

## CURS 6

### Evitare apeluri recursive repetate. Obiecte compuse.

#### Recursivitate de coadă

#### Cuprins

1. Evitare apeluri recursive repetate.....	1
2. Obiecte simple și obiecte compuse. ....	3
3. Optimizarea prin recursivitate de coadă (tail recursion).....	6
Funcționarea recursivității de coadă .....	6
Utilizarea tăieturii pentru păstrarea recursivității de coadă .....	8
4. Concluzii – programare logică.....	9

### 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.** (generală) 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).
```

**Soluția 2** (specifică)

```
minim([A], A).
```

```

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

```

**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. Soluția nu presupune înțelegerea semanticii codului.

```

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. Obiecte simple și obiecte compuse.

### Obiecte simple

Un obiect simplu este fie o variabilă, fie o constantă. O constantă este fie un caracter, fie un număr, fie un atom (simbol sau string).

Variabilele Prolog sunt locale, nu globale. Adică, dacă două clauze conțin fiecare câte o variabilă numită X, cele două variabile sunt distincte și, de obicei, nu au efect una asupra celeilalte.

### Obiecte compuse și functori

**Obiectele compuse** ne permit să tratăm mai multe informații ca pe un singur element, într-un astfel de mod încât să-l putem utiliza și pe bucăți. Fie, de exemplu, data de *2 februarie 1998*. Constă din trei informații, ziua, luna și anul, dar e util să o tratăm ca un singur obiect cu o structură arborescentă:

```
      DATA
    /  |  \
2 februarie 1998
```

Acest lucru se poate face scriind obiectul compus astfel:

```
data(2, "februarie", 1998)
```

Aceasta seamănă cu un fapt Prolog, dar nu este decât un obiect (o dată) pe care îl putem manevra la fel ca pe un simbol sau număr. Din punct de vedere sintactic, începe cu un nume (sau **functor**, în acest caz cuvântul **data**) urmat de trei argumente.

**Notă.** Functorul în Prolog nu este același lucru cu funcția din alte limbaje de programare. Este doar un nume care identifică un tip de date compuse și care ține argumentele laolaltă.

Argumentele unei date compuse pot fi chiar ele compuse. Iată un exemplu:  
naștere(persoana(“Ioan”, “Popescu”), data(2, “februarie”, 1918))

## Unificarea obiectelor compuse

Un obiect compus se poate unifica fie cu o variabilă simplă, fie cu un obiect compus care se potrivește cu el. De exemplu,

data(2, “februarie”, 1998)

se potrivește cu variabila liberă  $X$  și are ca rezultat legarea lui  $X$  de  $\text{data}(\dots)$ . De asemenea, obiectul compus de mai sus se potrivește și cu

data(Zi, Lu, An)

și are ca rezultat legarea variabilei  $Zi$  de valoarea 2, a variabilei  $Lu$  de valoarea “februarie” și a variabilei  $An$  de valoarea 1998.

### Observații

- Declarație pentru *specificarea* unui domeniu cu alternative  
    % domeniu = alternativa<sub>1</sub>(dom, dom, ..., dom);  
    %           alternativa<sub>2</sub>(dom, dom, ..., dom);  
    %           ...
- Functorii pot fi folosiți pentru controla argumentele care pot avea tipuri multiple  
    % element = i(integer); r(real); s(string)

## 2.1 Liste eterogene

Se dă o listă de numere întregi sau/și simboluri. Se cere suma numerelor întregi pare din listă.

?- suma([i(1),s(a),i(2)],S).  
S = 2.

?- suma([s(a),s(b),i(1)],S).  
S=0.

% suma(L: heterogeneous list, S: number)

% (i,o) - determ

suma([],0).

sumalist([i(H)|T], S) :-

    H mod 2 =:=0, !,

    suma(T,S1),

    S is S1+H.

% atenție la o clauză de forma suma ([i(\_)|T], S) :- suma(T,S).

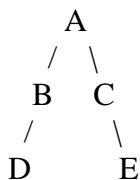
suma ([\_|T], S) :- suma(T,S).

## 2.2 Arbori binari

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



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
```

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

```
% (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
% afișează nodurile arborelui în inordine
inordine(nil).
inordine(arb(R,S,D)) :-
    inordine(S),
    write(R),
    nl,
    inordine(D).

```

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

```

% (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
% afișează 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 execuției procedurii (cadre de stivă) sunt memorate.

Totuși, există un caz special când o procedură se apelează pe ea fără 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-și memoreze stadiul execuției, deoarece nu mai are nevoie de acesta.

#### Funcționarea recursivității de coadă

Iată două reguli depre cum să faceți 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 folosește recursivitatea de coadă. Nu consumă memorie, și nu se oprește niciodată. Eventual, din cauza rotunjirilor, de la un moment va da rezultate incorecte, dar nu se va opri.

### Exemple greșite de recursivitate de coadă

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

1. Dacă apelul recursiv nu este ultimul pas, procedura nu folosește 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 pași intră în criză de memorie.

3. Alternativa neîncercată nu trebuie neaparat să fie o clauză 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 reușit, dar a doua nu a fost încercată. Deci, **tip** trebuie să-și pastreze o copie a cadrului de stivă.

### Utilizarea tăieturii pentru păstrarea recursivității de coadă

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

Exemplu la situația a doua:

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

Exemplu la situația a treia:

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



## 4. Concluzii – programare logică

1. *“Learning Prolog provides you with a different perspective on programming that, once understood, can be applied to other languages as well.”*
2. *“You should learn Prolog as a part of your personal quest to become a better programmer.”*

### 3. Inteligența Artificială

- a. *Expert systems* (Sisteme expert)
- b. Paradigma programării bazate pe constrângeri (**constraint programming**)
  - *Constraint Satisfaction Problems* (CSP)
- c. Învățare automată (**Machine Learning**)
  - *Inductive logic programming*
- d. *Multiagent Systems* (Sisteme multiagent)
  - Comunicare între agenți
  - Planificare
  - etc