Curs PL 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

$$
\begin{aligned}
\texttt{oslo} &\rightarrow \texttt{windy} \\
\texttt{oslo} &\rightarrow \texttt{norway} \\
\texttt{norway} &\rightarrow \texttt{cold} \\
\texttt{cold} \wedge \texttt{windy} &\rightarrow \texttt{winterIsComing} \\
&\quad\ \texttt{oslo}
\end{aligned}
$$

Exemplu de întrebare

Este adevărat `winterIsComing`?

Cuprins

PROLOG

Elemente de sintaxă

Putem să testăm în SWI-Prolog

Program:

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

Intrebare:

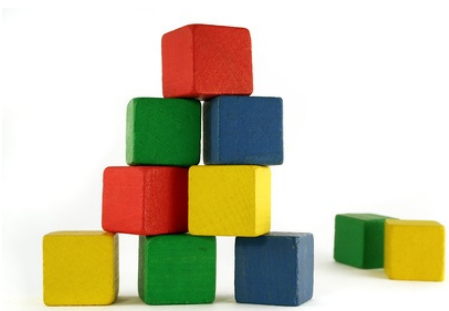
```
?- winterIsComing.
true
```

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

Sintaxă: constante, variabile, termeni compuși

- Atomi: `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: =, \=, ==, \==

`T = U`	reușește dacă există o potrivire (termenii se unifică)
`T \= U`	reușește dacă nu există o potrivire
`T == U`	reușește dacă termenii sunt identici
`T \== U`	reușește dacă termenii sunt diferiți

Exemplu

```
?- X = Y.
X = Y.

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

?- 2 = 1 + 1
false
```

```
?- X == Y .
false

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

?- 2 == 1 + 1
false
```

- În exemplul de mai sus, `1+1` este privită ca o expresie, nu este evaluată. Există și predicate care forțează evaluarea.

Programe și întrebări

Program în Prolog = bază de cunoștințe

Exemplu

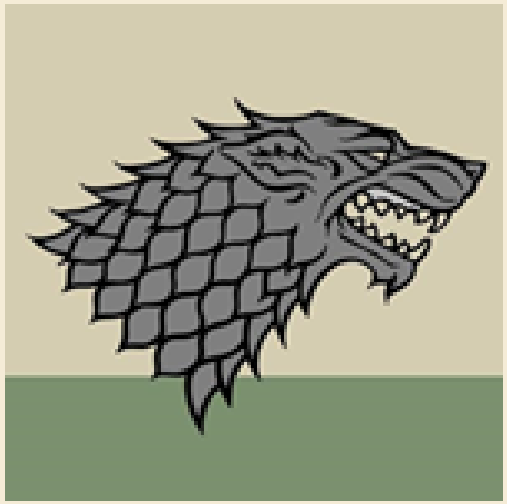
Un program în Prolog:

```
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

- Un program în Prolog este format din reguli de forma

```
Head :- 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 adevřat.

Interpretarea din punctul de vedere al logicii

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

Exemplu

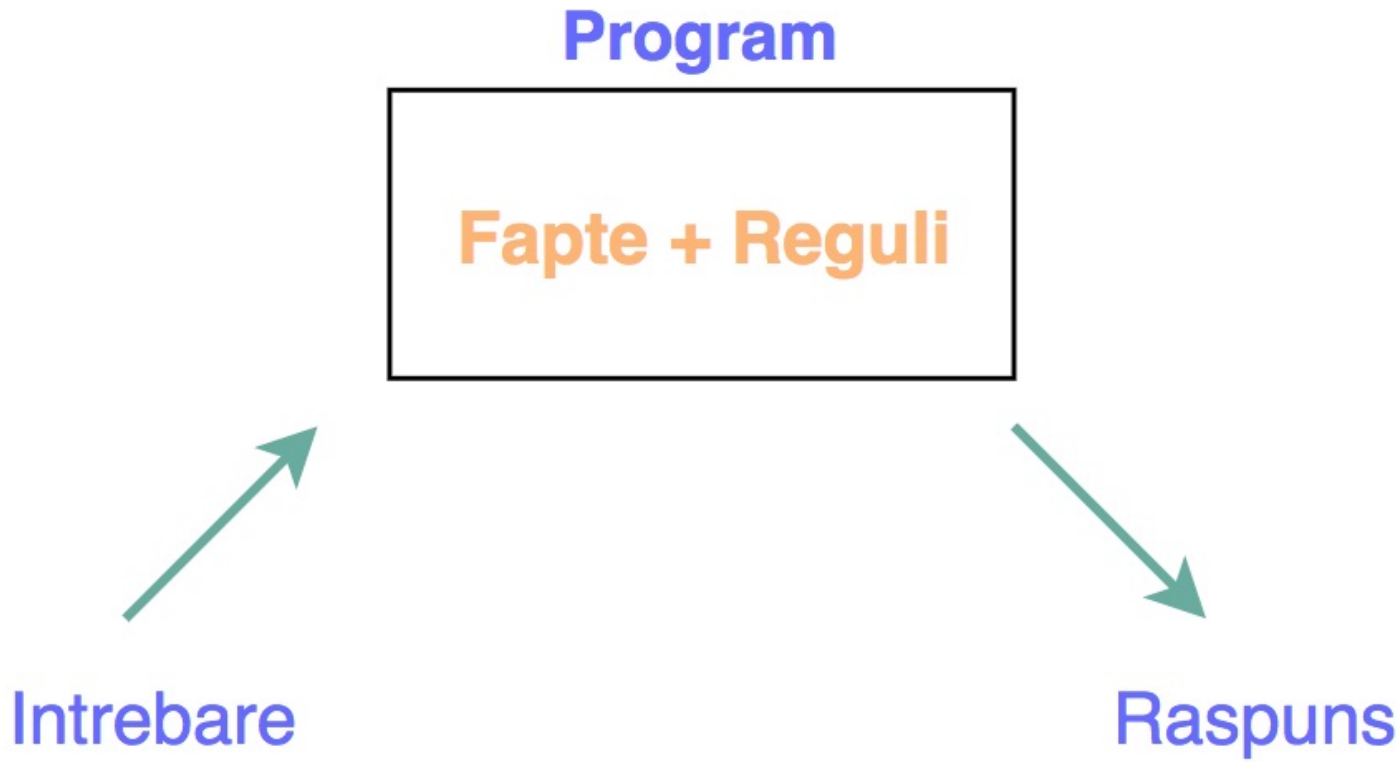
```
got_house(X) :- stark(X).
got_house(X) :- lannister(X).
got_house(X) :- targaryen(X).
got_house(X) :- baratheon(X).
```

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

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:

$$\texttt{?- predicat}_1\texttt{(...),...,predicat}_n\texttt{(...).}$$

- 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 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).
X = a.
```

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

Vom discuta detaliat algoritmul de unificare!

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).
X = a.

?- foo(d).
false
```

Dacaă nu se poate face potrivirea, răspunsul este false.

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).
X = a.
```

Dacă dorim mai multe răspunsuri, tastăm `;`

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

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).
```

```
?- 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 raspuns, Prolog redenumește variabilele.

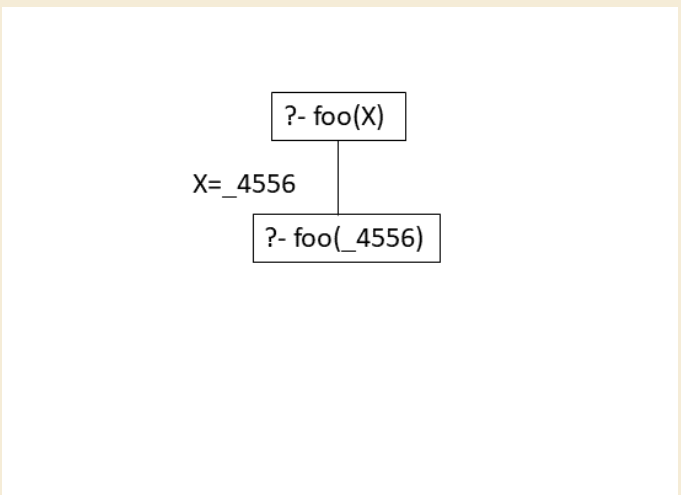
Exemplu

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.

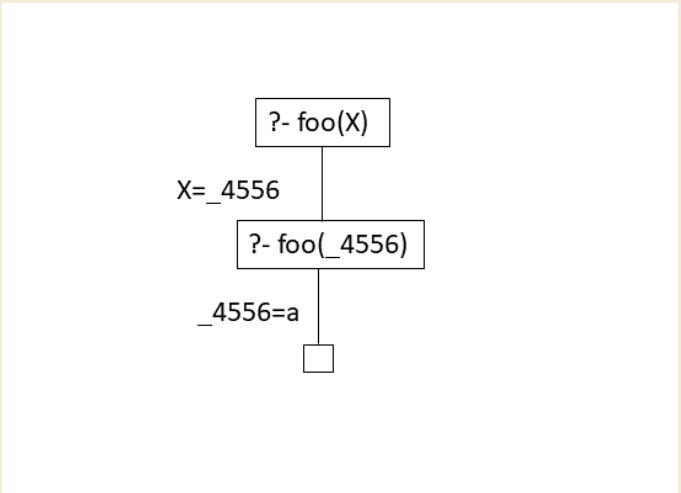
Exemplu

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 raspuns, Prolog încearcă clauzele în ordinea apariției lor.

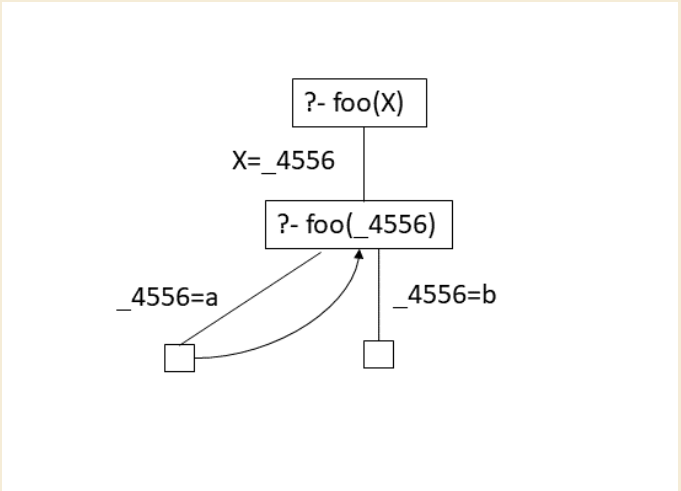
Exemplu

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 raspuns, Prolog încearcă clauzele în ordinea apariției lor.

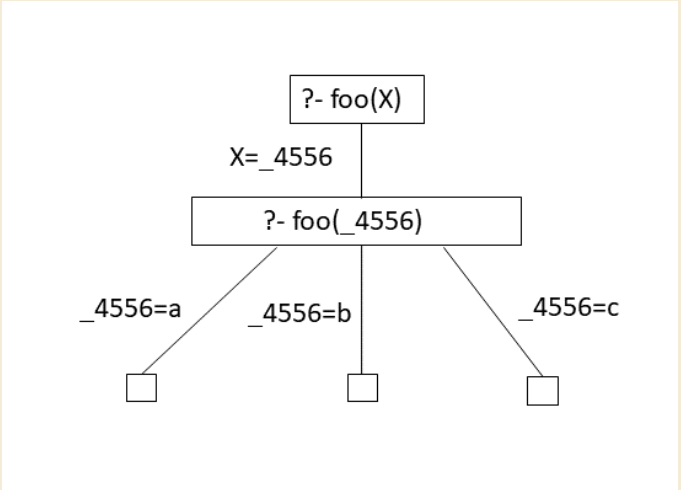
Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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



arborele de căutare

Cum găsește Prolog răspunsul

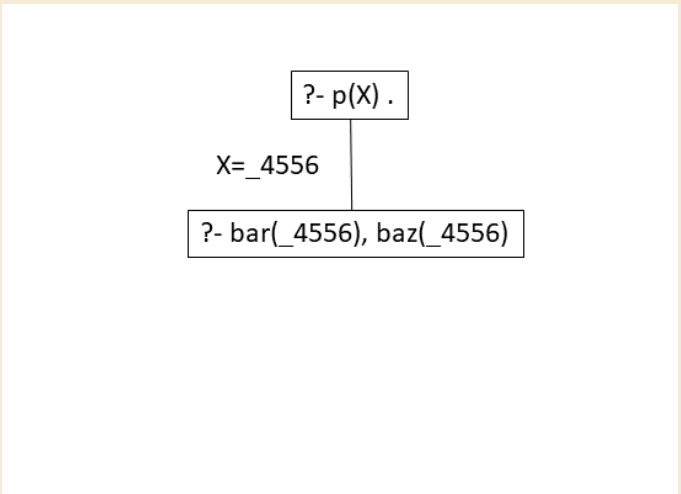
Exemplu

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-țintă eșuează.

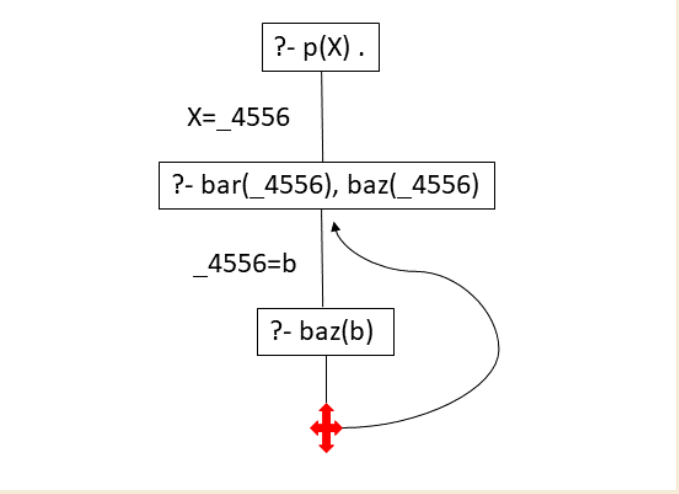
Exemplu

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-țintă eșuează.

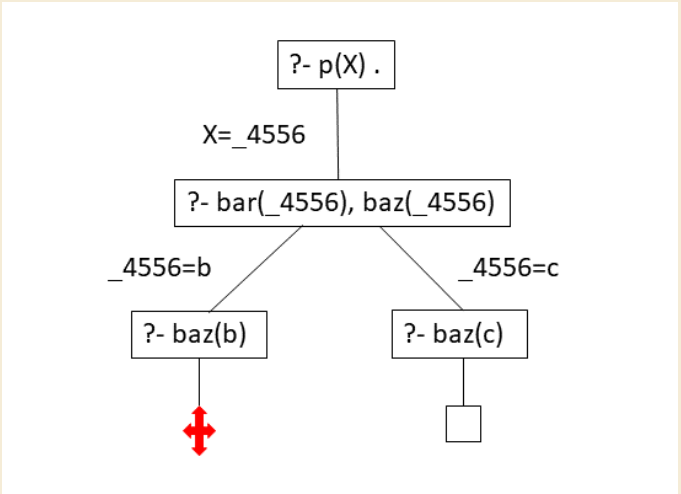
Exemplu

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

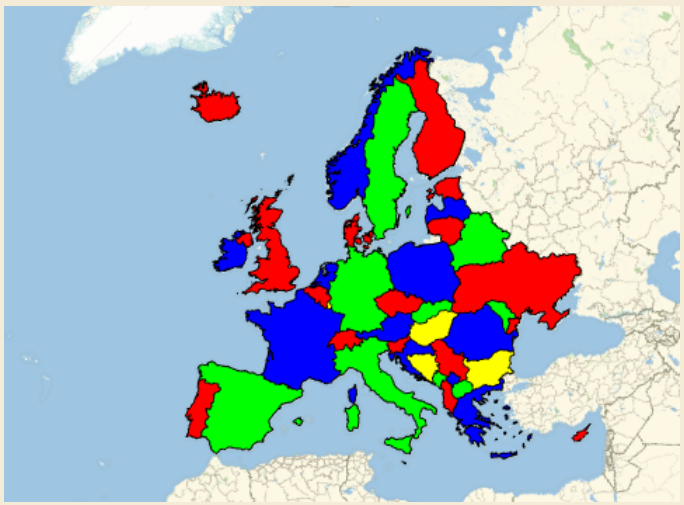
Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

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 [t1,...,tn]

- 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).
```

Liste

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`.

Liste

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 şi 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.
```

Recursie

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 iniţial nu am acumulat nimic.

revac([], R, R).
% cand lista iniţială a fost consumată,
% am acumulat rezultatul final.

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

Liste append/3

- Reamintim definiția funcției append/3:

```
?- listing(append/3).
append([],L,L).
append([X|T],L, [X|R]) :- append(T,L,R).

?- append(X,Y,[a,b,c]).
X = [],
Y = [a, b, c] ;
X = [a],
Y = [b, c] ;
X = [a, b],
Y = [c] ;
X = [a, b, c],
Y = [] ;
false
```

- Funcția astfel definită poate fi folosită atât pentru verificare, cât și pentru generare.

Liste

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

Observăm că funcția append parcurge prima listă.

Am putea rescrie această funcție astfel încât legătura să se facă direct, așa cum putem face în programarea imperativă?

Problema poate fi rezolvată scriind listele ca diferențe, o tehnică utilă în limbajul Prolog.

Liste ca diferențe

- Ideea: lista `[t1,...,tn]` va fi reprezentată printr-o pereche

  `([t1,...,tn|T], T)`

  Această pereche poate fi notată `[t1,...,tn|T]- T`, dar notația nu este importantă.
- Vrem să definim append/3 pentru liste ca diferențe:

  `dlappend((X1,T1),(X2,T2),(R,T)) :- ?.`

```
?- dlappend(([1,2,3|P],P),([4,5|T],T),RD).
P = [4, 5|T],
RD = ([1, 2, 3, 4, 5|T], T).
```

Liste ca diferențe ([t1,...,tn|T], T)

```
dlappend((X1,T1),(X2,T2),(R,T)) :- ?.
```

- Dacă [t1,..., tn] este diferența (X1,T1), iar [q1,..., qk] este diferența (X2,T2) observăm că diferența (R,T) trebuie să fie [t1,...,tn,q1..., qk].
- Obținem R=[t1,...,tn,q1..., qk|T], deci (X1,T1) = (R, P) și (X2,T2) = (P,T) unde P =[q1,...,qk|T]).
- Definiția este:

```
dlappend((R,P),(P,T),(R,T)).
```

```
?- dlappend(([1,2,3|P],P),([4,5|T],T),RD).
P = [4, 5|T],
RD = ([1, 2, 3, 4, 5|T], T).
```

- dlappend este foarte rapid, dar nu poate fi folosit pentru generare, ci numai pentru verificare.

DCG (Definite Clause Grammars)

Structura frazelor

- Aristotel, On Interpretation,
  `http://classics.mit.edu/Aristotle/interpretation.1.1.html`:

  "Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."

- N. Chomsy, Syntactic structure, Mouton Publishers, First printing 1957 - Fourteenth printing 1985 [Chapter 4 (Phrase Structure)]

(i) $Sentence \rightarrow NP + VP$
(ii) $NP \rightarrow T + N$
(iii) $VP \rightarrow Verb + NP$
(iv) $T \rightarrow the$
(q) $N \rightarrow fman, ball$, etc.
(vi) $V \rightarrow hit, took$, etc.

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N
VP	$\rightarrow$	V
VP	$\rightarrow$	V NP
Det	$\rightarrow$	*the*
Det	$\rightarrow$	*a*
N	$\rightarrow$	*boy*
N	$\rightarrow$	*girl*
V	$\rightarrow$	*loves*
V	$\rightarrow$	*hates*

- Neterminalele definesc categorii gramaticale:

  S (propozițiile),
  NP (expresiile substantivale),
  VP (expresiile verbale),
  V (verbele),
  N (substantivele),
  Det (articolele).
- Terminalele definesc cuvintele.

Gramatică independentă de context

GIC

S	$\rightarrow$	NP VP	Det	$\rightarrow$	*the*
NP	$\rightarrow$	Det N	Det	$\rightarrow$	*a*
VP	$\rightarrow$	V	N	$\rightarrow$	*boy*
VP	$\rightarrow$	V NP	N	$\rightarrow$	*girl*
			V	$\rightarrow$	*loves*
			V	$\rightarrow$	*hates*

Ce vrem să facem?

- Vrem să scriem un program în Prolog care să recunoască propozițiile generate de această gramatică.
- Reprezentăm propozițiile prin liste.

```
?- atomic_list_concat(SL,' ' , 'a boy loves a girl').
SL = [a, boy, loves, a, girl]
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

  De exemplu, interpretăm regula S $\rightarrow$ NP VP astfel:

  o propozi tie este o listă `L` care se obține prin concatenarea a două liste, `X` și `Y`, unde `X` reprezintă o expresie substantivală și `Y` reprezintă o expresie verbală.

  ```
  s(L) :- np(X), vp(Y), append(X,Y,L).
  ```

Definirea unei gramatici în Prolog

Gramatică independentă de context

S	$\rightarrow$	NP VP	Det	$\rightarrow$	*the*
NP	$\rightarrow$	Det N	Det	$\rightarrow$	*a*
VP	$\rightarrow$	V	N	$\rightarrow$	*boy*
VP	$\rightarrow$	V NP	N	$\rightarrow$	*girl*
			V	$\rightarrow$	*loves*
			V	$\rightarrow$	*hates*

Prolog

```
s(L) :- np(X), vp(Y),
        append(X,Y,L).
np(L) :- det(X), n(Y),
         append(X,Y,L) .
vp(L) :- v(L).
vp(L):- v(X), np(Y),
        append(X,Y,L) .
```

```
det([the]).
det([a]).
n([boy]).
n([girl]).
v([loves]).
v([hates]).
```

Definirea unei gramatici în Prolog

```
s(L) :- np(X), vp(Y),
        append(X,Y,L).

np(L) :- det(X), n(Y),
         append(X,Y,L) .

vp(L) :- v(L).
vp(L):- v(X), np(Y),
        append(X,Y,L) .

det([the]).
det([a]).
n([boy]).
n([girl]).
v([loves]).
v([hates]).
```

```
?- s([a,boy,loves, a,
girl]).
true .

?- s[a, girl|T].
T = [loves] ;
T = [hates] ;
T = [loves, the, boy] ;
⋮

?- s(S).
S = [the, boy, loves] ;
S = [the, boy, hates] ;
⋮
```

Definirea unei gramatici în Prolog

- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare.
- Pentru a optimiza, folosim *reprezentarea listelor ca diferențe*, plecând de la observația că

  `append(X,Y,L)` este echivalent cu `X = L - Y`
- Regula `s(L) :- np(X), vp(Y), append(X,Y,L)` devine

  `s(L,Z) :- np(L,Y), vp(Y,Z)`
- Această scriere are și următoarea semnificație:
  - fiecare predicat care definește o categorie gramaticală (în exemplu: `s`, `np`, `vp`,`det`, `n`, `v`) are ca argumente o *listă de intrare* `In` și o *listă de ieșire* `Out`
  - predicatul consumă din `In` categoria pe care o definește, iar lista `Out` este ceea ce a rămas neconsumat.

    De exemplu: `np(L,Y)` consumă expresia substantivală de la începutul lui `L`, `v(L,Y)` consumă verbul de la începutul lui `L`, etc.

Definirea unei gramatici în Prolog

```
s(L,M) :- np(L,Y),
          vp(Y,M).
np(L,M) :- det(L,Y),
           n(Y,M).
vp(L,M) :- v(L,M).
vp(L,M):- v(L,Y),
          np(Y,M).
det([the|M],M).
det([a|M],M).
n([boy|M],M).
n([girl|M],M).
v([loves|M],M).
v([hates|M],M).
```

```
?- s([a, boy, loves, a , girl], []).
true.

?- s([a, boy |M], M).
M = [loves|M] ;
M = [hates|M] ;
M = [loves, the, boy|M] ;
...

?- s(L, []).
L = [the, boy, loves] ;
L = [the, boy, hates] ;
...

?- s([X|M], M).
X = the,
M = [boy, loves|M] ;
X = the,
M = [boy, hates|M] ;
...
```

DCG în Prolog

- DCG(Definite Clause Grammar) este o notație introdusă pentru a facilita definirea gramaticilor.
- În loc de s(L,M) :- np(L,Y), vp(Y,M). vom scrie

  ```
  s --> np, vp.
  ```

  iar codul scris anterior va fi generat automat.

Definite Clause Grammar

```
s    -->  np, vp.
np   -->  det, n.
vp   -->  v.
vp   -->  v, np.
```

```
det  -->  [the].
det  -->  [a].
n    -->  [boy].
n    -->  [girl].
v    -->  [loves].
v    -->  [hates].
```

```
?- listing(s).
s(A, B) :- np(A, C), vp(C, B).
```

DCG în Prolog

Definite Clause Grammar

```
s    -->  np, vp.
np   -->  det, n.
vp   -->  v.
vp   -->  v, np.

det  -->  [the].
det  -->  [a].
n    -->  [boy].
n    -->  [girl].
v    -->  [loves].
v    -->  [hates].
```

- Putem pune întrebările ca înainte:

```
?- s([the, girl, hates, the, boy], []).
true.
```

- Putem folosi predicatul phrase/2:

```
?- phrase(s, [the, girl, hates, the, boy]).
true.
```

DCG în Prolog

```
?- phrase(s, [the, girl, hates, the, boy]).
true.

?- phrase(s, X).
X = [the, boy, loves] .
X = [the, boy, loves] ;
X = [the, boy, hates] ;
...


?- phrase(np,X). %toate expresiile substantivale
X = [the, boy] ;
X = [the, girl] ;
X = [a, boy] ;
X = [a, girl].


?- phrase(v,X). % toate verbele
X = [loves] ;
X = [hates].
```

DCG în Prolog

Exemplu

Definiți numerele naturale folosind DCG.

```
nat --> o.
nat --> [s], nat.
```

Definiția generată automat este:

```
?- listing(nat).
nat([o|A], A).
nat([s|A], B) :- nat(A, B).
```

Putem transforma listele în atomi:

```
is_nat(X) :- phrase(nat,Y), atomic_list_concat(Y,'',X).
```

```
?- is_nat(X).
X = o ; X = so ; X = sso ; X = ssso ; X = sssso;
...
```

Tipuri de date compuse

Termeni compuși `f(t1,..., tn)`

- Termenii sunt unitățile de bază prin care Prolog reprezintă datele.
- Sunt de 3 tipuri:
  - Constante: `23, sansa, 'Jon Snow'`
  - Variabile: `X, Stark, _house`
  - Termeni compuși:
    - predicate
    - termeni prin care reprezentăm datele

Exemplu

- `born(john, date(20,3,1977))`
  - `born/2` și `date/3` sunt *functori*
  - `born/2` este un predicat
  - `date/3` definește date compuse

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
  - `void` este arbore
  - `tree(X,A1,A2)` este arbore, unde `X` este un element, iar `A1` și `A2` sunt arbori

`tree(X,A1,A2)` este un termen compus, dar nu este un predicat!

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

- În Prolog putem să definim:

```
def(arb, tree(a, tree(b,
                      tree(d,void,void),
                      void),
                tree(c, void,
                      tree(e,void,void)))).
```

Arbori binari în Prolog

Deoarece în Prolog nu avem declarații explicite de date, pentru a defini arborii vom scrie un predicat care este adevărat atunci când argumentul său este un arbore;

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                         binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                         binary_tree(Right),
                                         element_binary_tree(Element)

element_binary_tree(X):- integer(X). /* de exemplu */

test:- def(arb,T), binary_tree(T).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că un element aparține unui arbore.

```
tree_member(X,tree(X,Left,Right)).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că doi arbori binari sunt izomorfi.

```
isotree(void,void).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
            isotree(Left1,Left2), isotree(Right1,Right2).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
             isotree(Left1,Right2), isotree(Right1,Left2).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

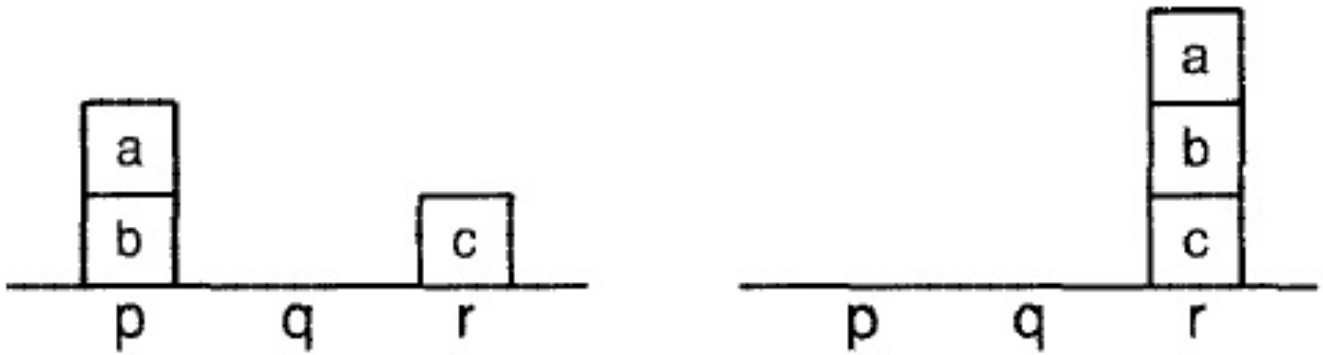
```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).

test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).

?- test(T,P).
T = tree(a, tree(b, tree(d, void, void), void), tree(c,
void, tree(e, void, void))),
P = [a, b, d, c, e]
```

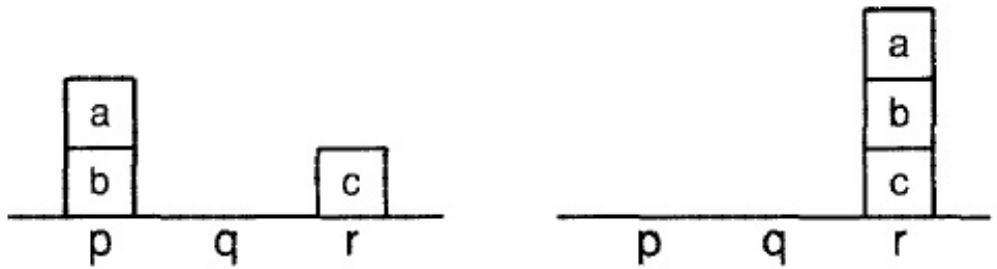
Planning în Prolog

Lumea blocurilor



- Lumea blocurilor este formată din:
  - trei blocuri: `a`,`b`, `c`
  - trei poziții: `p`,`q`, `r`
  - un bloc poate sta peste un alt bloc sau pe o poziție
- Un bloc poate fi mutat pe o poziție liberă sau pe un alt bloc.
- Problema este de a găsi un șir de mutări astfel încât dintr-o stare inițială să se ajungă într-o stare finală.

Lumea blocurilor



- Reprezentarea blocurilor, pozițiilor și a stărilor:

```
block(a). block(b). block(c).
place(p). place(q). place(r).
```

```
initial_state([on(a,b), on(b,p),on(c,r)]).
final_state([on(a,b),on(b,c),on(c,r)]).
```

Observați că `on(a,b)`, `on(b,c)`, etc. sunt date compuse.

- O stare este o listă de termenii de tipul `on(X,Y)`.
Într-o listă care reprezintă o stare, termenii `on(X,Y)` sunt ordonați după prima componentă.

Lumea blocurilor

- Predicatul `transform(State1,State2,Plan)` va genera în variabila `Plan` un șir de mutări permise care transformă starea `State1` în starea `State2`.

```
transform(State1,State2,Plan) :-
                    transform(State1,State2,[State1],Plan).

transform(State,State,Visited,[],_).
transform(State1,State2,Visited,[Action|Actions]) :-
                                legal_action(Action,State1),
                                update(Action,State1,State),
                                   \+ member(State,Visited),
          transform(State,State2,[State|Visited],Actions).
```

- Căutare de tip depth-first.

Lumea blocurilor

- Predicatul `legal_action(Action,State)` va instanția `Action` cu o mutare care poate fi efectuată în starea `State`. Există două mutări posibile: *mutarea pe un bloc* și *mutarea pe o poziție*.

```
legal_action(to_place(Block,Y,Place),State) :-
                        block(Block), clear(Block,State),
                         place(Place), clear(Place,State).

legal_action(to_block(Block1,Y,Block2),State) :-
                      block(Block1), clear(Block1,State),
                        block(Block2), Block1 \== Block2,
                                      clear(Block2,State).


clear(X,State) :- \+ member(on(A,X),State).

on(X,Y,State) :- member(on(X,Y),State).
```

Lumea blocurilor

- Predicatul `update(Action,State,State1)` are următoarea semnificație: făcând mutarea `Action` în starea `State` se ajunge în starea `State1`.

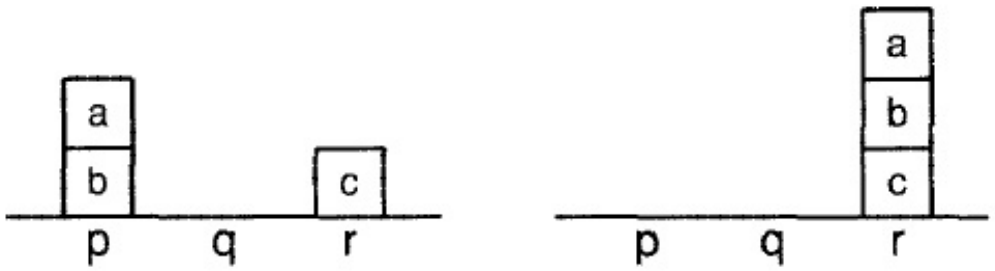
```
update(to_block(X,Y,Z),State,State1) :-
                substitute(on(X,Y),on(X,Z),State,State1).

update(to_place(X,Y,Z),State,State1) :-
                substitute(on(X,Y),on(X,Z),State,State1).

substitute(X,Y,[X|Xs],[Y|Xs]).

substitute(X,Y,[X1|Xs],[X1|Ys]) :- X \= X1,
                                   substitute(X,Y,Xs,Ys).
```

Lumea blocurilor




```
test_plan(Plan) :- initial_state(I), final_state(F),
                                    transform(I,F,Plan).
```

```
?- test(Plan).
Plan = [to_place(a, b, q), to_block(a, q, c),
to_place(b, p, q), to_place(a, c, p), to_block(a, p, b),
to_place(c, r, p), to_place(a, b, r), to_block(a, r, c),
to_place(b, q, r), to_place(a, c, q), to_block(a, q, b),
to_place(c, p, q), to_place(a, b, p), to_block(b, r, a),
to_place(c, q, r), to_block(b, a, c), to_place(a, p, q),
to_block(a, q, b)]
```

Lumea blocurilor

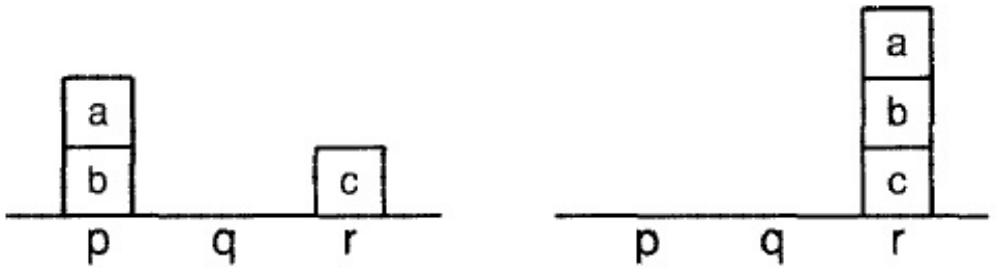
Pentru a obține o soluție mai simplă, putem limita numărul de mutări!

```
transform(State1,State2,Plan, N) :-
                    transform(State1,State2,[State1],Plan,N).


transform(State,State,Visited,[],_).

transform(State1,State2,Visited,[Action|Actions],N) :-
                                  legal_action(Action,State1),
                                  update(Action,State1,State),
        \+ member(State,Visited), length(Visited,M), M < N,
          transform(State,State2,[State|Visited],Actions,N).
```

Lumea blocurilor




```
test_plan(Plan,N) :- initial_state(I), final_state(F),
                                        transform(I,F,Plan,N).
```

```
?- test(Plan,3).
false
?- test(Plan,4).
Plan = [to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]
```

Chatbot: Eliza

Eliza

- Am văzut ca gramaticile DCG pot fi folosite pentru prelucrarea limbajului natural.
- În 1966, J. Weizenbaum (MIT) a scris un articol în care prelucarea limbajului natural se face are la bază recunoaștere șabloanelor (*pattern matching*).
  "ELIZA is a program operating within the MAC time-sharing system at MIT which makes certain kinds of natural language conversation between man and computer possible. Input sentences are analyzed on the basis of decomposition rules which are triggered by key words appearing in the input text. Responses are generated by reassembly rules associated with selected decomposition rules. The fundamental technical problems with which ELIZA is concerned are: 1.the identification of key words, 2.the discovery of minimal context, 3.the choice of appropriate transformations, 4.generation of responses in the absence of keywords, and 5.the provision of an ending capacity for ELIZA "scripts"."

Eliza: o variantă Prolog (cu menționarea autorilor)

```
Hello. I am ELIZA. How can I help you?
> I am upset.
Is it because you are upset that you came to me?
> yes.
You seem quite positive.
> I am not positive.
How long have you been not positive?
> Two days. How can you help me?
Why do you ask?
> I want your help.
What would it mean to you if you got my help?
> Maybe I will feel better.
You do not seem quite certain.
> I am not.
Do you believe it normal to be not?
> I think it is normal to be happy.
You say you think it is normal to be happy
> yes.
You are sure?
> Do you have any advise?
We were discussing you, not me.
```

Eliza: varianta simplă

- Vom implementa o varianta mai simplă:

```
?- eliza.
tell me
|: 'hello'.
Hi there !
|: 'i am very upset'.
How long have you been very upset ?
|: 'two days'.
Please go on .
|: 'can you help me'.
What makes you think I help you ?
|: 'my sister told me'.
Please you tell me more about sister
```

Eliza: varianta simplă

- Programul nostru va trebui să:
  - definească un set de perechi de șabloane, unul pentru intrare și unul pentru ieșire
  - să identifice ce șablon se aplică șirului de intrare
  - să construiască răspunsul pe baza șablonului pereche

```
pattern([i,am,1],['How',long,have,you,been,1,?]).
```

Intrarea:	`'i am very unhappy'`
Șablonul de intrare:	`[i, am, 1]`
Șablonul de ieșire	`['How',long,have,you,been,1,?]`
Ieșirea:	`How long have you been very unhappy?`

- procedeul este mecanic: nu se analizează din punct de vedere sintactic sau semantic frazele!

Eliza: varianta simplă

- Se definesc diferite șabloane:

```
pattern([i,am,1],['How',long,have,you,been,1,?]).

pattern([i,like,1],['Does,anyone,else,in,your,
                                        family,like,1,?]).

pattern([i,feel|_],['Do',you,often,feel,that,way,?]).

,
```

- Trebuie să existe un șablon pentru toate celelalte cazuri:

```
pattern(_,['Please',go,on,'.']).
```

Atenție! folosim numere și nu variable pentru a identifica elementele care lipsesc deoarece acestea pot fi formate din mai mulți atomi: `very unhappy`.

Eliza: varianta simplă

- Alte tipuri de șabloane:

```
pattern(R,['What',makes,you,think,'I',2,you,?]) :-
      member(R,[[i,know,you,2,me], [i,think,you,2,me],
                [i,belive,you,2,me],[can, you, 2, me]]).

pattern(G,AG) :- member(G,[[hi],[hello]]),
                        random_select(AG,[['Hi',there,!],
['Hello',!,'How',are,you,today,?]],_).

pattern([X],['Please',you,tell,me,more,about,X]) :-
                                          important(X).

important(father). important(mother).
important(sister). important(brother).
important(son). important(daughter).
```

Eliza: varianta simplă

- apelul, transformarea atomilor de intrare în liste

```
eliza :- write('tell me'),nl,read(AInput),
         atomic_list_concat(Input,' ',AInput),
         eliza(Input).
```

- programul principal

```
eliza([bye]) :- write(['Goodbye. I hope I have helped you'])

eliza(Input) :- ... /* slide-ul urmator */
```

Eliza: varianta simplă

- programul principal

```
eliza([bye]) :- write(['Goodbye. I hope I have helped you'])

eliza(Input) :- pattern(Stimulus,Response),
                match(Stimulus,Table,Input),
                make(Response,Table,Output),
                reply(Output),
                read(AInput1),
                atomic_list_concat(Input1,' ',AInput1),
                eliza(Input1).
```

Eliza: varianta simplă

Observați perechea

```
match(Stimulus,Table,Input),
make(Response,Table,Output)
```

- predicatul `match` va identifica un șablon în șirul de intrare și va construi o listă de corespondențe. De exemplu, pentru

  Input: `'i am very unhappy'`
  Stimulus: `[i, am, 1]`

  se va introduce în lista de corespondențe perechea `np(1,[very,unhappy])`.

- perdicatul `make` va construi răspunsul corespunzător pe baza listei de corespondențe. În cazul exemplului

  Response: `['How',long,have,you,been,1,?]`
  Output: `How long have you been very unhappy?`

Eliza: varianta simplă

```
match([N|Pattern],Table,Target) :- integer(N),
            lookup(N,Table,LeftTarget), LeftTarget \== [],
                    append(LeftTarget,RightTarget,Target),
                        match(Pattern,Table,RightTarget).

match([N|Pattern],Table,Target) :- integer(N),
  append(LeftTarget,RightTarget,Target), LeftTarget \== [],
       match(Pattern,[nw(N,LeftTarget)|Table],RightTarget).

match([X],_,Target):- member(X,Target),important(X).

match([Word|Pattern],Table,[Word|Target]) :- atom(Word),
                                  match(Pattern,Table,Target).

match([],Table,[]).
```

Eliza: varianta simplă

```
make([N|Pattern],Table, Target) :- integer(N),
                                   lookup(N,Table,LeftTarget),
                                   make(Pattern,Table,RightTarget),
                                   append(LeftTarget,RightTarget,Target).

make([Word|Pattern],Table,[Word|Target]) :- atom(Word),
                                   make(Pattern,Table,Target).

make([],Table,[]).
```

- pentru `N` dat, `lookup` caută perechea `np(N,L)` în lista de corespondențe și întoarce `L`.

Eliza: varianta simplă

```
?- eliza.
tell me
|: 'i am very upset'.
How long have you been very upset ?
|: 'two days'.
Please go on .
|: 'can you help me'.
What makes you think I help you ?
|: 'my sister told me'.
Please you tell me more about sister
|: 'i like her very much'.
Does anyone else in your family like her very much ?
|: 'yes my brother'.
Please you tell me more about brother
|: 'i like teasing him'.
Does anyone else in your family like teasing him ?
|: 'bye'.
Goodbye. I hope I have helped you
true .
```

Bibliografie

Bibliografie:

P. Blackburn, J. Bos, K. Striegnitz, Learn Prolog Now!
http://www.learnprolognow.org/

L.S. Sterling and E.Y. Shapiro, The Art of Prolog
`https://mitpress.mit.edu/books/art-prolog-second-edition`