Curs 5

2018-2019

Programare Logică

Cuprins

Liste și recursie

Listă [t1,...,tn]

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

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

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

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

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

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

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

Liste

Exercițiu

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

Liste

Exercițiu

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

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

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

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


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


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

Liste

Exercițiu

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

Liste

Exercițiu

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

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

Liste

Exercițiu

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

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

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

Liste

Exercițiu

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

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

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

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

Liste

Exercițiu

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

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

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

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

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

Liste

Exercițiu

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

Liste

Exercițiu

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

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

Liste

Exercițiu

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

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

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

Liste

Exercițiu

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

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

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

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

Liste

Exercițiu

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

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

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

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

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

Generează și testează

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

Exercițiu

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

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

Predicat util:

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

Generează și testează

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

Exercițiu

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

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

Predicat util:

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

Două abordări posibile:

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

Generează și testează

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

Exercițiu

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

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

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.
```

Recursie

Exercițiu

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

Recursie

Exercițiu

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

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

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

Cum putem defini o variantă mai rapidă?

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

Recursie cu acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul iniţial nu am acumulat nimic.
```

Recursie cu acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul iniţial nu am acumulat nimic.

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

Recursie cu acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul iniţial nu am acumulat nimic.

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

revac([X|T], Acc, R) :- revac(T,[X|Acc],R).
% Acc conţine inversa listei care a fost deja parcursă.
```

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

Recursie

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

Recursie

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

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.

biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

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

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

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

```
?- time(biglist(50000,X)).
100,000 inferences, 0.016 CPU in 0.038 seconds
(41% CPU, 6400000 Lips)
X = [50000, 49999, 49998|...] .
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

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

```
?- time(biglist(50000,X)).
100,000 inferences, 0.016 CPU in 0.038 seconds
(41% CPU, 6400000 Lips)
X = [50000, 49999, 49998|...] .
```

- Predicatul cu recursie la coadă:

```
biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

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

```
?- time(biglist(50000,X)).
100,000 inferences, 0.016 CPU in 0.038 seconds
(41% CPU, 6400000 Lips)
X = [50000, 49999, 49998|...] .
```

- Predicatul cu recursie la coadă:

```
biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).

?- time(biglist_tr(50000,X)).
100,000 inferences, 0.000 CPU in 0.007 seconds
(0% CPU, Infinite Lips)
X = [50000, 49999, 49998|...]
```

Liste append/3

- Reamintim definiția funcției append/3:

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

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

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

Liste

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

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

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

  Această pereche poate fi notată `[t1,...,tn|T]- T`, dar notația nu este importantă.

Liste ca diferențe

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

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

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

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

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

Liste ca diferențe ([t1,...,tn|T], T)

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

Liste ca diferențe ([t1,...,tn|T], T)

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

- Dacă [t1,..., tn] este diferența (X1,T1), iar [q1,..., qk] este diferența (X2,T2) observăm că diferența (R,T) trebuie să fie [t1,...,tn,q1..., qk].

Liste ca diferențe ([t1,...,tn|T], T)

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

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

Liste ca diferențe ([t1,...,tn|T], T)

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

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

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

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

Liste ca diferențe ([t1,...,tn|T], T)

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

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

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

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

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

DCG (Definite Clause Grammars)

Structura frazelor

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

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

Structura frazelor

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

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

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

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

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

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

- Neterminalele definesc categorii gramaticale:

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

Gramatică independentă de context

GIC

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

Ce vrem să facem?

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

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

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

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

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

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

Definirea unei gramatici în Prolog

Gramatică independentă de context

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

Prolog

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

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

Definirea unei gramatici în Prolog

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

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

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

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

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

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

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

Definirea unei gramatici în Prolog

- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare.

Definirea unei gramatici în Prolog

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

  `append(X,Y,L)` este echivalent cu `X = L - Y`

Definirea unei gramatici în Prolog

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

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

  `s(L,Z) :- np(L,Y), vp(Y,Z)`

Definirea unei gramatici în Prolog

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

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

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

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

Definirea unei gramatici în Prolog

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

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

Definirea unei gramatici în Prolog

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

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

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

Definirea unei gramatici în Prolog

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

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

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

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

Definirea unei gramatici în Prolog

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

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

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

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

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

DCG în Prolog

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

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

iar codul scris anterior va fi generat automat.

Definite Clause Grammar

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

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

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

DCG în Prolog

Definite Clause Grammar

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

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

- Putem pune întrebările ca înainte:

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

- Putem folosi predicatul phrase/2:

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

DCG în Prolog

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

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


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


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

DCG în Prolog

Exemplu

Definiți numerele naturale folosind DCG.

DCG în Prolog

Exemplu

Definiți numerele naturale folosind DCG.

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

Definiția generată automat este:

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

Putem transforma listele în atomi:

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

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

Prolog impur

Lista tuturor soluțiilor

Cum găsim lista tuturor soluțiilor unui predicat?

- În Prolog există meta-predicatul `findall/3`, care acceptă ca argument un predicat arbitrar.

  KB: `p(a). p(b). p(c). p(d). p(a).`

  ```
  ?- findall(X, p(X),S).
  S = [a, b, c, d, a].
  ```

- Definiția lui `findall/3`

```
?- listing(findall/3).
:- meta_predicate'$bags':findall(?,0,-).
'$bags':findall(Templ, Goal, List) :- findall(Templ, Goal, List, []).
?- listing(findall/4).
:- meta_predicate'$bags':findall(?,0,-,?).
'$bags':findall(Templ, Goal, List, Tail) :-
    setup_call_cleanup('$new_findall_bag',
                       findall_loop(Templ, Goal, List,Tail),
                       '$destroy_findall_bag').
```

Liste

Exercițiu

Fie `p/1` un predicat. Scrieți un predicat `all_p/1` astfel încât întrebarea `?- all_p(S)` să instanțieze `S` cu lista tuturor atomilor pentru care `p` este adevărat.

```
p(a). p(b). p(c). p(d). p(a).

?- all_p(S).
S = [d,c,b,a].
```

Liste

Exercițiu

Fie `p/1` un predicat. Scrieți un predicat `all_p/1` astfel încât întrebarea `?- all_p(S)` să instanțieze `S` cu lista tuturor atomilor pentru care `p` este adevărat.

```
p(a). p(b). p(c). p(d). p(a).

?- all_p(S).
S = [d,c,b,a].


find_all(X,L,S):- p(X), \+ member(X,L),
                  find_all(_,[X|L],S).
find_all(_,L,L).


all_p(S) :- find_all(_, [], S).
```

Lista tuturor soluțiilor fără repetiții

```
find_all(X,L,S):- p(X), \+ member(X,L), find_all(_,[X|L],S).
find_all(_,L,L).
all_p(S) :- find_all(_, [], S).


?- all_p(S).
S = [d, c, b, a].

?- all_p([a,b,c,d]).
true.

?- all_p([a,b,c]).
true.

?- all_p([a,b,c,a]).
false. % pentru că a apare de două ori
```

Predicate ca argumente

Putem scrie predicatul `all_p` astfel încât să-i transmitem predicatul ca argument?

Ar trebui ca numele predicatului să fie o variabilă care să fie instanțiată în momentul apelului, dar acest lucru nu este permis de sintaxa Prolog (un functor trebuie să fie un atom).

Există o soluție folosind predicatul predefinit `=../2` care convertește un predicat p(t1,...,tn) în lista [p,t1,..., tn].

```
?- p(a) =.. L.
L = [p, a].

?- X =.. [foo,a,b,c]. X = foo(a, b, c).
```

Predicate ca argumente

Predicatul predefinit =../2 convertește un predicat p(t1,...,tn) în lista [p,t1,..., tn].

```
find_all(P,X,L,S):- Pr =..[P,X],Pr, \+ member(X,L),
                    find_all(P,_,[X|L],S).
find_all(_,_,L,L).
all(P,S) :- find_all(P,_, [], S).
```

Predicate ca argumente

Predicatul predefinit =../2 convertește un predicat p(t1,...,tn) în lista [p,t1,..., tn].

```
find_all(P,X,L,S):- Pr =..[P,X],Pr, \+ member(X,L),
                    find_all(P,_,[X|L],S).
find_all(_,_,L,L).
all(P,S) :- find_all(P,_, [], S).

p(a). p(b). p(c). p(d). p(a).
q(a). q(b). q(c).

?- all(q,S).
S = [c, b, a].

?- all(p,S).
S = [d, c, b, a].
```

Concluzii

- Programarea logică este bazată pe o teorie matematică clară: logica clauzelor Horn. Semantica denotațională a unui program poate fi defnită matematic, iar semantica operațională este bazată pe rezoluție.
- Limbajul Prolog este un limbaj complex, care îmbină cosntrucii teoretice (care au un corespondent în logică) cu trăsături nelogice, care cresc puterea de expresivitate.
- La acest curs ne interesează în special partea pură a limbajului Prolog, adică acele programe care se pot exprima în logica clauzelor Horn definite.

Pe săptămâna viitoare!