

Curs 13

Cuprins

- 1 Clauza ! (cut)
- 2 Negarea unui predicat: \+ pred(X)
- 3 Baze de date dinamice

Bibliografie:

P. Blackburn, J. Bos. K. Striegnitz, Learn Prolog Now! <http://cs.union.edu/~striegnk/learn-prolog-now/html/node88.html>

Clauza ! (cut)

Backtracking

Prolog folosește **backtracking** pentru a răspunde întrebărilor:

- În momentul în care Prolog încearcă să găsească un răspuns la o întrebare, ține minte toate **punctele de decizie**.
 - ▣ Puncte de decizie = situațiile în care găsește mai multe instanțieri.
- De fiecare dată când un drum eșuează sau se termină, Prolog sare la ultima alegere făcută și încearcă următoarea alternativă.

Backtracking

Exemplu

a(1).

b(1). b(2).

c(1). c(2).

d(2). e(2).

f(3).

p(X) :- a(X).

p(X) :- b(X),c(X),d(X),e(X).

p(X) :- f(X).

?- p(X).

X = 1 ;

X = 2 ;

X = 3.

Backtracking

Exemplu

```
a(1).  
b(1). b(2).  
c(1). c(2).  
d(2). e(2).  
f(3).  
  
p(X) :- a(X).  
p(X) :- b(X),c(X),  
         d(X),e(X).  
p(X) :- f(X).
```

```
[trace] ?- p(X).  
Call: (8) p(_4430) ? creep  
Call: (9) a(_4430) ? creep  
Exit: (9) a(1) ? creep  
Exit: (8) p(1) ? creep  
X = 1 ;  
Redo: (8) p(_4430) ? creep  
Call: (9) b(_4430) ? creep  
Exit: (9) b(1) ? creep  
Call: (9) c(1) ? creep  
Exit: (9) c(1) ? creep  
Call: (9) d(1) ? creep  
Fail: (9) d(1) ? creep  
Redo: (9) b(_4430) ? creep  
Exit: (9) b(2) ?  
...
```

Cut

- În Prolog putem să "tăiem" punctele de decizie din backtracking, ghidând astfel căutarea soluțiilor și eliminând soluții alternative nedorite.
- O "tăietură" (**cut**) se introduce prin **!**.
- Este un predicat (de aritate 0) predefinit în Prolog care poate fi inserat oriunde în corpul unei reguli.
- Execuția subțintei **!** este mereu cu succes.
- De fiecare dată când **!** este întâlnit în corpul unei reguli, sunt **finale** toate alegerile făcute începând cu momentul în care capul acelei reguli a fost unificat cu scopul părinte.

Backtracking

Exemplu

a(1).

b(1). b(2).

c(1). c(2).

d(2). e(2).

f(3).

?- p(X).

X = 1.

p(X) :- a(X).

p(X) :- b(X), c(X), !, d(X), e(X).

p(X) :- f(X).

Backtracking

Exemplu

```
a(1).
b(1). b(2).
c(1). c(2).
d(2). e(2).
f(3).

p(X) :- a(X).
p(X) :- b(X),c(X),!,
        d(X),e(X).
p(X) :- f(X).

[trace] ?- p(X).
Call: (8) p(_4430) ? creep
Call: (9) a(_4430) ? creep
Exit: (9) a(1) ? creep
Exit: (8) p(1) ? creep
X = 1 ;
Redo: (8) p(_4430) ? creep
Call: (9) b(_4430) ? creep
Exit: (9) b(1) ? creep
Call: (9) c(1) ? creep
Exit: (9) c(1) ? creep
Call: (9) d(1) ? creep
Fail: (9) d(1) ? creep
Fail: (8) p(_4430) ? creep
false.
```



$p(X) :- q_1(X), \dots, q_n(X), !, r_1(X), \dots, r_k(X).$

- Mecanismul de backtracking poate fi restricționat folosind !.
- Predicatul ! reușește întotdeauna, dar predicatul părinte eșuează atunci când se încearcă backtracking "peste" !.
- Mecanismul de backtracking funcționează pentru clauzele care se găsesc înainte de ! sau după !
- Alegerile (instanțierile) făcute în execuția unui predicat înainte de a se ajunge la ! nu mai pot fi schimbate.

Backtracking și !

Exemplu

În exemplul nostru, cum putem modifica baza de cunoștințe astfel încât mecanismul de backtracking să ajungă și la ultima alternativă?

$p(X) \text{ :- } a(X).$

$p(X) \text{ :- } b(X), c(X), !, d(X), e(X).$

$p(X) \text{ :- } f(X).$

Backtracking și !

Exemplu

În exemplul nostru, cum putem modifica baza de cunoștințe astfel încât mecanismul de backtracking să ajungă și la ultima alternativă?

$p(X) \text{ :- } a(X).$

$p(X) \text{ :- } b(X), c(X), !, d(X), e(X).$

$p(X) \text{ :- } f(X).$

Pentru a ajunge la ultima alternativă, mecanismul de backtracking trebuie să funcționeze înainte de !:

$a(1). b(2). c(1). d(2). e(2). f(3).$

$?- p(X).$

$X = 1 ;$

$X = 3$

Backtracking și !

- Următorul predicat clasifică un număr folosind clauze care se exclud reciproc:

`range(X, 'A') :- X < 3.`

`range(X, 'B') :- 3 =< X , X < 6.`

`range(X, 'C') :- 6 =< X.`

- Pentru a evita backtracking-ul după ce o clasificare este obținută se poate folosi !

`range(X, 'A') :- X < 3, !.`

`range(X, 'B') :- 3 =< X , X < 6, !.`

`range(X, 'C') :- 6 =< X.`

Acest tip de utilizare se numește **cut verde** și este recomandat pentru a îmbunătăți performanța unui program.

Backtracking și !

- Putem scrie programul anterior astfel:

```
range(X, 'A') :- X < 3, !.  
range(X, 'B') :- X < 6, !.  
range(X, 'C').
```

- Ce se întâmplă dacă eliminăm ! în programul de mai sus?

```
range(X, 'A') :- X < 3, !.  
range(X, 'B') :- X < 6.  
range(X, 'C').
```

```
?- range(4, Cat).  
Cat = 'B' ;  
Cat = 'C'
```

Acest tip de utilizare se numește **cut roșu**, deoarece prezența predicatului **!** afectează logica programului. **Acest mod de utilizare trebuie evitat.**

Negarea unui predicat: $\neg \text{pred}(X)$

Răspunsurile din Prolog

- Pentru a da un răspuns pozitiv la o țintă, Prolog trebuie să construiască o "demonstrație" pentru a arată că mulțimea de fapte și reguli din program implică acea țintă.
- Astfel, răspunsul `true` la o țintă nu înseamnă doar că ținta este adevărată, ci și că este `demonstrabilă`.
- Astfel, un răspuns `false` nu înseamnă neapărat că ținta nu este adevărată, ci doar că `Prolog nu a reușit să găsească o demonstrație`.

Răspunsurile din Prolog

Exemplu

```
animal(dog).  
animal(elephant).  
animal(sheep).
```

```
?- animal(cat).  
false
```

Răspunsurile din Prolog

Exemplu

```
animal(dog).  
animal(elephant).  
animal(sheep).
```

```
?- animal(cat).  
false
```

- Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- Totuși, dacă **specificăm complet o problemă** (adică specificăm toate cazurile posibile), atunci noțiunile de nedemonstrabil și fals coincid. Atunci un false e chiar un fals.

Operatorul \+

- Câteodată poate dori **să negăm o țintă**.
- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.  
neg(Goal)
```

unde **fail/0** este un predicat care eșuează întotdeauna.

Operatorul \+

- Câteodată poate dori să negăm o țintă.

- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.  
neg(Goal)
```

unde **fail/0** este un predicat care eșuează întotdeauna.

- În PROLOG acest predicat este predefinit sub numele \+.
- Operatorul \+ se folosește pentru a nega un predicat.
- O țintă \+ Goal reușește dacă Prolog nu găsește o demonstrație pentru Goal.
- Semantica operatorului \+ se numește **negation as failure**.
- Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.

Operatorul \+

- Câteodată poate dori să negăm o țintă.

- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.  
neg(Goal)
```

unde **fail/0** este un predicat care eșuează întotdeauna.

- În PROLOG acest predicat este predefinit sub numele \+.
- Operatorul \+ se folosește pentru a nega un predicat.
- O țintă \+ Goal reușește dacă Prolog nu găsește o demonstrație pentru Goal.
- Semantica operatorului \+ se numește **negation as failure**.
- Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.

"nevinovat până la proba contrarie"

Negația ca eșec ("negation as failure")

Exemplu

Să presupunem că avem o listă de fapte cu perechi de oameni căsătoriți între ei:

```
married(peter, lucy).  
married(paul, mary).  
married(bob, juliet).  
married(harry, geraldine).
```

Negația ca eșec

Exemplu (cont.)

Putem să definim un predicat `single/1` care reușește dacă argumentul său nu este nici primul nici al doilea argument în faptele pentru `married`.

```
single(Person) :-  
    \+ married(Person, _),  
    \+ married(_, Person).  
  
?- single(mary).    ?- single(anne).    ?- single(X).  
false              true                false
```

Răspunsul la întrebarea `?- single(anne).` trebuie gândit astfel:

*Presupunem că Anne este single,
deoarece **nu am putut demonstra** că este maritată.*

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).  
married(bob, juliet). married(harry, geraldine).  
married(john, anne).
```

```
single(P) :- \+ married(P, _), \+ married(_, P).
```

```
?- single(anne).
```

```
false
```


Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).  
married(bob, juliet). married(harry, geraldine).  
married(john, anne).
```

```
single(P) :- \+ married(P, _), \+ married(_, P).
```

```
?- single(anne).
```

false

Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește **nemonoton**.

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).  
married(bob, juliet). married(harry, geraldine).  
married(john, anne).
```

```
single(P) :- \+ married(P, _), \+ married(_, P).
```

```
?- single(anne).
```

false

Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește **nemonoton**.

- Sistemele logice pe care le-am studiat până acum (calculul cu propoziții clasic, logica de ordinul I, logica clauzelor Horn) sunt **monotone**:
dacă din $\Gamma \vdash \varphi$ și $\Gamma \subseteq \Sigma$ atunci $\Sigma \vdash \varphi$.

Baze de date dinamice

Baze de date(cunoștințe)

- Până acum am folosit baze de cunoștințe **stative**, care fac parte dintr-un program și nu se schimbă pe parcursul programului.
- În Prolog se pot folosi și baze de date **dinamice**, care se schimbă pe durata execuției unui program. Acestea pot fi de două feluri:
 - create la fiecare execuție; dispar la terminarea programului; o astfel de baă de date dinamică poate fi gândită ca *memoria de lucru* (*working memory*) a programului; predicate specifice:
`assert/1`, `asserta/1`, `assertz/1`, `remove/1`, `removeall/1`;
 - memorate în fișiere; nu sunt parte ale unui program particular, pot fi modificate și consultate de mai multe programe; predicate specifice:
`consult/1`, `save/1`, etc.

Baze de date(cunoștințe)

Exemplu

```
?- assert(prop(a)).  
true.
```

```
?- assert(prop(a)), assert(prop(b)), asserta(prop(c)).  
true.
```

```
?- prop(X).  
X = c .
```

```
?- prop(X).  
X = c ;  
X = a ;  
X = a ;  
X = b.
```

Baze de date(cunoștințe)

Exemplu (cont.)

```
?- retract(prop(a)).
```

```
true .
```

```
?- prop(X).
```

```
X = c ;
```

```
X = a ;
```

```
X = b.
```

```
?- assert(prop(a)), retractall(prop(a)).
```

```
true.
```

```
?- prop(X).
```

```
X = c ;
```

```
X = b.
```

```
?- retractall(prop(X)).
```

```
true.
```

```
?- prop(X).
```

```
false.
```

Baze de date(cunoștințe)

Exemplu (cont.)

```
?- assert(prop(a)), assert(prop(b)).  
true.
```

```
?- assert((prop2(X,Y) :- prop(X),prop(Y), X \== Y)).  
true.
```

```
?- prop2(X,Y).  
X = a,  
Y = b ;  
X = b,  
Y = a ;  
false.
```

```
?- listing. % listeaza baza de cunostinte
```

Baze de date. Memoizare

- Pentru a manevra dinamic un predicat deja definit într-un program, acesta trebuie declarat **dynamic**.

Exemplu (Factorialul cu memoizare)

```
:- dynamic fib/2.
```

```
fib(0,0).
```

```
fib(1,1).
```

```
fib(N,V):- N > 0, N1 is (N-1), N2 is (N-2),  
           fib(N1, V1), fib(N2,V2), V is (V1+V2),  
           asserta(fib(N,V)).
```

```
?- fib(50,X).
```

```
X = 12586269025
```


Joc: labirint

Vom implementa un joc de tip labirint:

- labirintul este format din mai multe camere;
- camerele sunt conectate prin uși care sunt desemnate prin poziția lor: east, west, north, south;
- comenzile corespund direcțiilor;
- camera curentă este desemnată printr-un predicat dinamic.

Sursă: <https://cse3521.artifice.cc/prolog-examples.html>

Labirint: baza de cunoștințe

```
:- dynamic current_room/1.
```

```
room(garden, 'Garden', 'You are in the Garden...').  
room(hallway, 'Hallway', 'You are in the Hallway...').  
room(kitchen, 'Kitchen', 'You are in the kitchen...').  
room(library, 'Library', 'You are among many books in the Library...').  
room(lair, 'Lair', 'You have found an apparently quite evil lair...').  
connected(north, library, hallway).  
connected(south, hallway, library).  
connected(down, library, lair).  
connected(up, lair, library).  
connected(west, library, garden).  
connected(east, garden, library).  
connected(west, hallway, kitchen).  
connected(east, kitchen, hallway).
```

Joc: labirint

```
play :- retractall(current_room(_)),  
        assert(current_room(library)),  
        print_location,  
        get_input.
```

```
print_location :- current_room(Current),  
                  room(Current, Name, Description),  
                  write(Description), nl.
```

Joc: labirint

```
play :- retractall(current_room(_)),  
        assert(current_room(library)),  
        print_location,  
        get_input.
```

```
get_input :- read(Input), ge_input(Input).
```

```
get_input(quit).  
get_input(Input) :- process_input(Input),  
                    print_location,  
                    read(Input1),  
                    get_input(Input1).
```

Joc: labirint

```
process_input(Door) :- current_room(Current),  
                        connected(Door, Current, New),  
                        change_room(New).  
  
process_input(_) :- write('No exit that direction.'), nl.  
  
change_room(New) :- current_room(Current),  
                    retract(current_room(Current)),  
                    assert(current_room(New)).
```

Joc: labirint

?- play.

You are among many books in the Library...

|: north.

You are in the Hallway. Dusty broken lamps and flower pots...

|: west.

You are in the kitchen. Knives, pots, pans, ...

|: east.

You are in the Hallway. Dusty broken lamps and flower pots...

|: south.


You are among many books in the Library...

|: down.

You have found an apparently quite evil lair, of all things...

|: quit.

true



Cursul s-a încheiat!
Ultima săptămână: recapitulare.