

# Curs 8

# Cuprins



- 1 Programare logică & Prolog
- 2 Tipuri de date compuse
- 3 Liste și recursie
- 4 Exemplu: reprezentarea unei GIC

# Programare logică & Prolog

# Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.

# Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:

**Program = Logică + Control** (R. Kowalski)

# Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:  
**Program = Logică + Control** (R. Kowalski)
- Programarea logică poate fi privită ca o deducție controlată.

# Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:  
**Program = Logică + Control** (R. Kowalski)
- Programarea logică poate fi privită ca o deducție controlată.
- Un program scris într-un limbaj de programare logică este  
o listă de formule într-o logică  
ce exprimă fapte și reguli despre o problemă.

# Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:  
**Program = Logică + Control** (R. Kowalski)
- Programarea logică poate fi privită ca o deducție controlată.
- Un program scris într-un limbaj de programare logică este  
o listă de formule într-o logică  
ce exprimă fapte și reguli despre o problemă.
- Exemple de limbaje de programare logică:
  - Prolog
  - Answer set programming (ASP)
  - Datalog



# Ce veți vedea la laborator

## Prolog

- ☐ bazat pe logica clauzelor Horn
- ☐ semantica operațională este bazată pe rezoluție
- ☐ este Turing complet
- ☐ vom folosi implementarea [SWI-Prolog](#)

# Ce veți vedea la laborator

## Prolog

- ☐ bazat pe logica clauzelor Horn
- ☐ semantica operațională este bazată pe rezoluție
- ☐ este Turing complet
- ☐ vom folosi implementarea **SWI-Prolog**

Limbajul Prolog este folosit pentru programarea sistemului IBM Watson!



Puteți citi mai multe detalii [aici](#).

Learn Prolog Now!<http://www.let.rug.nl/bos/lpn/>

# 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 program logic

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

### Exemplu de întrebare

Este adevărat `winterIsComing`?

# 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
- ☐ Footmassage
- ☐ variable23
- ☐ Variable2000
- ☐ big\_kahuna\_burger
- ☐ 'big kahuna burger'
- ☐ big kahuna burger
- ☐ 'Jules'
- ☐ \_Jules
- ☐ '\_Jules'



## 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**

# 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  
**Head :- Body.**
- **Head** este un predicat, iar **Body** este o secvență de predicate separate prin virgulă.
- Regulile fără Body se numesc **fapte**.

# 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 ←

## Exemplu

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

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

# 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.

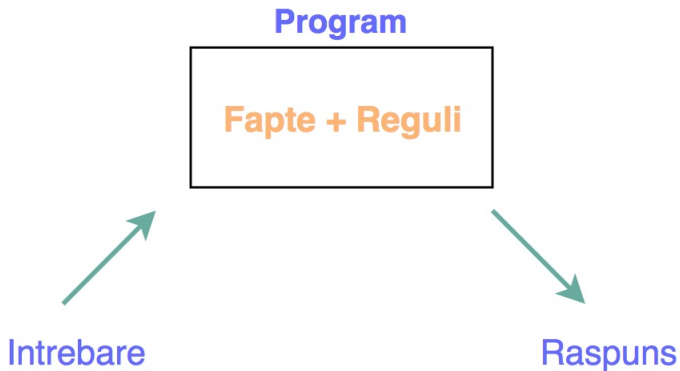
# Un program în Prolog

**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.

# 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
```

# 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`.

Răspunsul la întrebare este găsit prin 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.
```

# 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 raspuns, **Prolog redenumește variabilele**.

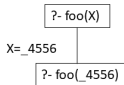
## Exemplu

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 răspuns, **Prolog încearcă regulile în ordinea apariției lor.**

## Exemplu

Să presupunem că avem programul:

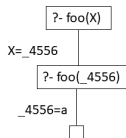
`foo(a) .`

`foo(b) .`

`foo(c) .`

și că punem următoarea întrebare:

`?- foo(X) .`



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.

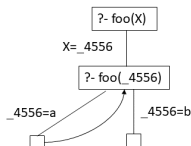
## Exemplu

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

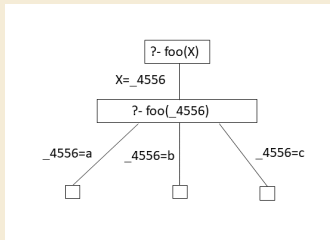
Pentru a găsi un răspuns, **Prolog încearcă clauzele în ordinea apariției lor.**

## 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:

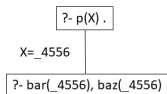
`bar(b) .`

`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-întrebă eșuează.

## Exemplu

Să presupunem că avem programul:

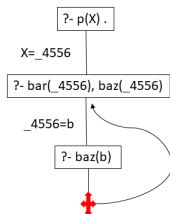
bar(b) .

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-întită eșuează.

## Exemplu

Să presupunem că avem programul:

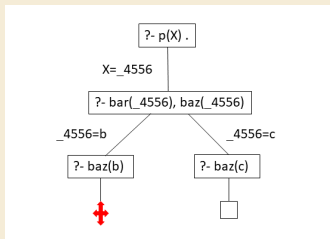
bar(b) .

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
```

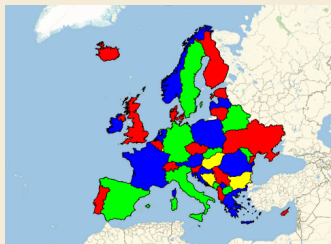
Vă explicați ce s-a întâmplat? Desenați arborele de căutare!

# 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.

## Exemplu



Sursa imaginii

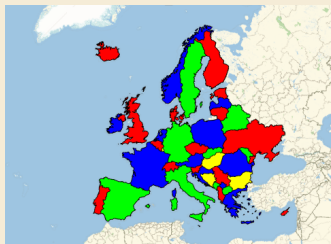
# Un program mai complicat

## 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



Sursa imaginii

# Un program mai complicat

## Problema colorării hărților

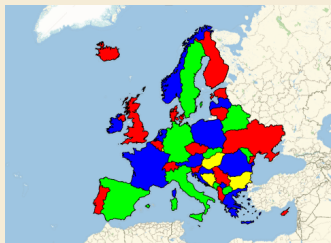
Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

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.

## 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.
```

# 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.
```

# 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 ■
```

## 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

## 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.



## Negarea unui predicat: $\neg$ pred(X)

### Exemplu

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

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

```
false
```

```
?-  $\neg$  animal(cat).
```

```
true
```

## Negarea unui predicat: $\backslash + \text{pred}(X)$

### Exemplu

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

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

```
false
```

```
?-  $\backslash +$  animal(cat).
```

```
true
```

- ❑ Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- ❑ 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, un răspuns **false** nu înseamnă neapărat că ținta nu este adevărată, ci doar că **Prolog nu a reușit să găsească o demonstrație**.

## Operatorul \+

- Negarea unei ținte se poate defini astfel:

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

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

## Operatorul \+

- 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.
- **!(cut)** este un predicat predefinit (de aritate 0) care restricționează mecanismul de backtracking: execuția subțintei ! se termină cu succes, deci alegerile (instanțierile) făcute înaintea de a se ajunge la ! nu mai pot fi schimbate.
- O țintă \+ Goal reușește dacă Prolog nu găsește o demonstrație pentru Goal. Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.
- Semantica operatorului \+ se numește **negation as failure**.

## 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.

## Predicatul $\rightarrow$ /2 (if-then-else)

### □ if-then

`If  $\rightarrow$  Then :- If, !, Then.`

## Predicatul $\rightarrow$ /2 (if-then-else)

### □ if-then

If  $\rightarrow$  Then :- If, !, Then.

### □ if-then-else

If  $\rightarrow$  Then; \_Else :- If, !, Then.

If  $\rightarrow$  Then; Else :- !, Else.

Se încearcă demonstrarea predicatului If. Dacă întoarce true atunci se încearcă demonstrarea predicatului Then, iar dacă întoarce false se încearcă demonstrarea predicatului Else.

`max(X,Y,Z) :- (X =< Y) -> Z = Y ; Z = X`

`?- max(2,3,Z).`

`Z = 3.`



## Predicatul $\rightarrow$ /2 (if-then-else)

### □ if-then

If  $\rightarrow$  Then :- If, !, Then.

### □ if-then-else

If  $\rightarrow$  Then; \_Else :- If, !, Then.

If  $\rightarrow$  Then; Else :- !, Else.

Se încearcă demonstrarea predicatului If. Dacă întoarce true atunci se încearcă demonstrarea predicatului Then, iar dacă întoarce false se încearcă demonstrarea predicatului Else.

```
max(X,Y,Z) :- (X =< Y) -> Z = Y ; Z = X
```

```
?- max(2,3,Z).
```

```
Z = 3.
```

Observăm că If  $\rightarrow$  Then este echivalent cu If  $\rightarrow$  Then ; fail.

## Tipuri de date compuse

# Termeni compuși $f(t_1, \dots, t_n)$

- **Termenii** sunt unitățile de bază prin care Prolog reprezintă datele.
- Sunt de 3 tipuri:
  - **Constante**: 23, sansa, 'Jon Snow'
  - **Variable**: 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

## 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?

## 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:

## 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

# 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



# 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?

# Arbori binari în Prolog

- Cum arată un arbore?

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

# Arbori binari în Prolog

- Cum arată un arbore?

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

- Cum dăm un "nume" arborelui de mai sus?

# Arbori binari în Prolog

- Cum arată un arbore?

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

- Cum dăm un "nume" arborelui de mai sus? Definim un predicat:

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

# Arbori binari în Prolog

- Cum arată un arbore?

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

- Cum dăm un "nume" arborelui de mai sus? Definim un predicat:

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

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.

## Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

## Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

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



## Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

```
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 */
```

## Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

```
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 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).
```

# 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,[]).
```

# 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).
```

# 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]
```

## 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 `|` delimitează 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.**

## Listă $[t_1, \dots, t_n] == [t_1 \mid [t_2, \dots, t_n]]$

- Simbolul  $\mid$  delimitează coada listei:

?-  $[1, 2, 3, 4, 5, 6] = [X \mid T]$ .

$X = 1,$

$T = [2, 3, 4, 5, 6]$ .

- Variabila anonimă  $\_$  este unificată cu orice termen Prolog:

?-  $[1, 2, 3, 4, 5, 6] = [X \mid \_]$ .

$X = 1.$

- Deoarece Prologul face unificare poate identifica șabloane mai complicate:

?-  $[5, 1, 1, 3, 2] = [\_ \mid [X \mid [X \mid \_]]]$ .

$X = 1.$

?-  $[5, 1, 4, 3, 2] = [\_ \mid [X \mid [X \mid \_]]]$ .

false.

## Exercițiu

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

```
is_list([]).
```

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

## 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) .`

## 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) .`

## 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 append/3

□ Funcția 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.



## Exercițiu

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

## 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).
```

## 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 `perm/2` care verifică dacă două liste sunt permutări.

## 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 `perm/2` care verifică dacă două liste sunt permutări.

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

## 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 `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]
```

# 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).
```



# 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).
```

# 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).
```

# 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.

## 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.
```

# 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.
```

# 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`.

# 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.

# 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|...] .
```

# 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).
```

## 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|...]
```

## Exemplu: reprezentarea unei GLC

# 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."



# 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. Chomsky, 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	→	NP VP
NP	→	Det N
VP	→	V
VP	→	V NP
Det	→	<i>the</i>
Det	→	<i>a</i>
N	→	<i>boy</i>
N	→	<i>girl</i>
V	→	<i>loves</i>
V	→	<i>hates</i>

- 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 → NP VP  
NP → Det N  
VP → V  
VP → V NP

Det → *the*  
Det → *a*  
N → *boy*  
N → *girl*  
V → *loves*  
V → *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]

## 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]).`

## 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.

# 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 propoziție 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 → NP VP  
NP → Det N  
VP → V  
VP → V NP

Det → *the*  
Det → *a*  
N → *boy*  
N → *girl*  
V → *loves*  
V → *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

- Reprezentăm propozițiile prin liste.

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]).

# 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.

# 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.
- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare. Pentru a optimiza, folosim reprezentarea listelor ca diferite.

# Bibliografie

- M. Ben-Ari, **Mathematical Logic for Computer Science**, Springer, 2012.
- P. Blackburn, J. Bos, K. Striegnitz, **Learn Prolog now**, College Publications, 2006.
- J.W. Lloyd, **Foundations of Logic Programming**, Springer, 1987.
- L.S. Sterling and E.Y. Shapiro, **The Art of Prolog** <https://mitpress.mit.edu/books/art-prolog-second-edition>
- **Logic Programming**, The University of Edinburgh, <https://www.inf.ed.ac.uk/teaching/courses/lp/>



Pe săptămâna viitoare!