție. Se folosește un predicat auxiliar.

```
f(1,1):-!.
f(2,2):-!.
f(K,X) :- K1 is K-1,
          f(K1, Y),
          aux(K, Y, X).
% aux(K: integer, Y:integer, X:integer)
% (i, i, o) - determinist
aux(K, Y, X) :-
          Y>1,
          !,
          K2 is K-2,
          X=K2.
aux(_, Y, Y).
```

2. Arbori

Folosind obiecte compuse, se pot defini și prelucra în Prolog diverse structuri de date, precum arborii.

```
% domeniul corespunzător AB – domeniu cu alternative
% arbore=arb(integer, arbore, arbore);nil
% Functorul nil îl asociem arborelui vid
```

De exemplu, arborele

```
        A
      /   \
     B     C
    /       \
   D         E
```

se va reprezenta astfel

```
arb(A, arb(B,
               arb(D, nil, nil),
               nil
           ),
       arb(C, nil,
                  arb(E, nil, nil)
           )
   )
```

Se dă o listă numerică. Se cere să se afișeze elementele listei în ordine crescătoare. Se va folosi sortarea arborescentă (folosind un ABC).

Indicație. Se va construi un ABC cu elementele listei. Apoi, se va parcurge ABC în inordine.

% domeniul corespunzător ABC – domeniu cu alternative
% **arbore=arb(integer, arbore, arbore);nil**
% Functorul **nil** îl asociem arborelui vid

$$inserare(e, arb(r,s,d)) = \begin{cases} arb(e,\emptyset,\emptyset) & daca\ arb(r,s,d)\ e\ vid \\ arb(r, inserare(e,s), d) & daca\ e \le r \\ arb(r,s,inserare(e,d)) & altfel \end{cases}$$

```
% (integer, arbore, arbore) – (i,i,o) determinist
% insereaza un element într-un ABC
inserare(E, nil, arb(E, nil, nil)).
inserare(E, arb(R, S, D),arb(R, SNou, D)) :-
                E =< R,
                !,
                inserare(E, S, SNou).
inserare(E, arb(R, S, D), arb(R, S, DNou)) :-
                inserare(E, D, DNou).

% (arbore) – (i) determinist
% afişează nodurile arborelui în inordine
inordine(nil).
inordine(arb(R,S,D)) :-
                inordine(S),
                write(R),
                nl,
                inordine(D).
```

$$creeazaArb(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ inserare(l_1, creeazaArb(l_2 \dots l_n)) & altfel \end{cases}$$

```
% (arbore, list) – (i,o) determinist
% creează un ABC cu elementele unei liste
creeazaArb([], nil).
creeazaArb([H|T], Arb) :-
                creeazaArb(T, Arb1),
                inserare(H, Arb1, Arb).

% (list) – (i) determinist
% afişează elementele listei în ordine crescătoare (folosind sortare arborescentă)
sortare(L) :-
              creeazaArb(L,Arb),
              inordine(Arb).
```

3. Optimizarea prin recursivitate de coadă (tail recursion)

Recursivitatea are o mare problemă: consumă multă memorie. Dacă o procedură se repetă de 100 ori, 100 de stadii diferite ale execuţiei procedurii (cadre de stivă) sunt memorate.

Totuşi, există un caz special când o procedură se apelează pe ea fară să genereze cadru de stivă. Dacă procedura apelatoare apelează o procedură ca ultim pas al sau (după acest apel urmează punctul). Când procedura apelată se termină, procedura apelatoare nu mai are altceva de făcut. Aceasta înseamnă că procedura apelatoare nu are sens să-şi memoreze stadiul execuţiei, deoarece nu mai are nevoie de acesta.

Funcţionarea recursivităţii de coadă

Iată două reguli depre cum să faceţi o recursivitate de coadă:

1. Apelul recursiv este ultimul subgoal din clauza respectivă.
2. Nu există puncte de backtracking mai sus în acea clauză (adică, subgoal-urile de mai sus sunt deterministe).

Iată un exemplu:

```
tip(N) :-
        write(N),
        nl,
        Nou is N + 1,
        tip(Nou).
```

Această procedură foloseşte recursivitatea de coadă. Nu consumă memorie, şi nu se opreşte niciodată. Eventual, din cauza rotunjirilor, de la un moment va da rezultate incorecte, dar nu se va opri.

Exemple greşite de recursivitate de coadă

Iată cateva reguli despre cum să NU faceţi o recursivitate de coadă:

1. Dacă apelul recursiv nu este ultimul pas, procedura nu foloseşte recursivitatea de coadă.

Exemplu:

```
tip (N) :-
        write(N),
        nl,
        Nou is N + 1,
        tip (Nou),
        nl.
```

2. Un alt mod de a pierde recursivitatea de coadă este de a lăsa o alternativă neîncercată la momentul apelului recursiv.

Exemplu:

```
tip(N) :-
        write(N),
        nl,
        Nou is N + 1,
        tip(Nou).
tip(N) :-
        N < 0,
        write('N este negativ.').
```

Aici, prima clauză se apelează înainte ca a doua să fie încercată. După un anumit număr de paşi intră în criză de memorie.

3. Alternativa neîncercată nu trebuie neaparat să fie o clauza separată a procedurii recursive. Poate să fie o alternativă a unei clauze apelate din interiorul procedurii recursive.

Exemplu:

```
tip (N) :-
        write(N),
        nl,
        Nou is N + 1,
        verif(Nou),
        tip(Nou).
verif(Z) :- Z >= 0.
verif(Z) :- Z < 0.
```

Dacă N este pozitiv, prima clauză a predicatului **verif** a reuşit, dar a doua nu a fost încercată. Deci, **tip** trebuie să-şi pastreze o copie a cadrului de stivă.

Utilizarea tăieturii pentru păstrarea recursivităţii de coadă

A doua şi a treia situaţie de mai sus pot fi înlăturate dacă se utilizează tăietura, chiar dacă există alternative neîncercate.

Exemplu la situaţia a doua:

```
tip (N) :-
        N>= 0,
        !,
        write(N),
        nl,
        Nou = N + 1,
        tip(Nou).
tip(N) :-
        N < 0,
        write("N este negativ.").
```

Exemplu la situaţia a treia:

```
tip(N) :-
        write(N),
        nl,
        Nou = N + 1,
        verif(Nou),
        !,
        tip(Nou).
verif(Z) :- Z >= 0.
verif(Z) :- Z < 0.
```

4. Exemple predicate nedeterministe (continuare)

EXEMPLU 3.1 Să se scrie un predicat nedeterminist care generează combinări cu *k* elemente dintr-o mulțime nevidă reprezentată sub forma unei liste.

```
? comb([1, 2, 3], 2, C).     /*model de flux (i, i, o) - nedeterminist*/
C = [2, 3];
C = [1, 2];
C = [1, 3].
```

Observaţie: Pentru determinarea combinărilor unei liste [E|L] (care are capul E şi coada L) luate câte **K**, sunt următoarele cazuri posibile:

i. dacă **K**=1, atunci o combinare este chiar [E]
ii. determină o combinare cu K elemente a listei L;
iii. plasează elementul E pe prima poziţie în combinările cu **K**-1 elemente ale listei L (dacă **K**>1).

Modelul recursiv pentru generare este:

$comb(l_1\, l_2 \ldots l_n, k) =$

1. (l_1) $\quad daca\ k = 1$
2. $comb(l_2 \ldots l_n, k)$
3. $l_1 \oplus \quad comb(l_2 \ldots l_n, k-1) \quad daca\ k > 1$

Vom folosi predicatul nedeterminist **comb** care va genera toate combinările. Dacă se dorește colectarea combinărilor într-o listă, se va putea folosi predicatul **findall**.

Programul SWI-Prolog este următorul:

```
% comb(L: list, K:integer, C:list)
% (i, i, o) - nedeterminist
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).
```

EXEMPLU 3.2 Să se scrie un predicat nedeterminist care inserează un element, pe toate pozițiile, într-o listă.

```
? insereaza(1, [2, 3], L).    /*model de flux (i, i, o)  - nedeterminist*/
L = [1, 2, 3];
L = [2, 1, 3];
L = [2, 3, 1].
```

Model recursiv

$insereaza(e, l_1 l_2 \dots l_n) =$

1. $e \oplus l_1, l_2 \dots l_n$
2. $l_1 \oplus insereaza(e, l_2 \dots l_n)$

```
% insereaza(E: element, L:List, LRez:list)
% (i, i, o) - nedeterminist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
                insereaza(E, T, Rez).
```

Observăm că, pe lângă modelul de flux (i, i, o) descris anterior, predicatul **insereaza** funcţionează cu mai multe modele de flux (în unele modele de flux preducatul fiind determinist, în altele nedeterminist).

- insereaza(E, L, [1, 2, 3]), cu modelul de flux (o, o, i) şi soluţiile
  E=1, L = [2, 3]
  E=2, L = [1, 3]
  E=3, L = [1, 2]
- insereaza(1, L, [1, 2, 3]), cu modelul de flux (i, o, i) şi soluţia
  L = [2, 3]
- insereaza(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) şi soluţia
  E = 2

EXEMPLU 3.3 Să se scrie un predicat nedeterminist care şterge un element, pe rând, de pe toate pozițiile pe care acesta apare într-o listă.

? elimin(1, L, [1, 2, 1, 3]). /*model de flux (i, o, i) - nedeterminist*/
L = [2, 1, 3];
L = [1, 2, 3];

$elimin(e, l_1 l_2 \dots l_n) =$

1. $l_2 \dots l_n \qquad daca\ e = l_1$
2. $l_1 \oplus elimin(e, l_2 \dots l_n)$

```
% elimin(E: element, LRez:list, L:list)
% (i, o, i) – nedeterminist
elimin(E, L, [E|L]).
elimin(E, [A|L], [A|X]) :-
	elimin(E, L, X).
```

Observăm că predicatul **elimin** funcţionează cu mai multe modele de flux. Astfel, următoarele întrebări sunt valide:

- elimin(E, L, [1, 2, 3]), cu modelul de flux (o, o, i) şi soluţiile
  E=1, L = [2, 3]
  E=2, L = [1, 3]
  E=3, L = [1, 2]
- elimin(1, [2, 3], L), cu modelul de flux (i, i, o) şi soluţiile
  L = [1, 2, 3]
  L = [2, 1, 3]
  L = [2, 3, 1]
- elimin(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) şi soluţia
  E = 2

EXEMPLU 3.4 Să se scrie un predicat nedeterminist care generează permutările unei liste.

? perm([1, 2, 3], P). /*model de flux (i, o,) - nedeterminist*/
P = [1, 2, 3];
P = [1, 3, 2];
P = [2, 1, 3];
P = [2, 3, 1];
P = [3, 1, 2];
P = [3, 2, 1]

Cum obținem permutările listei [1, 2, 3] dacă ştim să generăm permutările sublistei [2, 3] (adică [2, 3] şi [3, 2])?

Pentru determinarea permutărilor unei liste [E|L], care are capul E şi coada L, vom proceda în felul următor:

1. determină o permutare L1 a listei L;

2. plasează elementul E pe toate poziţiile listei L1 şi produce în acest fel lista X care va fi o permutare a listei iniţiale [E|L].

Modelul recursiv este:

$perm(l_1 l_2 \dots l_n) =$

1. $\emptyset$ $\quad daca\ l\ e\ vida$
2. $insereaza(l_1, perm(l_2 \dots l_n)) \quad altfel$

```
% perm(L:list, LRez:list)
% (i, o) – nedeterminist
perm([], []).
perm([E|T], P) :-
        perm(T, L),
        insereaza(E, L, P). % (i, i, o)



% alternativa pentru clauza 2
perm(L, [H|T]) :-
        elimin(H, Z, L),   % (o o, i)
        perm(Z, T).
% alternativa pentru clauza 2
perm(L, [H|T]) :-
        insereaza(H, Z, L),   % (o o, i)
        perm(Z, T).
% alternativa pentru clauza 2
perm([E|T], P) :-
        perm(T, L),
        elimin(E, L, P),   % (i ,i, o)
```

TEMĂ Să se scrie un predicat nedeterminist care generează aranjamente cu *k* elemente dintr-o mulțime nevidă reprezintată sub forma unei liste.

```
? aranj([1, 2, 3], 2, A).    /*model de flux (i, i, o) - nedeterminist*/
A = [2, 3];
A = [3, 2];
A = [1, 2];
A = [2, 1] ;
A = [1, 3];
A = [3, 1] ;
```