

Curs ID PROLOG

2018-2019

Programare Logică

Programare logică - în mod idealist

- Un "program logic" este o colecție de proprietăți presupuse (sub formă de formule logice) despre lume (sau mai degrabă despre lumea programului).
- Programatorul furnizează și o proprietate (o formula logică) care poate să fie sau nu adevărată în lumea respectivă (întrebare, *query*).
- Sistemul determină dacă proprietatea aflată sub semnul întrebării este o consecință a proprietăților presupuse în program.
- Programatorul nu specifică metoda prin care sistemul verifică dacă întrebarea este sau nu consecință a programului.

Exemplu de program logic

```
oslo → windy
oslo → norway
norway → cold
cold ∧ windy → winterIsComing
oslo
```

Exemplu de întrebare

Este adevărat `winterIsComing`?

Cuprins

PROLOG

- 1 Elemente de sintaxă
- 2 Programe și întrebări
- 3 Execuția unui program
- 4 Exemplu: colorarea hărților
- 5 Aritmetica în Prolog
- 6 Liste și recursie

Elemente de sintaxă

Putem să testăm în SWI-Prolog

Program:

```
windy :- oslo.  
norway :- oslo.  
cold :- norway.  
winterIsComing :- windy, cold.  
oslo.
```

Intrebare:

```
?- winterIsComing.  
true
```

<http://swish.swi-prolog.org/>

Sintaxă: constante, variabile, termeni compuși

- **Atomii**: `sansa`, `'Jon Snow'`, `jon_snow`
- **Numere**: `23`, `23.03`, `-1`
Atomii și **numerele** sunt **constante**.
- **Variabile**: `X`, `Stark`, `_house`
- Termeni **compuși**: `father(eddard, jon_snow)`,
`and(son(bran, eddard), daughter(arya, eddard))`
 - forma generală: **atom**(**termen**, ..., **termen**)
 - atom-ul care denumește termenul se numește **functor**
 - numărul de argumente se numește **aritate**



Un mic exercițiu sintactic

Care din următoarele șiruri de caractere sunt **constante** și care sunt **variabile** în Prolog?

- ☐ vINCENT – **constantă**
- ☐ Footmassage – **variabilă**
- ☐ variable23 – **constantă**
- ☐ Variable2000 – **variabilă**
- ☐ big_kahuna_burger – **constantă**
- ☐ 'big kahuna burger' – **constantă**
- ☐ big kahuna burger – **nici una, nici alta**
- ☐ 'Jules' – **constantă**
- ☐ _Jules – **variabilă**
- ☐ '_Jules' – **constantă**

Compararea termenilor: $=$, $\backslash=$, $==$, $\backslash==$

$T = U$	reuşeşte dacă există o potrivire (termenii se unifică)
$T \backslash= U$	reuşeşte dacă nu există o potrivire
$T == U$	reuşeşte dacă termenii sunt identici
$T \backslash== U$	reuşeşte dacă termenii sunt diferiţi

Exemplu

?- $X = Y$.

$X = Y$.

?- $X == Y$.

false

?- $p(X, q(Z)) = p(Y, X)$.

$X = Y, Y = q(Z)$.

?- $p(X, Y) == p(X, Y)$.

true

?- $2 = 1 + 1$

false

?- $2 == 1 + 1$

false

- În exemplul de mai sus, $1+1$ este privită ca o expresie, nu este evaluată. Există şi predicate care forţează evaluarea.

Programe și întrebări

Program în Prolog = bază de cunoștințe

Exemplu

Un program în Prolog:

```
father(eddard,sansa).  
father(eddard,jon_snow).
```

```
mother(catelyn,sansa).  
mother(wylla,jon_snow).
```

```
stark(eddard).  
stark(catelyn).
```

```
stark(X) :- father(Y,X), stark(Y).
```



Un program în Prolog este o **bază de cunoștințe** (Knowledge Base).

Program în Prolog = mulțime de predicate

Practic, gândim un program în Prolog ca o mulțime de **predicate** cu ajutorul cărora descriem *lumea* (*universul*) programului respectiv.

Exemplu

```
father(eddard,sansa).  
father(eddard,jon_snow).
```

```
mother(catelyn,sansa).  
mother(wylla,jon_snow).
```

```
stark(eddard).  
stark(catelyn).
```

```
stark(X) :- father(Y,X), stark(Y).
```

Predicate:

father/2

mother/2

stark/1

Un program în Prolog

Program

Fapte + Reguli

Program

- Un **program** în Prolog este format din **reguli** de forma
$$\text{Head} \text{ :- } \text{Body}.$$
- **Head** este un predicat, iar **Body** este o secvență de predicate separate prin virgulă.
- Regulile fără Body se numesc **fapte**.

Exemplu

- Exemplu de regulă: `stark(X) :- father(Y,X), stark(Y).`
- Exemplu de fapt: `father(eddard, jon_snow).`

Interpretarea din punctul de vedere al logicii

- operatorul `:-` este implicația logică \leftarrow

Exemplu

`winterfell(X) :- stark(X).`

dacă `stark(X)` *este adevărat*, *atunci* `winterfell(X)` *este adevărat*.

- virgula `,` este conjuncția \wedge

Exemplu

`stark(X) :- father(Y,X), stark(Y).`

dacă `father(Y,X)` *și* `stark(Y)` *sunt adevărate*,
atunci `stark(X)` *este adevărat*.

Interpretarea din punctul de vedere al logicii

- mai multe reguli cu același Head definesc același predicat, între definiții fiind un sau logic.

Exemplu

```
got_house(X) :- stark(X).  
got_house(X) :- lannister(X).  
got_house(X) :- targaryen(X).  
got_house(X) :- baratheon(X).
```

dacă

stark(X) este adevărat sau
lannister(X) este adevărat sau
targaryen(X) este adevărat sau
baratheon(X) este adevărat,

atunci

got_house(X) este adevărat.

Un program în Prolog

Program

Fapte + Reguli

Cum folosim un program în Prolog?

Întrebări în Prolog



Întrebări și ținte în Prolog

- Prolog poate răspunde la întrebări legate de consecințele relațiilor descrise într-un program în Prolog.
- Întrebările sunt de forma:
$$?- \text{predicat}_1(\dots), \dots, \text{predicat}_n(\dots).$$
- Prolog verifică dacă întrebarea este o consecință a relațiilor definite în program.
- Dacă este cazul, Prolog caută valori pentru variabilele care apar în întrebare astfel încât întrebarea să fie o consecință a relațiilor din program.
- un predicat care este analizat pentru a se răspunde la o întrebare se numește țintă (goal).

Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

- ❑ **false** – în cazul în care întrebarea nu este o consecință a programului.
- ❑ **true** sau **valori pentru variabilele din întrebare** în cazul în care întrebarea este o consecință a programului.

Exemplu

```
?- stark(jon_snow).  
true  
?- stark(wylla)  
false
```

```
?- stark(X)  
X = eddard ;  
X = catelyn ;  
X = sansa ;  
X = jon_snow ;  
false
```

Execuția unui program

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Exemplu

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```

```
X = a.
```

Pentru a răspunde la întrebare se caută o potrivire (unificator) între scopul `foo(X)` și baza de cunoștințe. Răspunsul este substituția care realizează potrivirea, în cazul nostru `X = a`.

Vom discuta detaliat algoritmul de unificare!

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, **Prolog încearcă regulile în ordinea apariției lor.**

Exemplu

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```

```
X = a.
```

```
?- foo(d).
```

```
false
```

Dacă nu se poate face potrivirea, răspunsul este **false**.

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, [Prolog încearcă regulile în ordinea apariției lor.](#)

Exemplu

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```

```
X = a.
```

Dacă dorim mai multe răspunsuri, tastăm ;

```
?- foo(X).
```

```
X = a ;
```

```
X = b ;
```

```
X = c.
```


Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, **Prolog încearcă regulile în ordinea apariției lor.**

Exemplu

Să presupunem că avem programul:

foo(a).

foo(b).

foo(c).

și că punem următoarea întrebare:

?- foo(X).

```
?- trace.  
true.  
  
[trace] ?- foo(X).  
  Call: (8) foo(_4556) ? creep  
  Exit: (8) foo(a) ? creep  
X = a ;  
  Redo: (8) foo(_4556) ? creep  
  Exit: (8) foo(b) ? creep  
X = b ;  
  Redo: (8) foo(_4556) ? creep  
  Exit: (8) foo(c) ? creep  
X = c.
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, **Prolog redenumeste variabilele.**

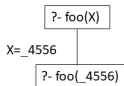
Exemplu

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```



Cum găsește Prolog răspunsul

Pentru a găsi un raspuns, Prolog încearcă regulile în ordinea apariției lor.

Exemplu

Să presupunem că avem programul:

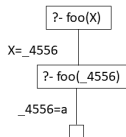
`foo(a).`

`foo(b).`

`foo(c).`

și că punem următoarea întrebare:

`?- foo(X).`



În acest moment, a fost găsită prima soluție: `X=_4556=a`.

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă clauzele în ordinea apariției lor.

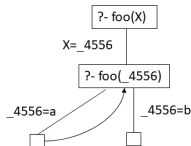
Exemplu

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```



Dacă se dorește încă un răspuns, atunci se face un pas înapoi în arborele de căutare și se încearcă satisfacerea țintei cu o nouă valoare.

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, **Prolog încercă clauzele în ordinea apariției lor.**

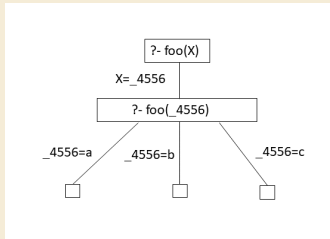
Exemplu

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```



arborele de căutare

Cum găsește Prolog răspunsul

Exemplu

Să presupunem că avem programul:

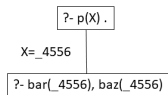
`bar(b) .`

`bar(c) .`

`baz(c) .`

și că punem următoarea întrebare:

`?- bar(X), baz(X) .`



Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o sub-întrebare eșuează.

Exemplu

Să presupunem că avem programul:

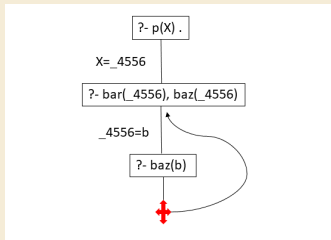
bar(b) .

bar(c) .

baz(c) .

și că punem următoarea întrebare:

`?- bar(X), baz(X) .`



Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o sub-întă eșuează.

Exemplu

Să presupunem că avem programul:

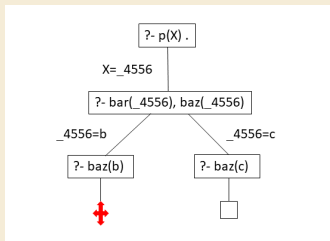
`bar(b) .`

`bar(c) .`

`baz(c) .`

și că punem următoarea întrebare:

`?- bar(X), baz(X) .`



Soluția găsită este: `X=_4556=c`.

Cum găsește Prolog răspunsul

Ce se întâmplă dacă schimbăm ordinea regulilor?

Exemplu

Să presupunem că avem programul:

```
bar(c) .
```

```
bar(b) .
```

```
baz(c) .
```

și că punem următoarea întrebare:

```
?- bar(X), baz(X) .
```

Cum găsește Prolog răspunsul

Ce se întâmplă dacă schimbăm ordinea regulilor?

Exemplu

Să presupunem că avem programul:

```
bar(c).
```

```
bar(b).
```

```
baz(c).
```

și că punem următoarea întrebare:

```
?- bar(X), baz(X).
```

```
X = c ;
```

```
false
```

Vă explicați ce s-a întâmplat? Desenați arborele de căutare!

Exemplu: colorarea hărților

Un program mai complicat

Problema colorării hărților

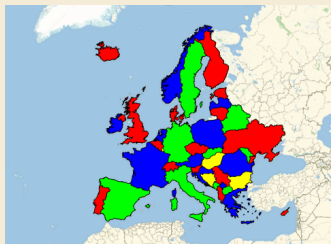
Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

Cum modelăm această problemă în Prolog?

Exemplu

Trebuie să definim:

- ☐ culorile
- ☐ harta
- ☐ constrângerile



Sursa imaginii

Problema colorării hărților

Definim culorile

Exemplu

```
culoare(albastru).  
culoare(rosu).  
culoare(verde).  
culoare(galben).
```

Problema colorării hărților

Definim culorile, harta

Exemplu

```
culoare(albastru).
```

```
culoare(rosu).
```

```
culoare(verde).
```

```
culoare(galben).
```

```
harta(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),  
                               vecin(RO,MD), vecin(RO,BG),  
                               vecin(RO,HU), vecin(UA,MD),  
                               vecin(BG,SE), vecin(SE,HU).
```

Problema colorării hărților

Definim culorile, harta și constrângerile. Cum punem întrebarea?

Exemplu

```
culoare(albastru).
```

```
culoare(rosu).
```

```
culoare(verde).
```

```
culoare(galben).
```

```
harta(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),  
                                vecin(RO,MD), vecin(RO,BG),  
                                vecin(RO,HU), vecin(UA,MD),  
                                vecin(BG,SE), vecin(SE,HU).
```

```
vecin(X,Y) :- culoare(X),  
               culoare(Y),  
               X \== Y.
```

```
?- harta(RO,SE,MD,UA,BG,HU).
```

Problema colorării hărților

Ce răspuns primim?

Exemplu

```
culoare(albastru).
```

```
culoare(rosu).
```

```
culoare(verde).
```

```
culoare(galben).
```

```
harta(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),  
                                vecin(RO,MD), vecin(RO,BG),  
                                vecin(RO,HU), vecin(UA,MD),  
                                vecin(BG,SE), vecin(SE,HU).
```

```
vecin(X,Y) :- culoare(X),  
               culoare(Y),  
               X \== Y.
```

```
?- harta(RO,SE,MD,UA,BG,HU).
```


Problema colorării hărților

Exemplu

```
culoare(albastru).
culoare(rosu).
culoare(verde).
culoare(galben).
harta(RO,SE,MD,UA,BG,HU) :-    vecin(RO,SE), vecin(RO,UA),
                                vecin(RO,MD), vecin(RO,BG),
                                vecin(RO,HU), vecin(UA,MD),
                                vecin(BG,SE), vecin(SE,HU).

vecin(X,Y) :- culoare(X),
               culoare(Y),
               X \== Y.

?- harta(RO,SE,MD,UA,BG,HU).
RO = albastru,
SE = UA, UA = rosu,
MD = BG, BG = HU, HU = verde ■
```

Aritmetica în Prolog

Aritmetica în Prolog

Exemplu

```
?- 3+5 = +(3,5).
```

```
true
```

```
?- 3+5 = +(5,3).
```

```
false
```

```
?- 3+5 = 8.
```

```
false
```

Explicații:

- $3+5$ este un termen.
- Prolog trebuie anunțat explicit pentru a îl evalua ca o expresie aritmetică, folosind predicate predefinite în Prolog, cum sunt `is/2`, `:=/2`, `>/2` etc.

Aritmetica în Prolog

Operatorul `is`:

- Primește două argumente
- Al doilea argument trebuie să fie o expresie aritmetică validă, cu toate variabilele inițializate
- Primul argument este fie un număr, fie o variabilă
- Dacă primul argument este un număr, atunci rezultatul este `true` dacă este egal cu evaluarea expresiei aritmetice din al doilea argument.
- Dacă primul argument este o variabilă, răspunsul este pozitiv dacă variabila poate fi unificată cu evaluarea expresiei aritmetice din al doilea argument.

Pentru a compara două expresii aritmetice, ci operatorul `==`.

Aritmetica în Prolog

Exercițiu. Analizați următoarele exemple:

```
?- 3+5 is 8.
```

```
false
```

```
?= X is 3+5.
```

```
X = 8
```

```
?- 8 is 3+X.
```

```
is/2: Arguments are not sufficiently instantiated
```

```
?- X=4, 8 is 3+X.
```

```
false
```

```
?- (3+4) == (2+5).
```

```
true.
```

Liste și recursie

Listă $[t_1, \dots, t_n]$

- O listă în Prolog este un șir de elemente, separate prin virgulă, între paranteze drepte:

`[1,cold, parent(jon),[winter,is,coming],X]`

- O listă poate conține termeni de orice fel.
- Ordinea termenilor din listă are importanță:

```
?- [1,2] == [2,1] .  
false
```

- Lista vidă se notează `[]`.
- Simbolul `|` desemnează coada listei:

```
?- [1,2,3,4,5,6] = [X|T] .  
X = 1, T = [2, 3, 4, 5, 6] .  
  
?- [1,2,3|[4,5,6]] == [1,2,3,4,5,6] .  
true.
```

Liste

Exercițiu

- Definiți un predicat care verifică că un termen este lista.

```
is_list([]).  
is_list(_|_).
```

- Definiți predicate care verifică dacă un termen este primul element, ultimul element sau coada unei liste.

```
head([X|_],X).
```

```
last([X],X).  
last(_|T,Y):- last(T,Y).
```

```
tail([],[]).  
tail(_|T,T).
```


Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]) .
```

```
member(H, [_|T]) :- member(H,T) .
```

- Definiți un predicat `append/3` care verifică dacă o listă se obține prin concatenarea altor două liste.

```
append([],L,L) .
```

```
append([X|T],L, [X|R]) :- append(T,L,R) .
```

Există predicatele predefinite `member/2` și `append/3`.

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).  
elim(X, [H|T], [H|L]) :- elim(X,T,L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

```
perm([], []). perm([X|T],L) :- elim(X,L,R), perm(R,T).
```

Predicatele predefinite `select/3` și `permutation/2` au aceeași funcționalitate.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

Predicat util:

```
?- name(relay,L). % conversie între atomi și liste  
L = [114, 101, 108, 97, 121]
```

Două abordări posibile:

- ☐ se generează o posibilă, soluție apoi se testează dacă este în KB.
- ☐ se parcurge KB și pentru fiecare termen se testează dacă e soluție.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

```
KB: word(relay).  word(early).  word(layer).
```

```
anagram1(A,B) :- name(A,L), permutation(L,W),  
                 name(B,W), word(B).
```

```
anagram2(A,B) :- name(A,L), word(B),  
                 name(B,W), permutation(L,W).
```

```
?- anagram1(layre,X).
```

```
X = layer ;
```

```
X = relay ;
```

```
X = early ;
```

```
false.
```

```
?- anagram2(layre,X).
```

```
X = relay ;
```

```
X = early ;
```

```
X = layer ;
```

```
false.
```

Exercițiu

- Definiți un predicat `rev/2` care verifică dacă o listă este inversa altei liste.

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

Soluția de mai sus este corectă, dar foarte costisitoare computațional, datorită stilului de programare declarativ.

Cum putem defini o variantă mai rapidă?

O metodă care prin care recursia devine mai rapidă este folosirea **acumulatorilor**, în care se păstrează rezultatele parțiale.

Recursie cu acumulatori

- Varianta inițială:

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

- Varianta cu acumulator

```
rev(L, R) :- revac(L, [], R).
```

% la momentul inițial nu am acumulat nimic.

```
revac([], R, R).
```

% cand lista inițială a fost consumată,

% am acumulat rezultatul final.

```
revac([X|T], Acc, R) :- revac(T, [X|Acc], R).
```

% Acc conține inversa listei care a fost deja parcursă.

- Complexitatea a fost redusă de la $O(n^2)$ la $O(n)$, unde n este lungimea listei.

Recursie

- Multe implementări ale limbajului Prolog aplică " *last call optimization*" atunci când un apel recursiv este ultimul predicat din corpul unei clauze (*tail recursion*).
- Atunci când este posibil, se recomandă utilizare recursiei la coadă (*tail recursion*).
- Vom defini un predicat care generează liste lungi în două moduri și vom analiza performanța folosind predicatul [time/1](#).

```
biglist(0, []).
```

```
biglist(N, [N|T]) :- N >= 1, M is N-1, biglist(M, T), M=M.
```

```
biglist_tr(0, []).
```

```
biglist_tr(N, [N|T]) :- N >= 1, M is N-1, biglist_tr(M, T).
```

Recursie la coadă

- Predicat **fără** recursie la coadă:

```
biglist(0, []).
```

```
biglist(N, [N|T]) :- N >= 1, M is N-1, biglist(M, T), M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, această valoare urmând a fi prelucrată.

```
?- time(biglist(50000, X)).
```

```
100,000 inferences, 0.016 CPU in 0.038 seconds  
(41% CPU, 6400000 Lips)
```

```
X = [50000, 49999, 49998|...] .
```

- Predicatul **cu** recursie la coadă:

```
biglist_tr(0, []).
```

```
biglist_tr(N, [N|T]) :- N >= 1, M is N-1, biglist_tr(M, T).
```

```
?- time(biglist_tr(50000, X)).
```

```
100,000 inferences, 0.000 CPU in 0.007 seconds  
(0% CPU, Infinite Lips)
```

```
X = [50000, 49999, 49998|...]
```




Rezolvați exercițiile din laborator!