CURS 4

Controlarea backtrackingului

Cuprins

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

1.1 Predicatul ! (cut) – “tăietura”

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

Există două utilizări importante ale tăieturii:

1. Când ştim dinainte că anumite posibilităţi nu vor duce la soluţii, este o pierdere de timp să lăsăm sistemul să lucreze. În acest caz, tăietura se numeşte **tăietură verde**.
2. Când logica programului cere o tăietură, pentru prevenirea luării în consideraţie a subgoal-urilor alternative, pentru a evita obţinerea de soluţii eronate. În acest caz, tăietura se numeşte **tăietură roşie**.

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 multumiţi cu primele soluţii descoperite cu subgoal-urile a şi b. Deşi Prolog ar putea găsi mai multe soluţii prin apelul la c, nu este autorizat să revină la a şi b. De-asemenea, nu este autorizat să revina la altă clauză care defineşte 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 găseşte la prima clauză. Faptul că imediat după intrarea în clauză urmează o tăietură, împiedică Prolog să mai caute şi alte potriviri ale apelului r(1) cu alte clauze.

Observaţie. Acest tip de structură este echivalent cu o instrucţiune de tip *case* unde condiţia de test a fost inclusă în capul clauzei. La fel de bine s-ar fi putut spune şi

```
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 secvenţe sunt echivalente:

```
case X of
        v1: corp1;
        v2: corp2;
        ...
        else corp_else.
```

```
predicat(X) :- X = v1, !, corp1.
predicat(X) :- X = v2, !, corp2.
...
predicat(X) :- corp_else.
```

De asemenea, următoarele secvenţe sunt echivalente:

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

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

EXEMPLE

1. Fie următorul cod Prolog

```
% p(E: integer)
% (o) – nedeterminist
p(1).
p(2).
% q(E: integer)
% (o) – nedeterminist
q(3).
q(4).
```

```
% 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), !.
```

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

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

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

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 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 acelaşi 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** şi **toate1** sunt fără parametri, şi ca atare sistemul va trebui să răspundă dacă există X şi 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 eşuează întotdeauna) obligă Prolog să înceapă backtracking prin corpul regulii **toate**. Prolog va reveni până la ultimul apel care poate oferi mai multe soluţii. Predicatul **write** nu poate da alte soluţii, deci revine la apelul lui **predicat**.

Observaţii.

- Acel **false** de la sfârşitul soluţiilor semnifică faptul că predicatul 'toate' nu a fost satisfăcut.
- Dupa *fail* nu are rost să puneţi nici un predicat, deoarece Prolog nu va ajunge să-l execute niciodată.

Notă. Secvenţele următoare sunt echivalente:

```
cât timp condiţie execută
        corp
```

```
predicat :-
        condiţie,
        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. Exemple

EXEMPLU 2.1 Să se scrie un predicat care concatenează două liste.

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

Pentru combinarea listelor L1 si 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 şi coada lui L3 se obţine prin combinarea cozii lui L1 cu lista L2.

Să explicăm puţin acest algoritm. Fie listele L1 = [a_1, ..., a_n] şi L2 = [b_1, ..., b_m]. Atunci lista L3 va trebui să fie L3 = [a_1, ..., a_n, b_1, ..., b_m] sau, dacă o separăm în cap şi coadă, L3 = [a_1 | [a_2, ..., a_n, b_1, ..., b_m]]. De aici rezultă că:

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

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

Programul SWI-Prolog este următorul:

Model recursiv

$$concatenare(\ l_1 l_2 \dots l_n, l'_2 \dots l'_m) = \begin{cases} l' & daca\ l = \emptyset \\ l_1 \oplus concatenare(l_2 \dots l_n, l'_2 \dots l'_m) & altfel \end{cases}$$

```
%( 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 funcţionează 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 2.2 Să se scrie un predicat care determină lista submulțimilor unei mulțimi reprezentate sub formă de listă.

? submulţimi([1, 2], L)
va produce L = [[], [2], [1], [1, 2]]

Observaţie: Dacă lista e vidă, submulţimea sa e lista vidă. Pentru determinarea submulţimilor unei liste [E|L], care are capul E şi coada L, vom proceda în felul următor:

i. determină o submulţime a listei L
ii. plasează elementul E pe prima poziţie într-o submulţime a listei L

$$subm(l_1, l_2, \dots, l_n) =$$

1. $\emptyset$ $\quad daca\ l\ e\ vida$
2. $subm(l_2, \dots, l_n)$
3. $l_1 \oplus \quad subm(l_2, \dots, l_n)$

Vom folosi predicatul nedeterminist **subm** care va genera submulţimile, după care vor fi colectate toate soluţiile 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 2.3

```
% 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).
```

```
% 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).
```

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

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

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

[3]
[2, 3]
[1]
[1, 2]
false
```

EXEMPLU 2.4 Fie următoarele definiții de predicate. Care este efectul următaorei 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).
```

Soluția pentru refactorizare (secvență marcată 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).
```

EXEMPLU 2.5 Fie următoarele definiții de predicate.

```
% 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).
```

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

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

- Funcționează predicatele în alte modele de flux?

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