

# Curs 4

# Cuprins

- 1 Prolog. Liste (continua)
- 2 Prolog. Tipuri de date compuse
- 3 Planning în Prolog
- 4 Prolog. Reprezentarea unei GIC (optional)
- 5 Prolog. Mai multe despre liste (optional)

## Prolog. Liste (continuare)

# 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.

## Prolog. 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

## Example

- `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 dacă un element aparține unui arbore.



# Arbori binari în Prolog

## Exercițiu

Scrieți un predicat care verifică dacă 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.

# 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).
```

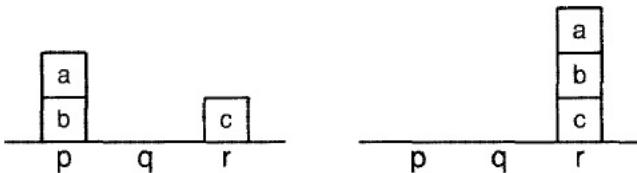
```
?- test(T,P).
```

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

```
P = [a, b, d, c, e]
```

## Planning în Prolog

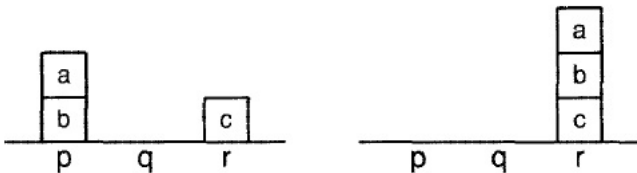
## Problemă: Lumea blocurilor



□ Lumea blocurilor este formată din:

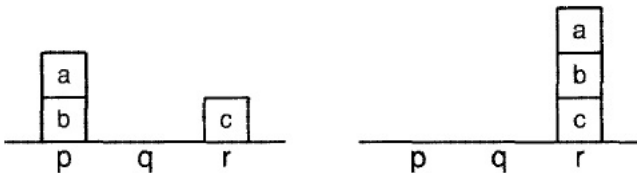
- trei blocuri: a, b, c
- trei poziții: p, q, r
- un bloc poate sta peste un alt bloc sau pe o poziție

## Problemă: 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.

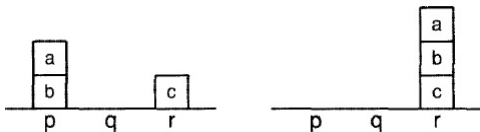
## Problemă: Lumea blocurilor



- 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

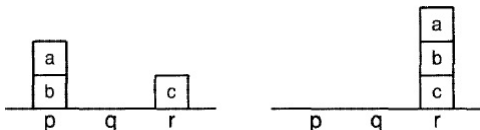


- Reprezentarea blocurilor, pozițiilor și a stărilor:

`block(a). block(b). block(c).`

`place(p). place(q). place(r).`

# Lumea blocurilor



- Reprezentarea blocurilor, pozițiilor și a stărilor:

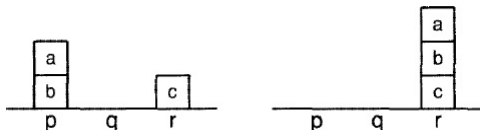
`block(a). block(b). block(c).`

`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)]).`

# Lumea blocurilor



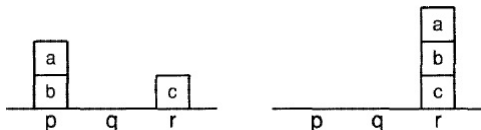
- Reprezentarea blocurilor, pozițiilor și a stărilor:

```
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.

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

# Lumea blocurilor

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

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

# Lumea blocurilor

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

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

- În modelarea noastră, `valid_plan_aux/4` este un predicat auxiliar, cu ajutorul căruia reținem stările "vizitate".
- 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.

# 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.

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



# 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.

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

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

# 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.

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

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

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

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

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

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

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

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

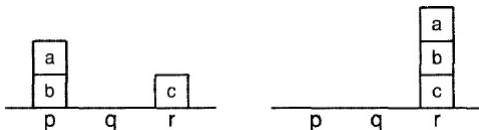
```
update(to_block(X,Z), State, State1) :-  
    substitute(on(X,_), on(X,Z), State, State1).
```

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

- `substitute(X,Y,L,R)` substituie `X` cu `Y` în lista `L`, rezultatul fiind `R`.

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

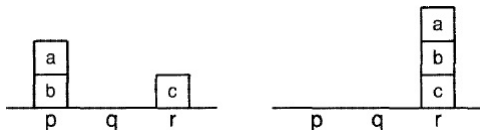
## Lumea blocurilor



```
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)]).
```

## Lumea blocurilor



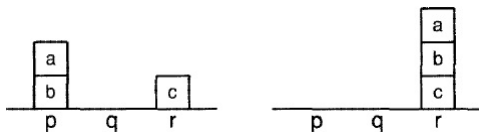
```
block(a). block(b). block(c).  
place(p). place(q). place(r).
```

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

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

```
?- test(Plan).
```

## Lumea blocurilor



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

```
?- test(Plan).
```

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



## Lumea blocurilor

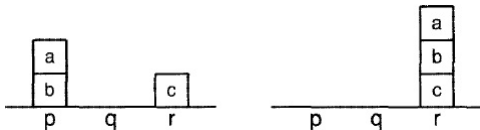
Pentru a obține o soluție mai simplă, putem limita numărul de mutări!

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

```
valid_plan_aux(State,State,_,[],_).
```

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

## Lumea blocurilor



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

```
?- test(Plan,3).
```

false

```
?- test(Plan,4).
```

```
Plan = [to_place(a, q), to_block(b, c), to_block(a, b)]
```

## În general

- Predicatul `valid_plan(State1,State2,Plan)` generează, printr-o căutare de tip depth-first, în variabila Plan un șir de mutări permise care transformă starea State1 în starea State2.

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

- Reprezentarea stărilor, a acțiunilor, a soluției depinde de problema concretă pe care o rezolvăm.

## Prolog. Reprezentarea unei GIC (optional)

# 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)  $N \rightarrow man, ball, etc.$
  - (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]

- 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.

## Prolog. Mai multe despre liste (optional)



## 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).
```

## Exercițiu

Definiți `prefix/2` și `suffix/2` folosind `append`.

# Liste

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

## Exercițiu

Definiți `prefix/2` și `suffix/2` folosind `append`.

```
prefix(P,L) :- append(P,_, L).  
suffix(S,L) :- append(_,S,L).
```

# Liste

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

## Exercițiu

Definiți `prefix/2` și `suffix/2` folosind `append`.

```
prefix(P,L) :- append(P,_, L).  
suffix(S,L) :- append(_,S,L).
```

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?

# Liste

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

## Exercițiu

Definiți `prefix/2` și `suffix/2` folosind `append`.

```
prefix(P,L) :- append(P,_, L).  
suffix(S,L) :- append(_,S,L).
```

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  $[t_1, \dots, t_n]$  va fi reprezentată printr-o pereche

$$([t_1, \dots, t_n | T], T)$$

Această pereche poate fi notată  $[t_1, \dots, t_n | T] - T$ , dar notația nu este importantă.

## Liste ca diferențe

- Ideea: lista  $[t_1, \dots, t_n]$  va fi reprezentată printr-o pereche

$([t_1, \dots, t_n | T], T)$

Această pereche poate fi notată  $[t_1, \dots, t_n | T] - T$ , dar notația nu este importantă.

- Vrem să definim append/3 pentru liste ca diferențe:

$\text{dlappend}((X_1, T_1), (X_2, T_2), (R, T)) \text{ :- } ?.$

?-  $\text{dlappend}([1, 2, 3 | P], P, [4, 5 | T], T), \text{RD}.$

$P = [4, 5 | T],$

$\text{RD} = ([1, 2, 3, 4, 5 | T], T).$

## Liste ca diferențe ( $[t_1, \dots, t_n | T], T$ )

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



## Liste ca diferențe ( $[t_1, \dots, t_n | T], T$ )

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

- Dacă  $[t_1, \dots, t_n]$  este diferența  $(X1,T1)$ , iar  $[q_1, \dots, q_k]$  este diferența  $(X2,T2)$  observăm că diferența  $(R,T)$  trebuie să fie  $[t_1, \dots, t_n, q_1, \dots, q_k]$ .

## Liste ca diferențe ( $[t_1, \dots, t_n | T], T$ )

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

- Dacă  $[t_1, \dots, t_n]$  este diferența  $(X1, T1)$ , iar  $[q_1, \dots, q_k]$  este diferența  $(X2, T2)$  observăm că diferența  $(R, T)$  trebuie să fie  $[t_1, \dots, t_n, q_1, \dots, q_k]$ .
- Obținem  $R = [t_1, \dots, t_n, q_1, \dots, q_k | T]$ , deci  $(X1, T1) = (R, P)$  și  $(X2, T2) = (P, T)$  unde  $P = [q_1, \dots, q_k | T]$ .

## Liste ca diferențe ( $[t_1, \dots, t_n | T]$ , $T$ )

$dlappend((X_1, T_1), (X_2, T_2), (R, T)) :- ?$ .

- Dacă  $[t_1, \dots, t_n]$  este diferența  $(X_1, T_1)$ , iar  $[q_1, \dots, q_k]$  este diferența  $(X_2, T_2)$  observăm că diferența  $(R, T)$  trebuie să fie  $[t_1, \dots, t_n, q_1, \dots, q_k]$ .
- Obținem  $R = [t_1, \dots, t_n, q_1, \dots, q_k | T]$ , deci  $(X_1, T_1) = (R, P)$  și  $(X_2, T_2) = (P, T)$  unde  $P = [q_1, \dots, q_k | T]$ .
- Definiția este:

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

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

$P = [4, 5 | T]$ ,

$RD = [1, 2, 3, 4, 5 | T], T$ .

## Liste ca diferențe ( $[t_1, \dots, t_n | T], T$ )

$dlappend((X_1, T_1), (X_2, T_2), (R, T)) :- ?$ .

- Dacă  $[t_1, \dots, t_n]$  este diferența  $(X_1, T_1)$ , iar  $[q_1, \dots, q_k]$  este diferența  $(X_2, T_2)$  observăm că diferența  $(R, T)$  trebuie să fie  $[t_1, \dots, t_n, q_1, \dots, q_k]$ .
- Obținem  $R = [t_1, \dots, t_n, q_1, \dots, q_k | T]$ , deci  $(X_1, T_1) = (R, P)$  și  $(X_2, T_2) = (P, T)$  unde  $P = [q_1, \dots, q_k | T]$ .
- 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.

# 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|...]
```



Pe săptămâna viitoare!