PARADIGME DE PROGRAMARE

Curs 10

Limbajul Prolog.

Programare logică în Prolog

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Descriere generală (I)

Prolog = logică cu predicate de ordinul întâi + restricții

- **Propoziții = clauze Horn** – particularizate în
  - **Fapte** de forma true => propoziție (se va scrie doar propoziție.)

```
om(gica). om(ilie). impiedicat(gica).
traverseaza(ilie, santier). %% un șantier anume (identificat prin constanta santier)
sapa_groapa(ilie, gica).    %% o groapă oarecare (nu trebuie identificată)
```

  - **Reguli** de forma propoziție$_1$ Λ … Λ propoziție$_n$ => propoziție
    (se folosește scrierea propoziție :- propoziție$_1$, … , propoziție$_n$.)

```
cade_in_groapa(X) :- impiedicat(X), traverseaza(X, santier).
cade_in_groapa(X) :- sapa_groapa(X, Y), X \= Y.
```

Descriere generală (II)

Prolog = logică cu predicate de ordinul întâi + restricții

- **Ipoteza lumii închise** = există adevăruri doar în program
  (tot ce nu poate fi demonstrat în program este fals)
  - Se rezolvă problema semidecidabilității
  - Negația din Prolog ≠ negația logică
    - \+p în Prolog = programul curent nu poate demonstra p

```
bun(X) :- \+sapa_groapa(X,_).          %% false, fiindca sapa_groapa e satisfiabil
om_bun(X) :- om(X), \+sapa_groapa(X,_). %% X = gica
```

    - ¬p în LPOI = p este fals

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Elemente de sintaxă

- **Constante:** `1, 2, ilie, gica` (valoare sau identificator care începe cu literă mică)
- **Variabile:** `X, List, Carte, _` (identificator care începe cu majusculă sau _)
- **Funcții:** `+, -, mod, div, abs` (puține! unitatea de bază e propoziția, nu funcția)
- **Structuri:** (modelează proprietățile și relațiile obiectelor)

```
carte(titlu('Magicianul'), autor('John Fowles'))
sapa_groapa(ilie, gica)
```

- **Conective:** `, ;` (virgulă = Λ, punct și virgulă = V)
  (virgula are prioritate mai mare)

Sintaxă

termen = constantă | variabilă | structură
(obiect cu componente)

clauză = fapt | regulă

```
sapa_groapa(ilie, gica).
```

fapt = structură.

```
om_bun(X) :- om(X), \+sapa_groapa(X,_).
```

regulă = antet :- corp.
antet = structură
corp = structură | structură, corp

Atenție: Faptele și regulile trebuie să se termine cu semnul „.” (punct)

Sintaxă liste

[ ]	- lista vidă
[X | Rest]	- lista cu head-ul X și tail-ul Rest
[_ | Rest]	- lista cu un head a cărui valoare e irelevantă și tail-ul Rest
[X, Y | Rest]	- lista formată din 2 elemente X și Y urmate de lista Rest
[a, B, c]	- lista cu 3 elemente dintre care primul și ultimul sunt fixate la constantele a și c

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Semantică

Propoziții adevărate

- **Faptele** din program
- **Regulile** din program
- **Propozițiile derivabile din fapte și reguli** (via reducere la absurd folosind Rezoluție)

Scop = propoziție (cu sau fără variabile) care trebuie demonstrată

Interogare = solicitare de satisfacere a unui scop
(cu precizarea eventualelor variabile legate în acest proces)

Exemple

```
?- cade_in_groapa(ilie).
true.
```

```
?- cade_in_groapa(gica).
false.
```

```
?- cade_in_groapa(X).
X = ilie.
```

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Inferență

- Programul este doar o colecție de fapte și reguli
- Procesul de inferență este **încorporat în limbaj (ascuns de utilizator)** și constă în
  - **Backward chaining** – folosește doar propozițiile care pot duce la satisfacerea scopului
  - **DFS** – (sub)scopurile nou adăugate în stivă sunt primele pe care încearcă să le satisfacă
  - **Backtracking** – scopurile vizitate de DFS (prin găsirea unui mod de a le satisface și adăugarea subscopurilor rezultate în stivă) sunt revizitate (se explorează toate modurile diferite de a satisface un anumit scop, nu doar primul)
  - **Unificare** – o cale de satisfacere a unui scop este marcată printr-o serie de unificări ale (sub)scopurilor cu faptele și concluziile regulilor

Exemplu (la calculator)

```
?- titlu(X).
X = 'Razboi si pace' ;  %% o primă satisfacere a scopului titlu(X)
X = 'Sonata Kreutzer' ; %% ; cere o resatisfacere (se obține grație bkt)
X = 'Magicianul'.       %% . de la utilizator cere oprirea aici
                        %% . de la Prolog înseamnă că s-au terminat soluțiile
```

Inferență – Idei de implementare

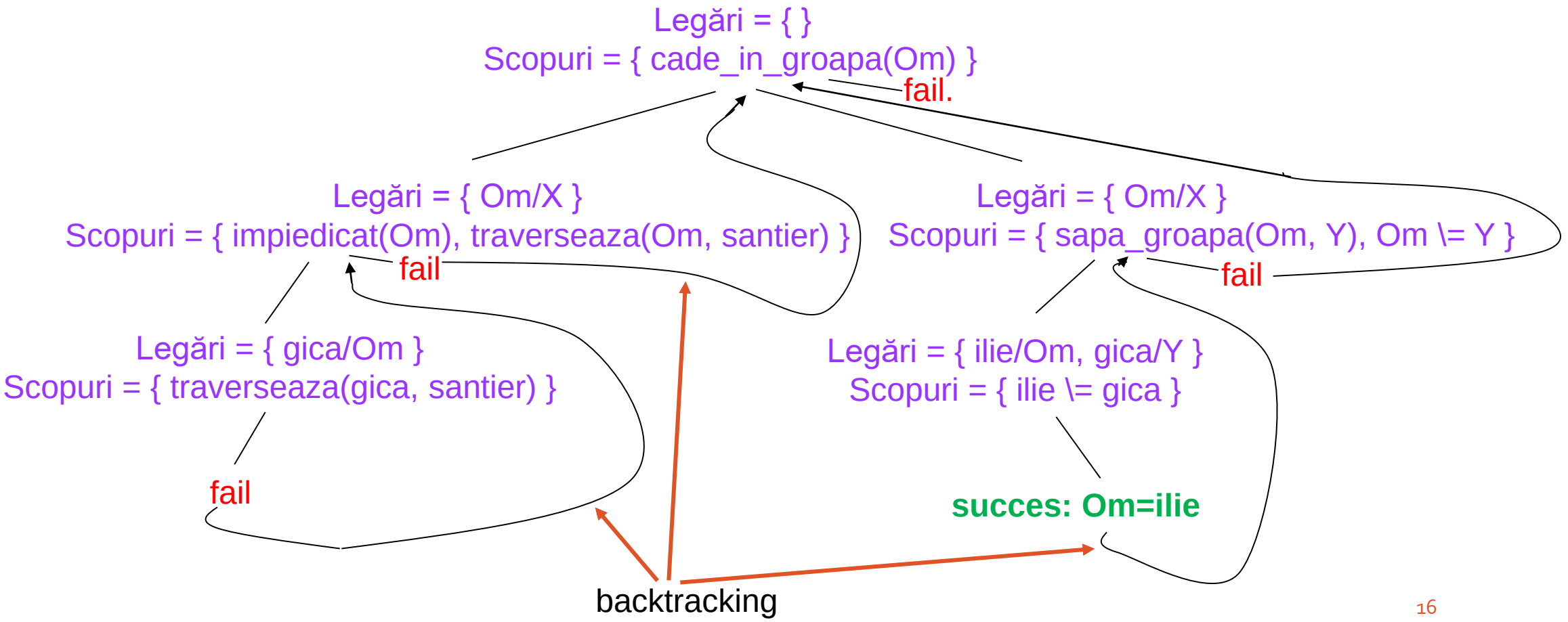
- Algoritmul de inferență pe bază de unificare folosește
  - O **stivă Scopuri** – pornește cu scopul (scopurile) din interogare
  - O **substituție Legări** (mulțime de legări pentru variabilele din scopuri și subscopuri) – inițial vidă
- La fiecare iterație a algoritmului
  - Se extrage un scop din stivă
  - Dacă acesta unifică cu concluzia vreunei reguli (sau vreun fapt) prin substituția S
    - se adaugă S la Legări
    - se adaugă premisele regulii la Scopuri
    - se continuă cu noi scopuri din stivă până se golește stiva (succes) sau unificarea eșuează
  - Backtracking pentru a încerca alte variante

Inferență – Algoritm

```
backward_chaining(Clauze, Scopuri, Legări)
    if Scopuri == [ ]
       success //adaugă la soluții
       return

    scop = head(Scopuri); Scopuri = tail(Scopuri)

    for_each clauză in Clauze   //bkt
       if unifică(scop, antet(clauză), Legări, S))   //S = substituția întoarsă de unificare
            Subst = Legări U S
            Stivă = append(corp(clauză), Scopuri)  // DFS (noile scopuri la începutul stivei)
            backward_chaining(Clauze, Stivă, Subst)
```

Exemplu

```
impiedicat(gica).
traverseaza(ilie, santier).
sapa_groapa(ilie, gica).
cade_in_groapa(X) :- impiedicat(X), traverseaza(X, santier).
cade_in_groapa(X) :- sapa_groapa(X, Y), X \= Y.
```



Declarativ versus procedural

```
predecessor(Parent, Child) :- parent(Parent, Child).
predecessor(Pred, Succ) :- parent(Pred, Child), predecessor(Child, Succ).
```

Semnificația declarativă

- Interesează obiectele și relațiile definite în program

  Pred este predecesorul lui Succ dacă Pred este părintele lui Child și Child este predecesorul lui Succ.
- Determină care va fi rezultatul
- Nu contează ordinea clauzelor și a premiselor în reguli

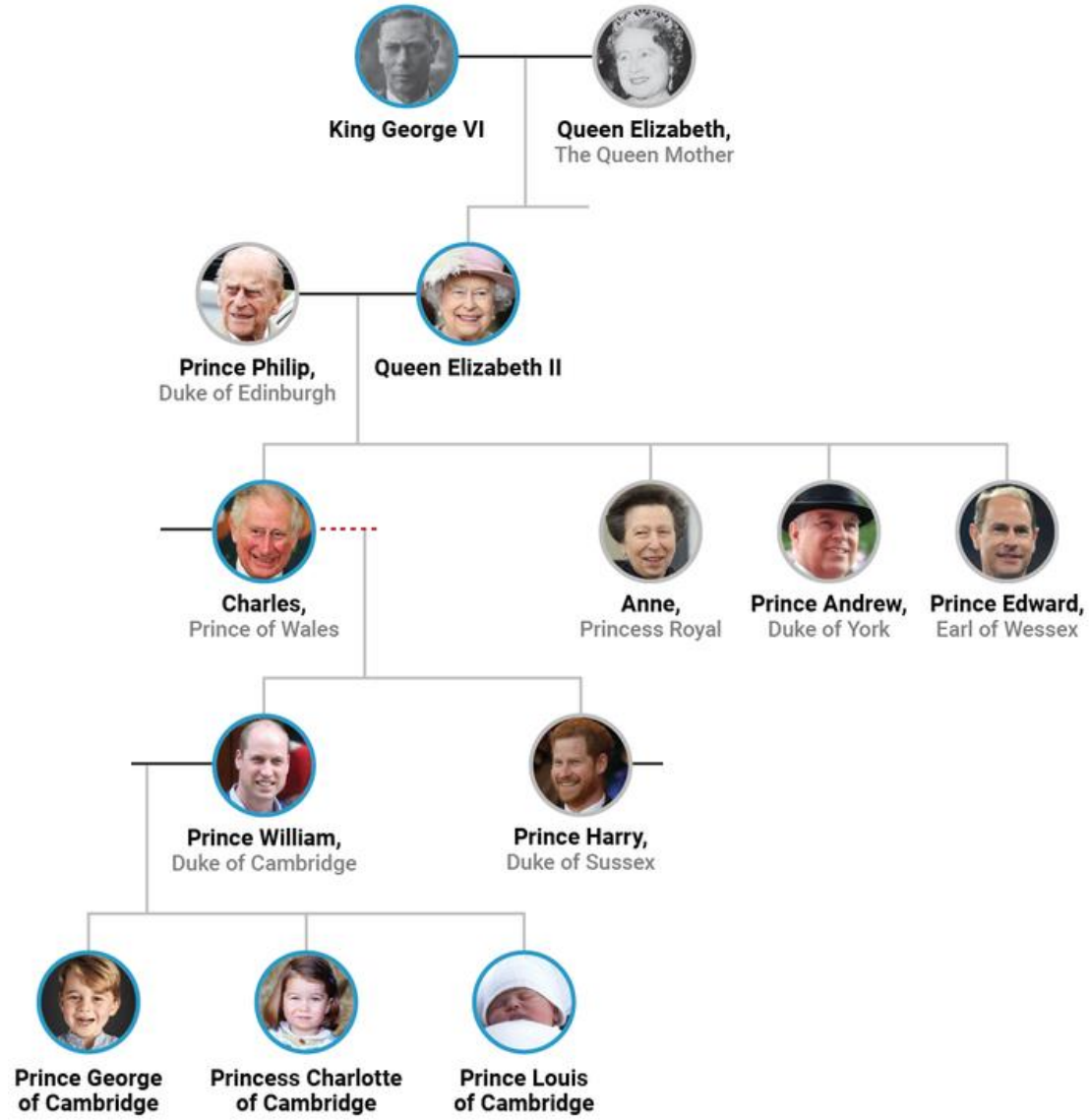
Semnificația procedurală

- Interesează pașii care sunt urmați de Prolog în evaluarea obiectelor și relațiilor

  Pentru a arăta că Pred este predecesorul lui Succ, arată întâi că Pred e părintele lui Child, apoi că Child e predecesorul lui Succ.
- Determină cum se obține rezultatul
- Contează ordinea clauzelor și a premiselor în reguli

Ordinea contează

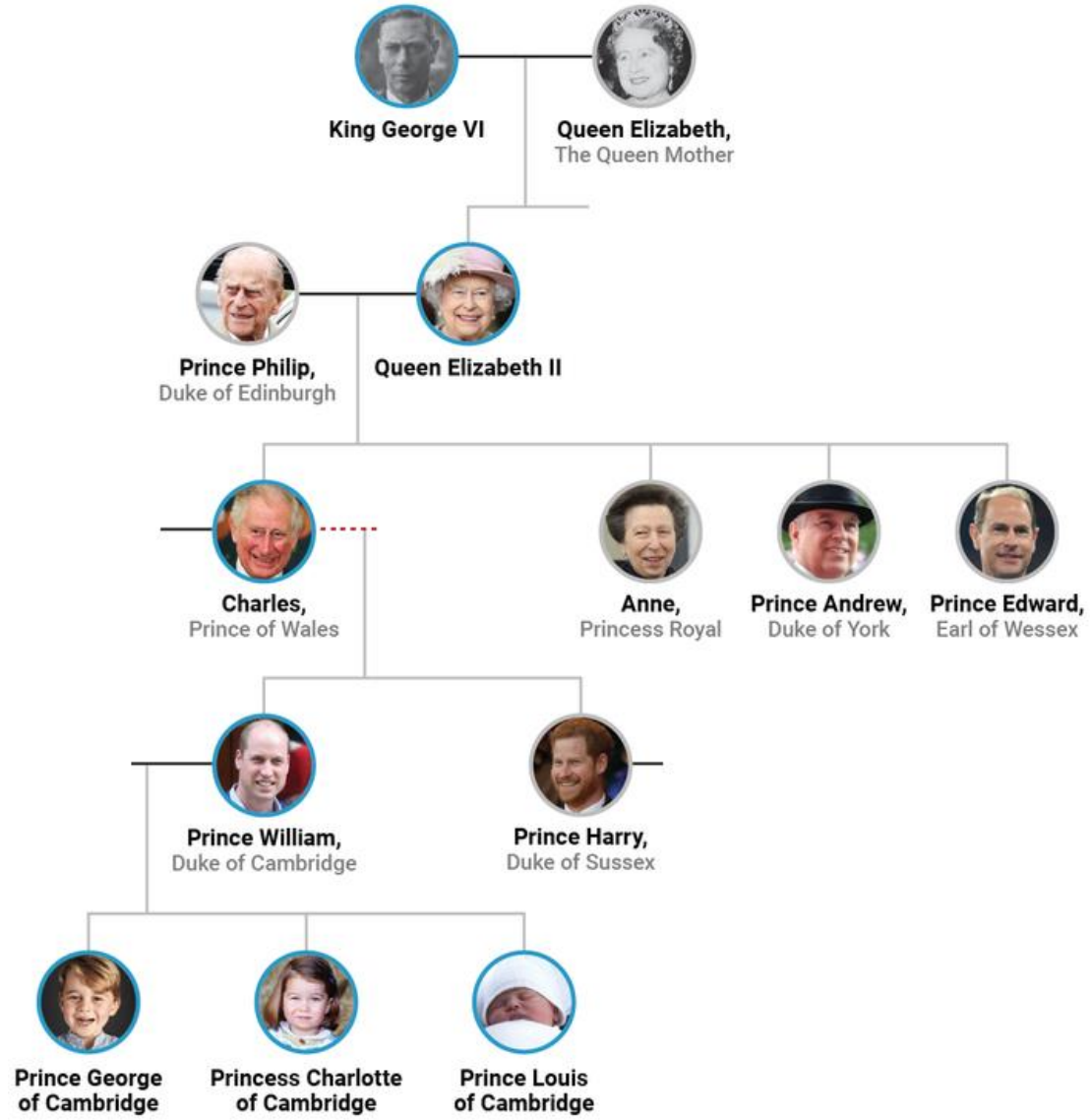
```
pred(P, C) :- parent(P, C).
pred(P, S) :- parent(P, C), pred(C, S).

%% clauze invers
pred1(P, S) :- parent(P, C), pred1(C, S).
pred1(P, C) :- parent(P, C).

%% premise invers
pred2(P, C) :- parent(P, C).
pred2(P, S) :- pred2(C, S), parent(P, C).

%% clauze și premise invers
pred3(P, S) :- pred3(C, S), parent(P, C).
pred3(P, C) :- parent(P, C).

?- pred(X, william).
```



Exemplu

```
pred2(P, C) :- parent(P, C).
pred2(P, S) :- pred2(C, S), parent(P, C).
?- pred2(X, william).
```

- parent(charles, william) -> X = charles
- pred2(C,william), parent(X,C)
  - C=charles -> X = philip, X = elizabeth2
  - pred2(C',william), parent(C,C')
    - C'=charles->C...-> X = george6, X = elizabeth
    - pred2(C'',william), parent(C',C'')
      - C''=charles->fail
      - pred2(C''',william), parent(C'',C''')
        - C'''=charles->fail
        - pred2(C'''',william), parent(C''',C'''')
          etc



Ordinea contează – Concluzii

- Contează ordinea clauzelor în program (se încearcă unificarea în ordine)
- Contează ordinea premiselor în corpul regulilor (se încearcă satisfacerea lor în ordine)
  - **Eficiență:** premisele a căror satisfacere **reduce mult spațiul de căutare** se pun primele
  - **Funcționalitate:** premisele care **instanțiază variabilele** se pun primele (v. parent și v. mai jos)

```
1.  factorial(0, 1).
2.  factorial(N, F) :-
3.          N > 0,
4.          N1 is N-1, factorial(N1, F1),
5.          F is N * F1.
6.
7.  factorial_1(0, 1).
8.  factorial_1(N, F) :-
9.          N > 0,
10.         factorial_1(N1, F1), N1 is N-1,
11.         F is N*F1.
```

Pentru interogarea factorial(5, X), programul știe să încerce apoi să satisfacă scopul factorial(4,X), etc.

Pentru interogarea factorial(5,X), programul încearcă apoi satisfacerea lui factorial(N1, X), și dă o eroare tip Arguments are not sufficiently instantiated

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

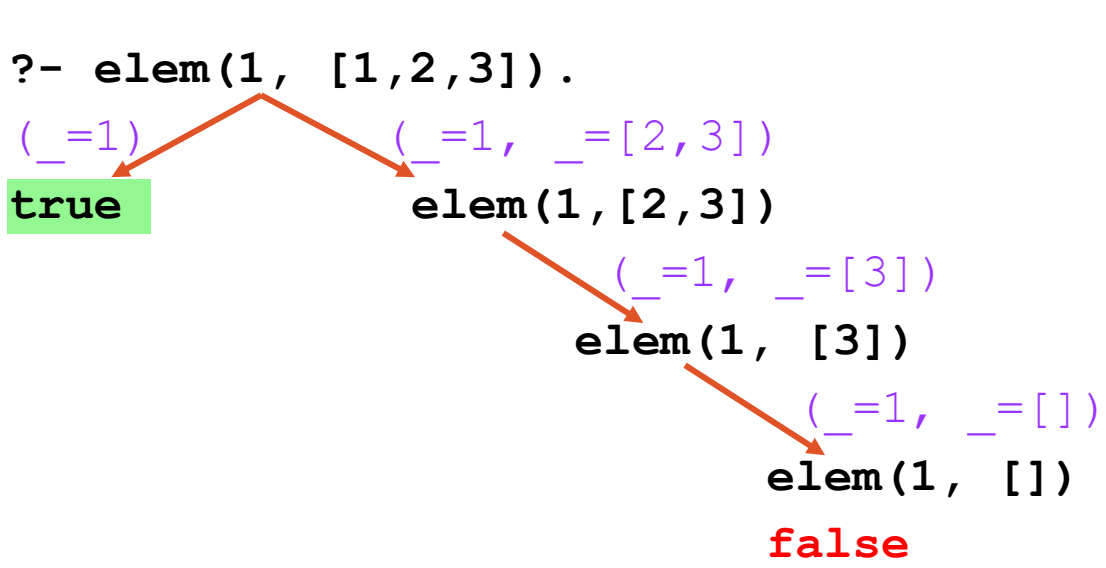
Exemplu – elem

```
1. elem(X, [X|_]).                          %% pt că member există
2. elem(X, [_|Rest]) :- elem(X, Rest).
```

```
?- elem(1, [1,2,3]).

?- elem(1, [1,2,1,3]).

?- elem(1, [1,2,X,Y]).

?- elem(X, [1,2,X,Y]).

?- elem(X, L).
```

Exemplu – elem

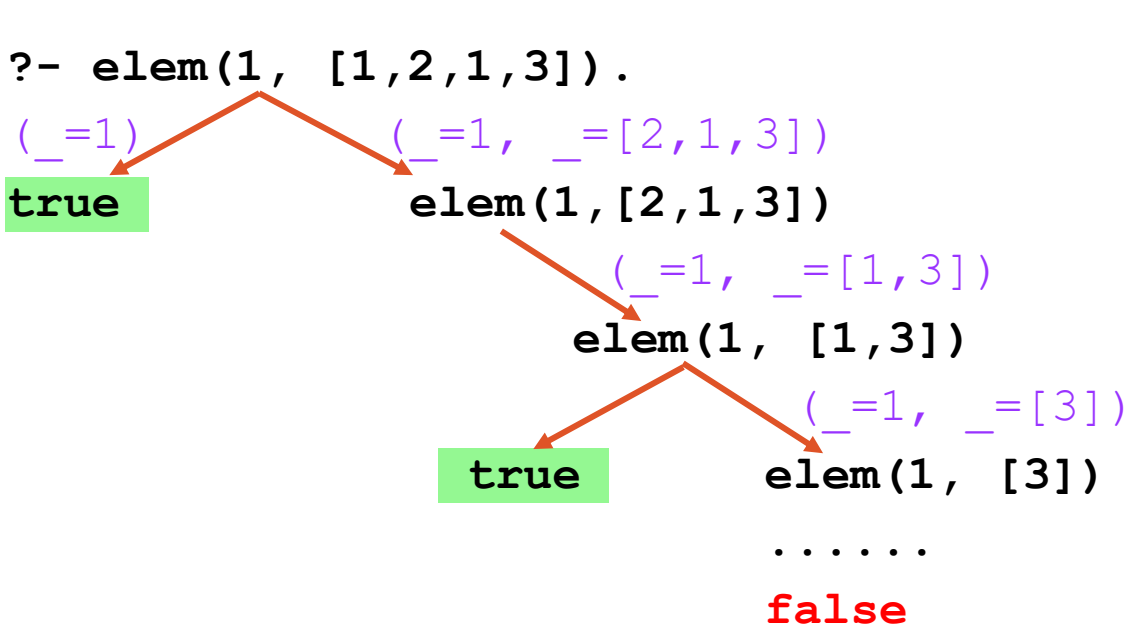
```
1. elem(X, [X|_]).                          %% pt că member există
2. elem(X, [_|Rest]) :- elem(X, Rest).
```




```
?- elem(1, [1,2,3]).
true ;
false.
```

Exemplu – elem

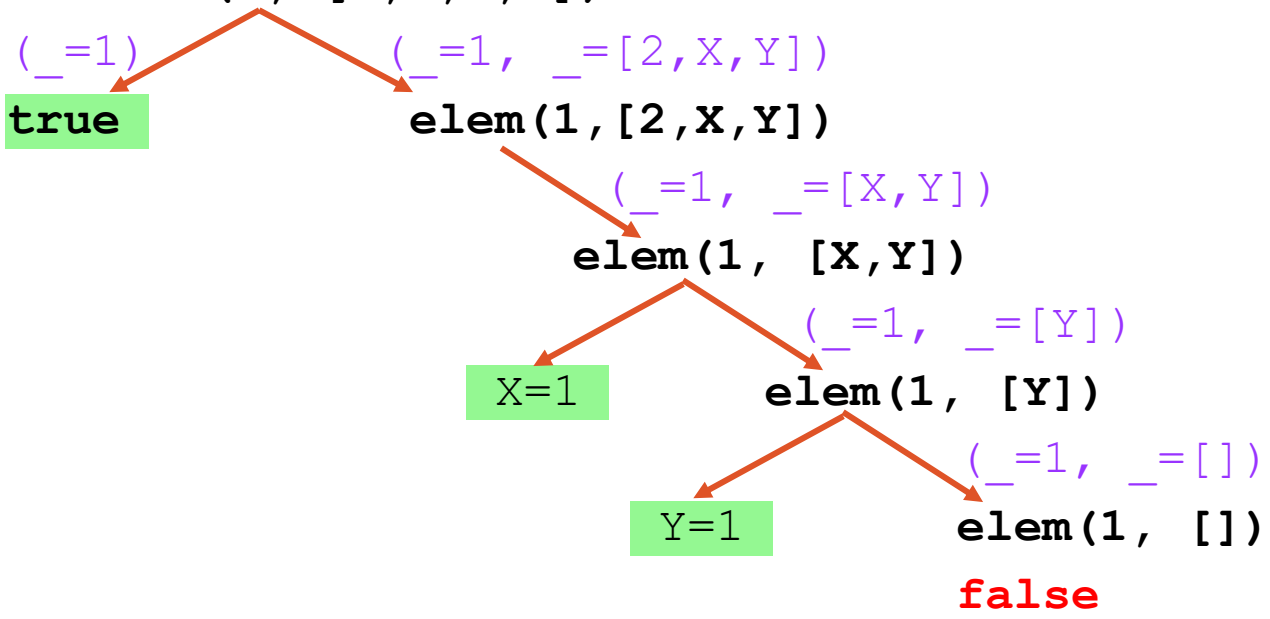
```
1. elem(X, [X|_]).                          %% pt că member există
2. elem(X, [_|Rest]) :- elem(X, Rest).
```




```
?- elem(1, [1,2,1,3]).
true ;
true ;
false.
```

Exemplu – elem

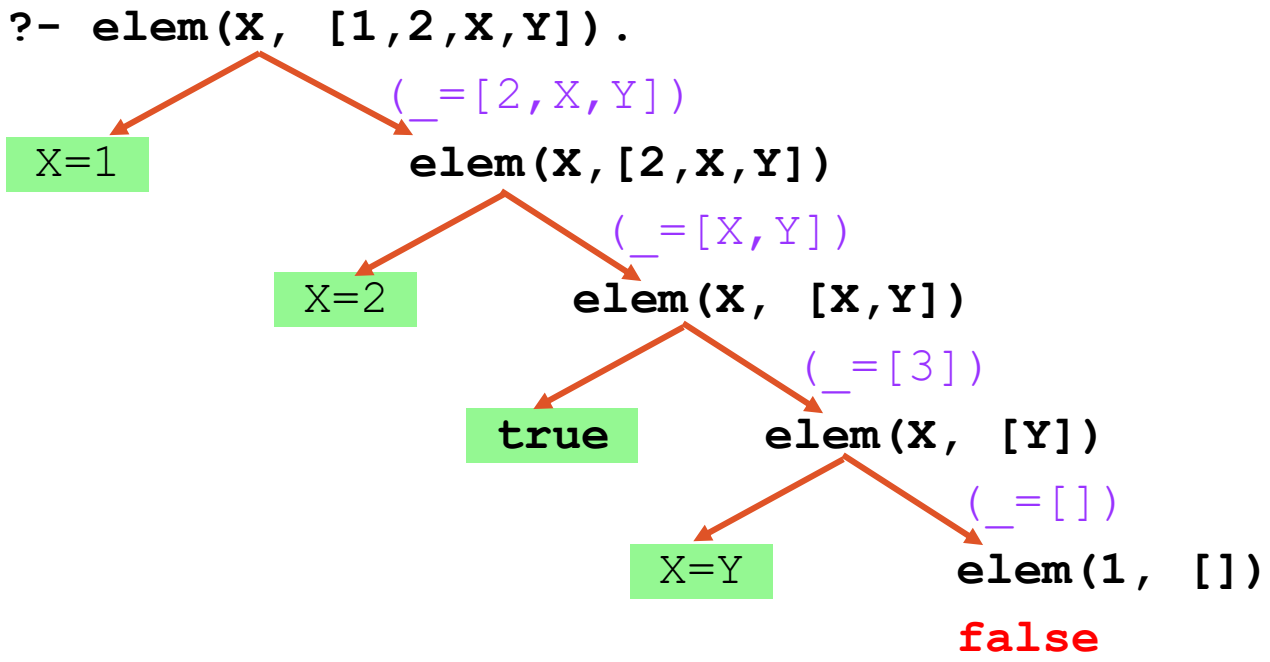
```
1. elem(X, [X|_]).                          %% pt că member există
2. elem(X, [_|Rest]) :- elem(X, Rest).
```




```
?- elem(1, [1,2,X,Y]).
true ;
X = 1 ;
Y = 1 ;
false.
```

Exemplu – elem

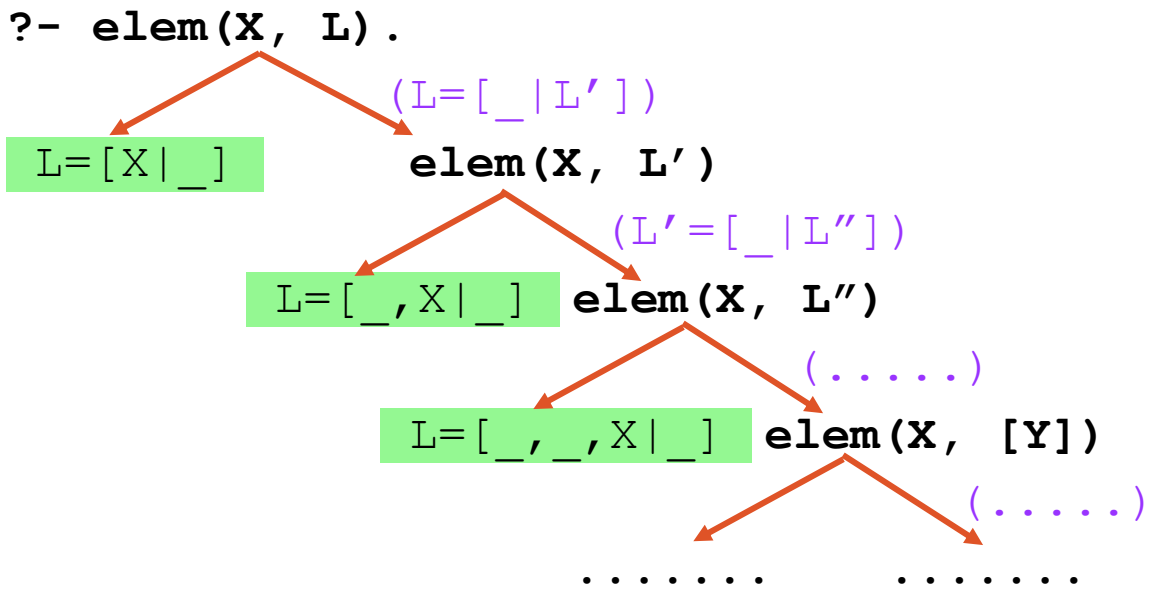
```
1. elem(X, [X|_]).                          %% pt că member există
2. elem(X, [_|Rest]) :- elem(X, Rest).
```




```
?- elem(X, [1,2,X,Y]).
X = 1 ;
X = 2 ;
true ;
X = Y ;
false.
```

Exemplu – elem

```
1. elem(X, [X|_]).                          %% pt că member există
2. elem(X, [_|Rest]) :- elem(X, Rest).
```




```
?- elem(X, L).
L = [X|_4056] ;
L = [_4054, X|_4062] ;
L = [_4054, _4060, X|_4068] .
```

Observații – elem

- Structurile Prolog **elimină separarea între datele de intrare și rezultate** (ieșire)
  - Atât intrările cât și ieșirile sunt termeni în structură (statutul de intrare/ieșire nu depinde nicicum de poziția în structură)
  - Spre deosebire de funcții, care separă clar intrarea (argumente) de ieșire (rezultat)
- Mulțumită mecanismului de unificare, elem poate fi folosit pentru
  - a **verifica** apartenența unui element dat la o listă dată: `elem(1, [1,2,3]).`
  - a **afla condițiile (instanțierile de variabile)** în care un element este membru într-o listă: `elem(1, [1,2,X,Y]).`
  - a **genera** elementele unei liste date: `elem(X, [1,2,3,4]).`
  - a genera liste care conțin un anumit element: `elem(1, L).`
- Nu contează doar satisfacerea scopului, ci satisfacerea scopului în toate modurile posibile: `elem(1, [1,2,1,3]).`

Exercițiu – concat

```
1. concat([], L, L).
2. concat([X|Rest], L, [X|Rez]) :- concat(Rest, L, Rez).
```


```
?- concat([1,2],[3,4],[1,2,3,4]).
?- concat([1,2],[3,4],[1,2,X,4]).
?- concat([1,2],[3,4],[1,2,3]).
?- concat(X,[3,4],[1,2,3,4]).
?- concat([1,2],Y,[1,2,3,4]).
?- concat([1,2],[3,4],Z).
?- concat(X,Y,[1,2,3,4]).
?- concat(X,[3,4],Z).
?- concat([1,2],Y,Z).
?- concat(X,Y,Z).
```

Regula se citește: dacă L1 este o listă cu head-ul X și tail-ul Rest, și dacă Rest concatenat cu L dă Rez, atunci L1 concatenat cu L dă o listă cu head-ul X și tail-ul Rez.

Atenție: În Prolog, spre deosebire de Haskell, pattern-urile se potrivesc și între ele. Prolog știe că în [X|Rest] și [X|Rez] folosesc **aceeași valoare X**.

Exercițiu – concat

```
1. concat([], L, L).
2. concat([X|Rest], L, [X|Rez]) :- concat(Rest, L, Rez).
```


```
?- concat([1,2],[3,4],[1,2,3,4]).        true.
?- concat([1,2],[3,4],[1,2,X,4]).        X = 3.
?- concat([1,2],[3,4],[1,2,3]).          false.
?- concat(X,[3,4],[1,2,3,4]).            X = [1,2]; false.
?- concat([1,2],Y,[1,2,3,4]).            Y = [3,4].
?- concat([1,2],[3,4],Z).                Z = [1,2,3,4].
?- concat(X,Y,[1,2,3,4]).  X = [], Y = [1,2,3,4]; X = [1], Y = [2,3,4]; ...
?- concat(X,[3,4],Z).      X = [], Z = [3,4]; X = [_582], Z = [_582,3,4]; ...
?- concat([1,2],Y,Z).                    Z = [1,2|Y].
?- concat(X,Y,Z).          X = [], Y = Z; X = [_588], Z = [_588|Y]; ...
```

Exerciții

Să se implementeze elem folosind doar concat.

Să se implementeze last folosind doar concat.

Să se șteargă primele și ultimele 2 elemente dintr-o listă folosind doar concat.

Exerciții

Să se implementeze elem folosind doar concat.

```
elem_(X, L) :- concat(_, [X|_], L).
```

Semnificație declarativă sporită
Semnificație procedurală diminuată

Să se implementeze last folosind doar concat.

```
last_(L, X) :- concat(_, [X], L).
```

Să se șteargă primele și ultimele 2 elemente dintr-o listă folosind doar concat.

```
del22(L, Del) :- concat([_,_|Del], [_,_], L).
```

Unificare, atribuire, evaluare

- **Unificarea** se poate realiza și explicit folosind operatorul **=**
  - Nu se produce niciun fel de evaluare, unificarea reușește doar dacă există o instanțiere a variabilelor în urma căreia cei 2 termeni devin identici

```
?- 1+2 = 1+2.
?- 1+2 = 1+X.
?- 1+2 = X+1.
?- X = 1+2.
```

  - Operatorul **\=** înseamnă **„nu unifică”**: `1+2 \= 3.`
  - Pentru **atribuire cu evaluarea** operațiilor aritmetice se folosește **is**: `X is 1+2.`
  - Pentru **verificarea egalității / inegalității valorilor** se folosesc **=:=** și **=\=**: `1+2 =\= 3.`
  - Pentru **verificarea egalității / inegalității structurale** se folosesc **==** și **\==**:

```
1+X == 1+2.
1+2 == 1+2.
```

Unificare, atribuire, evaluare

- **Unificarea** se poate realiza și explicit folosind operatorul **=**
  - Nu se produce niciun fel de evaluare, unificarea reușește doar dacă există o instanțiere a variabilelor în urma căreia cei 2 termeni devin identici

```
?- 1+2 = 1+2.        true.
?- 1+2 = 1+X.        X = 2.
?- 1+2 = X+1.        false.
?- X = 1+2.          X = 1+2.
```

Unificarea este în general însoțită de instanțieri

  - Operatorul **\=** înseamnă **„nu unifică”**: `1+2 \= 3. (true)`
  - Pentru **atribuire cu evaluarea** operațiilor aritmetice se folosește **is**: `X is 1+2. (X = 3)`
  - Pentru **verificarea egalității / inegalității valorilor** se folosesc **=:=** și **=\=**: `1+2 =\= 3. (false)`
  - Pentru **verificarea egalității / inegalității structurale** se folosesc **==** și **\==**:

```
1+X == 1+2. (false)
1+2 == 1+2. (true)
```

Limbajul Prolog – Cuprins

- Descriere generală
- Sintaxă
- Semantică
- Inferență
- Unificare și instanțiere
- Mecanisme de control

Oprirea backtracking-ului

Fie analogul lui compare din Haskell:

```
1. comp(X, Y, 'LT') :- X < Y.
2. comp(X, Y, 'EQ') :- X =:= Y.
3. comp(X, Y, 'GT') :- X > Y.
```

```
?- comp(2+3, 3+1, Ord).
```

```
?- comp(2+3, 3+1+1, Ord).
```

```
?- comp(2, 3+1, Ord).
```

Oprirea backtracking-ului

Fie analogul lui compare din Haskell:

```
1. comp(X, Y, 'LT') :- X < Y.
2. comp(X, Y, 'EQ') :- X =:= Y.
3. comp(X, Y, 'GT') :- X > Y.
```

Cele 3 reguli sunt **mutual exclusive**. Cum putem să îi spunem Prolog-ului ca, dacă s-a putut aplica o regulă, să nu le mai încerce pe celelalte?

```
?- comp(2+3, 3+1, Ord).          Ord = 'GT'.

?- comp(2+3, 3+1+1, Ord).        Ord = 'EQ' ; false.

?- comp(2, 3+1, Ord).            Ord = 'LT' ; false.
```

Predicatul ! (cut)

Rol cut: oprirea backtracking-ului la prima satisfacere a unui anumit scop

- în sensul că se vor încerca în continuare toate soluțiile care se pot obține din acest punct în dreapta, dar nu vom încerca să resatisfacem vreun scop din trecut

Funcționare cut

- Prima dată, cut reușește (este satisfiabil)
- Când se revine prin backtracking la cut, cut eșuează
- Toate regulile următoare cu același fel de antet (același predicat) sunt ignorate

```
1. comp(X, Y, 'LT') :- X < Y, !.
2. comp(X, Y, 'EQ') :- X =:= Y, !.
3. comp(_, _, 'GT').
```

Nu încercați să scrieți fără cut, ca în Haskell.
Se va face backtracking și veți obține că toate X și Y sunt în relația 'GT'!

Aplicabilitate cut

- **Eficientizarea programelor** (nu doar cazul regulilor mutual exclusive)

Exemplu

Să presupunem că vrem doar funcționalitatea de predicat a lui elem (testarea apartenenței)

```
1. elemP(X, [X|_]) :- !.
2. elemP(X, [_|Rest]) :- elemP(X, Rest).
```

```
?- elemP(1, [1,2,3]).
?- elemP(1, [1,2,1,3]).
?- elemP(1, [1,2,X,Y]).
?- elemP(X, [1,2,X,Y]).
?- elemP(X, L).
```

Aplicabilitate cut

- **Eficientizarea programelor** (nu doar cazul regulilor mutual exclusive)

Exemplu

Să presupunem că vrem doar funcționalitatea de predicat a lui elem (testarea apartenenței)

```
1. elemP(X, [X|_]) :- !.
2. elemP(X, [_|Rest]) :- elemP(X, Rest).
```

```
?- elemP(1, [1,2,3]).          true.
?- elemP(1, [1,2,1,3]).        true.
?- elemP(1, [1,2,X,Y]).        true.
?- elemP(X, [1,2,X,Y]).        X = 1.
?- elemP(X, L).                L = [X|_6714].
```

Se pierde abilitatea generativă a lui elem

Semnificația procedurală devine mai importantă decât cea declarativă (ea dictează acum care va fi rezultatul)

Backtracking doar la dreapta lui cut

```
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
sibling_(X, Y) :- parent(P, X), !, parent(P, Y), X \= Y.
```

```
?- sibling(X, anne).
```

```
?- sibling(anne, X).
```

```
?- sibling_(X, anne).
```

```
?- sibling_(anne, X).
```

Backtracking doar la dreapta lui cut

```
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
sibling_(X, Y) :- parent(P, X), !, parent(P, Y), X \= Y.
```

```
?- sibling(X, anne).
X = charles ;
X = andrew ;
X = edward ;
X = charles ;
X = andrew ;
X = edward ;
false.
```

```
?- sibling_(X, anne).
false.
```

Resatisfacerea lui parent(P, X) produce duplicate

parent(george6, elizabeth) leagă P la george6 fără posibilitate de revenire, după care parent(george6, anne) eșuează

```
?- sibling(anne, X).
X = charles ;
X = andrew ;
X = edward ;
X = charles ;
X = andrew ;
X = edward.
```

```
?- sibling_(anne, X).
X = charles ;
X = andrew ;
X = edward.
```

Mecanisme de control

- **true** – predicat care reușește întotdeauna
- **fail** – predicat care eșuează întotdeauna
- **cut** – predicat care oprește backtracking-ul pe structurile anterioare și regulile ulterioare
- **once(P)** – permite lui P să reușească o singură dată (once(P) :- P, !.)
- **not(P)** – reușește dacă P nu e satisfiabil (not(P) :- P, !, fail ; true.)

  (se preferă scrierea **\+** în loc de not, pentru a sugera că nu este vorba de negație logică)

true după sau (;) nu este atins, este ca și cum s-ar afla într-o regulă ulterioară

Negația ca eșec

Revenind pe șantier...

```
bun(X) :- \+sapa_groapa(X,_).

?- bun(X).
false.
```

- Interogarea **sapa_groapa(X,_)** se poate citi și „Există X care sapă groapa cuiva?”
- Interogarea \+**sapa_groapa(X,_)** **nu** se poate citi și „Există X care nu sapă groapa cuiva?”
  - Ea se citește ca un eșec al variantei afirmative:
    „not(Există X care sapă groapa cuiva)” ⇔ „Oricare X, X nu sapă groapa nimănui”
  - Funcționare: sapa_groapa(X,_) unifică cu sapa_groapa(ilie, gica)
    combinația !, fail condamnă satisfacerea scopului curent la eșec definitiv

Negația ca eșec – Exercițiu

```
1. om_bun(X) :- om(X), \+sapa_groapa(X,_).
2. om_bun_(X) :- \+sapa_groapa(X,_), om(X).
```

?- om_bun(X).

?- om_bun_(X).

Negația ca eșec – Exercițiu

```
1. om_bun(X) :- om(X), \+sapa_groapa(X,_).
2. om_bun_(X) :- \+sapa_groapa(X,_), om(X).
```

```
?- om_bun(X).
X = gica ;
false.
```

```
?- om_bun_(X).
false.
```

Observații:

- Se recomandă evitarea predicatului cut atunci când acesta distruge corespondența între semnificația declarativă și cea procedurală
- Semnificație declarativă = semnificație procedurală ⇔
  Programe ușor de înțeles, care fac ceea ce ne așteptăm să facă
- Prin urmare, atât cut cât și not trebuie folosite cu grijă și numai cu un motiv bun

Rezumat

Propoziții Prolog

Ipoteza lumii închise

Regula de inferență

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise

Regula de inferență

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile: forma relațională permite funcționarea în diverse sensuri (alternând ce este intrare și ce este rezultat)

Operatori de unificare/atribuire/verificare egalitate

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile: forma relațională permite funcționarea în diverse sensuri (alternând ce este intrare și ce este rezultat)

Operatori de unificare/atribuire/verificare egalitate: = / is / =:=, ==

Mecanisme de control

Rezumat

Propoziții Prolog: clauze Horn

Ipoteza lumii închise: există adevăruri doar în program, negația a ceva = eșec de a demonstra ceva

Regula de inferență: Rezoluția

Strategia de căutare: backward chaining

Algoritmi folosiți în procesul de inferență: DFS, backtracking, unificare

Semnificații ale programului: declarativă (ce este soluția), procedurală (cum se obține soluția)

Reguli reversibile: forma relațională permite funcționarea în diverse sensuri (alternând ce este intrare și ce este rezultat)

Operatori de unificare/atribuire/verificare egalitate: = / is / =:=, ==

Mecanisme de control: true, fail, cut, once, not