

## CURS 4

### Controlarea backtrackingului. Liste eterogene

#### Cuprins

|   |   |
|---|---|
| 1. Controlarea backtracking-ului. Predicatele “cut” (tăietura) și fail..... | 1 |
| 1.1 Predicatul ! (cut) – “tăietura” .....                                   | 1 |
| 1.2 Predicatul “fail” .....   | 4 |
| 2. Obiecte simple și obiecte compuse. ....                                  | 6 |
| 3. Exemple .....  | 8 |

**Exemplu** Fie următoarele definiții de predicate.

1) Care este efectul următoarelor interogări? Apare **false** la finalul căutării?

? g([1, 3, 4], 2, P).

? f([1,2,3,4], P).

% g(L:list, E: element, LRez: list)

% (i, i, o) – nedeterminist

g([H|\_], E, [E,H]).

g([\_|T], E, P):-

    g(T, E, P).

% f(L:list, LRez: list)

% (i, o) – nedeterminist

f([H|T],P):-

    g(T, H, P).

f([\_|T], P):-

    f(T, P).

2) Funcționează predicatele în alte modele de flux?

?- g([1,2,3], E, [4,1]).

### 1. Controlarea backtracking-ului. Predicatele “cut” (tăietura) și fail

#### 1.1 Predicatul ! (cut) – “tăietura”

Limbajul Prolog conține predicatul cut (!) folosit pentru a preveni backtracking-ul. Când se procesează predicatul !, apelul reușește imediat și se trece la subgoalul următor. O dată ce s-a trecut peste o tăietură, nu este posibilă revenirea la subgoal-urile plasate înaintea ei și nu este posibil backtracking-ul la alte reguli ce definesc predicatul în execuție.

Există două utilizări importante ale tăieturii:

1. Când știm dinainte că anumite posibilități nu vor duce la soluții, este o pierdere de timp să lăsăm sistemul să lucreze. În acest caz, tăietura se numește **tăietură verde**.
2. Când logica programului cere o tăietură, pentru prevenirea luării în considerație a subgoal-urilor alternative, pentru a evita obținerea de soluții eronate. În acest caz, tăietura se numește **tăietură roșie**.

### Prevenirea backtracking-ului la un subgoal anterior

În acest caz tăietura se utilizează astfel:  $r1 :- a, b, !, c.$

Aceasta este o modalitate de a spune că suntem mulțumiți cu primele soluții descoperite cu subgoal-urile a și b. Deși Prolog ar putea găsi mai multe soluții prin apelul la c, nu este autorizat să revină la a și b. De asemenea, nu este autorizat să revină la altă clauză care definește predicatul r1.

### Prevenirea backtracking-ului la următoarea clauză

Tăietura poate fi utilizată pentru a-i spune sistemului Prolog că a ales corect clauza pentru un predicat particular. De exemplu, fie codul următor:

```
r(1) :- !, a, b, c.  
r(2) :- !, d.  
r(3) :- !, e.  
r(_) :- write("Aici intră restul apelurilor").
```

Folosirea tăieturii face predicatul r determinist. Aici, Prolog apelează predicatul r cu un argument întreg. Să presupunem că apelul este r(1). Prolog caută o potrivire a apelului. O găsește la prima clauză. Faptul că imediat după intrarea în clauză urmează o tăietură, împiedică Prolog să mai caute și alte potriviri ale apelului r(1) cu alte clauze.

**Observație.** Acest tip de structură este echivalent cu o instrucțiune de tip *case* unde condiția de test a fost inclusă în capul clauzei. La fel de bine s-ar fi putut spune și

```
r(X) :- X = 1, !, a, b, c.  
r(X) :- X = 2, !, d.  
r(X) :- X = 3, !, e.  
r(_) :- write("Aici intra restul apelurilor").
```

**Notă.** Deci, următoarele secvențe sunt echivalente:

|                 |                                  |
|-----------------|----------------------------------|
| case X of       |                                  |
| v1: corp1;      | predicat(X) :- X = v1, !, corp1. |
| v2: corp2;      | predicat(X) :- X = v2, !, corp2. |
| ...             | ...                              |
| else corp_else. | predicat(X) :- corp_else.        |

De asemenea, următoarele secvențe sunt echivalente:

```

if cond1 then
    corp1
else if cond2 then
    corp2
...
else
    corp_else.

```

```

predicat(...) :-
    cond1, !, corp1.
predicat(...) :-
    cond2, !, corp2.
...
predicat(...) :-
    corp_else.

```

## EXAMPLE

### 1. Fie următorul cod Prolog

|  |  |
|--|--|
| <pre> % p(E: integer) % (o) – nedeterminist p(1). p(2). % q(E: integer) % (o) – nedeterminist q(3). q(4). </pre> | <pre> % r(E1: integer, E2: integer) % (o, o) – nedeterminist % V1 r(X, Y) :- !, p(X), q(Y). % V2 r(X, Y) :- p(X), !, q(Y). % V3 r(X, Y) :- p(X), q(Y), !. </pre> |
|--|--|

Care sunt soluțiile furnizate la goal-ul

? r(X,Y).

în cele 3 variante (V1, V2, V3) de definire a preducatului r?

**R:**

- V1 - sunt 4 soluții:  
X=1 Y=3  
X=1 Y=4  
X=2 Y=3  
X=2 Y=4
- V2 - sunt 2 soluții:  
X=1 Y=3  
X=1 Y=4
- V3 – e 1 soluție:  
X=1 Y=3

### 2. Fie următorul cod Prolog

```

% p(L: list, S:integer)
% (i, o) –determinist
p([], 0).
p([H|T], S) :-
    H > 0,
    !,
    p(T, S1),
    S is S1 + H.

```

```

? p([1, 2, 3, 4], S).
S=10.

```

```

? p([1, -1, 2, -2], S).
S=3.

```

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

Ce se întâmplă dacă se mută tăietura înaintea condiției, adică dacă în cea de-a doua clauză se mută tăietura înaintea condiției?

**% p(L: list, S:integer)**

**% (i, o) –determinist**

$p([], 0).$

$p([H|T], S) :-$

**!,**

$H > 0,$

$p(T, S1),$

$S \text{ is } S1 + H.$

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

?  $p([1, 2, 3, 4], S).$

$S=10.$

?  $p([1, -1, 2, -2], S).$

**false**

## 1.2 Predicatul “fail”

Valoarea lui *fail* este eșec. Prin aceasta el încurajează backtracking-ul. Efectul lui este același cu al unui predicat imposibil, de genul  $2 = 3$ . Fie următorul exemplu:

predicat(a, b).

predicat(c, d).

predicat(e, f).

toate :-

predicat(X, Y),

write(X),write(Y),nl,

fail.

toate1 :-

predicat(X, Y),

write(X),write(Y),nl.

Predicatele **toate** și **toate1** sunt fără parametri, și ca atare sistemul va trebui să răspundă dacă există X și Y astfel încât aceste predicate să aibă loc.

?- toate.

ab

cd

ef

**false.**

?-toate1.

ab

**true;**

cd

**true;**

ef

**true.**

?-predicat(X,Y).

X = a,

Y = b ;

X = c,

Y = d ;

X = e,

Y = f.

Faptul că apelul predicatului **toate** se termină cu *fail* (care eșuează întotdeauna) obligă Prolog să înceapă backtracking prin corpul regulii **toate**. Prolog va reveni până la ultimul apel care poate oferi mai multe soluții. Predicatul **write** nu poate da alte soluții, deci revine la apelul lui **predicat**.

### Observații.

- Acel **false** de la sfârșitul soluțiilor semnifică faptul că predicatul ‘toate’ nu a fost satisfăcut.
- După *fail* nu are rost să puneti nici un predicat, deoarece Prolog nu va ajunge să-l execute niciodată.

**Notă.** Secvențele următoare sunt echivalente:

cât timp *condiție* execută  
*corp*

predicat :-  
*condiție*,  
*corp*,  
*fail*.

### EXEMPLU

Fie următorul cod Prolog

```
% q(E: integer)
% (o) – nedeterminist
q(1).
q(2).
q(3).
q(4).
p :- q(I), I<3, write (I), nl, fail.
```

Care este rezultatul următoarei întrebări **?p.**, în condițiile în care apare/nu apare fail la finalul ultimei clauze?

- cu **fail** la finalul clauzei, soluțiile sunt  
1  
2  
**false**
- fără **fail** la finalul clauzei, soluțiile sunt  
1  
true;  
2  
true;  
**false**

## 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 data(...). 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.

### Observatii

- Convenim să folosim următoarea 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)

### Liste eterogene

În SWI-Prolog listele sunt eterogene, elementele componente pot fi de tipuri diferite. Pentru a se determina tipul unui element al listei, se folosesc predicatele predefinite în SWI (**number**, **is\_list**, etc)

Varianta 1. Se dă o listă eterogenă formată din numere, simboluri și/sau liste de numere. Se cere să se determine suma numerelor din lista eterogenă.

```
?- suma([1,a,[1,2,3],4],S).  
S = 11.
```

```
?- suma([a,b,[],],S).  
S=0.
```

```
%(L:list of numbers, S: number)  
% (i,o) - determ  
sumalist([],0).  
sumalist([H|T],S) :- sumalist(T,S1),  
                  S is S1+H.
```

```
%(L:list, S: number)  
% (i,o) - determ  
suma([],0).  
suma([H|T],S):-number(H),  
              !,  
              suma(T,S1),  
              S is H+S1.  
suma([H|T],S):-is_list(H),  
              !,  
              sumalist(H,S1),  
              suma(T,S2),  
              S is S1+S2.  
suma([_|T],S):-suma(T,S).
```

Varianta 2. O altă posibilitate (mai generală) de a gestiona liste eterogene este folosind obiecte compuse/functori.

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).

### 3. Exemple

**EXEMPLU 3.1** Să se scrie un predicat care concatenează două liste.

? concatene([1, 2], [3, 4], L).  
L = [1, 2, 3, 4].

Pentru combinarea listelor L1 și L2 pentru a forma lista L3 vom utiliza un algoritm recursiv de genul:

1. Dacă L1 = [] atunci L3 = L2.
2. Altfel, capul lui L3 este capul lui L1 și coada lui L3 se obține prin combinarea cozii lui L1 cu lista L2.

Să explicăm puțin acest algoritm. Fie listele L1 = [a<sub>1</sub>, ..., a<sub>n</sub>] și L2 = [b<sub>1</sub>, ..., b<sub>m</sub>]. Atunci lista L3 va trebui să fie L3 = [a<sub>1</sub>, ..., a<sub>n</sub>, b<sub>1</sub>, ..., b<sub>m</sub>] sau, dacă o separăm în cap și coadă, L3 = [a<sub>1</sub> | [a<sub>2</sub>, ..., a<sub>n</sub>, b<sub>1</sub>, ..., b<sub>m</sub>]]. De aici rezultă că:

1. capul listei L3 este capul listei L1;
2. coada listei L3 se obține prin concatenarea cozii listei L1 cu lista L2.

Mai departe, deoarece recursivitatea constă din reducerea complexității problemei prin scurtarea primei liste, rezultă că ieșirea din recursivitate va avea loc odată cu epuizarea listei L1, deci când L1 este []. De remarcat că condiția inversă, anume dacă L2 este [] atunci L3 este L1, este inutilă.

Programul SWI-Prolog este următorul:



## Model recursiv

$$\text{concatenare}(l_1 l_2 \dots l_n, l'_1 \dots l'_m) = \begin{cases} l'_1 & \text{daca } l = \emptyset \\ l_1 \oplus \text{concatenare}(l_2 \dots l_n, l'_1 \dots l'_m) & \text{altfel} \end{cases}$$

**% (concatenare(L1: list, L2: list, L3: list))**

**% (i, i, o) - determinist**

concatenare([], L, L).

concatenare([H|L1], L2, [H|L3]) :-

concatenare(L1, L2, L3).

Se observă că predicatul **concatenare** descris mai sus funcționează cu mai multe modele de flux, fiind nedeterminist în unele modele de flux, determinist în altele.

De exemplu, pentru întrebările

? concatenare (L1, L2, [1, 2, 3]).

**/\* model de flux (o,o,i) - nedeterminist \*/**

L1=[] L2=[1, 2, 3]

L1=[1] L2=[2, 3]

? concatenare (L, [3, 4], [1, 2, 3, 4]).

**/\* model de flux (o,i,i) sau (i,o,i) - determinist\*/**

L=[1, 2]

**EXEMPLU 3.2** Să se scrie un predicat care determină lista submulțimilor unei liste reprezentate sub formă de mulțime.

? submulțimi([1, 2], L)

va produce L = [[], [2], [1], [1, 2]]

**Observație:** Dacă lista e vidă, submulțimea sa e lista vidă. Pentru determinarea submulțimilor unei liste [E|L], care are capul E și coada L, vom proceda în felul următor:

- i. determină o submulțime a listei L
- ii. plasează elementul E pe prima poziție într-o submulțime a listei L

$$\text{subm}(l_1, l_2, \dots, l_n) =$$

1.  $\emptyset$  *daca l e vida*

2.  $\text{subm}(l_2, \dots, l_n)$

3.  $l_1 \oplus \text{subm}(l_2, \dots, l_n)$

Vom folosi predicatul nedeterminist **subm** care va genera submulțimile, după care vor fi colectate toate soluțiile acestuia, folosind predicatul **findall**.

Codul SWI-Prolog este indicat mai jos

```
% subm(L: list, Subm:list)
% (i, o) - nedeterminist
subm([], []).
subm([_|T], S) :- subm(T, S).
subm([H|T], [H|S]) :- subm(T, S).

% submultimi(L: list, LRez:list of lists)
% (i, o) - determinist
submultimi(L, LRez):-findall(S, subm(L,S), LRez).
```

### **EXEMPLU 3.3**

|  |   |
|--|---|
| <pre>% sP(L:list of numbers, L: list of number) % (i,o) – nondeterm sP([], []). sP ([_ T],S):-sP(T,S). sP ([H T],[H S]):- H mod 2 =:=0,                     !,                     sP(T,S). sP ([H T],[H S]):-sI(T,S).</pre> | <pre>% sI(L:list of numbers, L: list of number) % (i,o) – nondeterm  sI([H],[H]):-H mod 2 =\=0, !. sI([_ T],S):-sI(T,S). sI([H T],[H S]):-H mod 2 =:=0,                     !,                     sI(T,S). sI([H T],[H S]):-sP(T,S).</pre> |
|--|---|

? sP([1, 2, 3], S).

[]  
[2]  
[1, 3]  
[1, 2, 3]

?sI([1, 2, 3], S).

[3]  
[2, 3]  
[1]  
[1, 2]  
**false**

**EXEMPLU 3.4** Fie următoarele definiții de predicate. Care este efectul următoarei interogări?

?- det([1,2,1,3,1,7,8],1,L).

```
% det( L:list of elements, E:element, LRez: list of numbers)
%(i, i, o) - determinist
det(L, E, LRez):- det_aux(L, E, LRez, 1).
```

```
% det_aux( L:list of elements, E:element, LRez: list of numbers, P:intreg)
% (i, i, o, i) - determinist
```

```

det_aux([], _, [], _).
det_aux([E|T], E, [P|LRez], P) :-
    !,
    P1 is P+1,
    det_aux(T, E, LRez, P1).
det_aux([_|T], E, LRez, P) :-
    P1 is P+1,
    det_aux(T, E, LRez, P1).

```

**Solutia pentru evitarea apelului repetat** (marcat cu albastru)

- folosirea unui predicat auxiliar

```

det_aux([], _, [], _).
det_aux([H|T], E, LRez, P) :-
    P1 is P+1,
    det_aux(T, E, L, P1),
    prel(H, E, P, L, LRez).

```

```

% prel(H: element, E:element, P: number, L:list, LRez: list of numbers)
% (i, i, i, i, o) - determinist
prel(E, E, P, L, [P|L]) :- !.
prel(_, _, _, L, L).

```