Curs 6

2024-2025

Programare Logică și Funcțională

Cuprins

Bibliografie:
P. Blackburn, J. Bos. K. Striegnitz, Learn Prolog Now! `http://cs.union.edu/~striegnk/learn-prolog-now/html/node88.html`

Clauza ! (cut)

Backtracking

Prolog folosește backtracking pentru a răspunde întrebărilor:

- În momentul în care Prolog încearcă să găsească un răspuns la o întrebare, ține minte toate punctele de decizie.
  - Puncte de decizie = situațiile în care găsește mai multe instanțieri.
- De fiecare dată când un drum eșuează sau se termină, Prolog sare la ultima alegere făcută și încearcă următoarea alternativă.

Backtracking

Exemplu

```
a(1).
b(1). b(2).
c(1). c(2).
d(2). e(2).
f(3).

p(X) :- a(X).
p(X) :- b(X),c(X),d(X),e(X).
p(X) :- f(X).
```

```
?- p(X).

X = 1 ;
X = 2 ;
X = 3.
```

Backtracking

Exemplu

```
a(1).
b(1). b(2).
c(1). c(2).
d(2). e(2).
f(3).

p(X) :- a(X).
p(X) :- b(X),c(X),
          d(X),e(X).
p(X) :- f(X).
```

```
[trace] ?- p(X).
Call: (8) p(_4430) ? creep
Call: (9) a(_4430) ? creep
Exit: (9) a(1) ? creep
Exit: (8) p(1) ? creep
X = 1 ;
Redo: (8) p(_4430) ? creep
Call: (9) b(_4430) ? creep
Exit: (9) b(1) ? creep
Call: (9) c(1) ? creep
Exit: (9) c(1) ? creep
Call: (9) d(1) ? creep
Fail: (9) d(1) ? creep
Redo: (9)b(_4430) ? creep
Exit: (9)b(2) ?
...
```

Cut

- În Prolog putem să "tăiem" punctele de decizie din backtracking, ghidând astfel căutarea soluțiilor și eliminând soluții alternative nedorite.
- O "tăietură" (cut) se introduce prin !.
- Este un predicat (de aritate 0) predefinit în Prolog care poate fi inserat oriunde în corpul unei reguli.
- Execuția subțintei ! este mereu cu succes.
- De fiecare dată când ! este întâlnit în corpul unei reguli, sunt **finale** toate alegerile făcute începând cu momentul în care capul acelei reguli a fost unificat cu scopul părinte.

Backtracking

Exemplu

```
a(1).
b(1). b(2).
c(1). c(2).
d(2). e(2).
f(3).

p(X) :- a(X).
p(X) :- b(X),c(X),!,d(X),e(X).
p(X) :- f(X).
```

```
?- p(X).

X = 1.
```

Backtracking

Exemplu

```
a(1).
b(1). b(2).
c(1). c(2).
d(2). e(2).
f(3).

p(X) :- a(X).
p(X) :- b(X),c(X),!,
          d(X),e(X).
p(X) :- f(X).
```

```
[trace] ?- p(X).
Call: (8) p(_4430) ? creep
Call: (9) a(_4430) ? creep
Exit: (9) a(1) ? creep
Exit: (8) p(1) ? creep
X = 1 ;
Redo: (8) p(_4430) ? creep
Call: (9) b(_4430) ? creep
Exit: (9) b(1) ? creep
Call: (9) c(1) ? creep
Exit: (9) c(1) ? creep
Call: (9) d(1) ? creep
Fail: (9) d(1) ? creep
Fail: (8) p(_4430) ? creep
false.
```

!

```
p(X):- q1(X),...,qn(X),!,r1(X),...,rk(X).
```

- Mecanismul de backtracking poate fi restricționat folosind !.
- Predicatul ! reușește întotdeauna, dar predicatul părinte eșuează atunci când se încearcă backtracking "peste" !.
- Mecanismul de backtrackig funcționează pentru clauzele care se găsesc înainte de ! sau după !
- Alegerile (instanțierile) făcute în execuția unui predicat înaite de a se ajunge la ! nu mai pot fi schimbate.

Backtracking și !

Exemplu

În exemplul nostru, cum putem modifica baza de cunoștințe astfel încât mecanismul de backtracking să ajungă și la ultima alternativă?

```
p(X) :- a(X).
p(X) :- b(X),c(X),!, d(X),e(X).
p(X) :- f(X).
```

Backtracking și !

Exemplu

În exemplul nostru, cum putem modifica baza de cunoștințe astfel încât mecanismul de backtracking să ajungă și la ultima alternativă?

```
p(X) :- a(X).
p(X) :- b(X),c(X),!, d(X),e(X).
p(X) :- f(X).
```

Pentru a ajunge la ultima alternativă, mecanismul de backtracking trebuie să funționeze înainte de !:

```
a(1). b(2). c(1). d(2). e(2). f(3).
```

```
?- p(X).
X = 1 ;
X = 3
```

Backtracking și !

- Următorul predicat clasifică un număr folosind clauze care se exclud reciproc:

```
range(X,'A'):- X < 3.
range(X,'B'):- 3 =< X , X < 6.
range(X,'C'):-  6 =< X.
```

- Pentru a evita backtracking-ul după ce o clasificare este obținută se poate folosi !

```
range(X,'A'):- X < 3,!.
range(X,'B'):- 3 =< X , X < 6, !.
range(X,'C'):-  6 =< X.
```

Acest tip de utilizare se numește cut verde și este recomandat pentru a îmbunătăți performanța unui program.

Backtracking și !

- Putem scrie programul anterior astfel:

```
range(X,'A'):- X < 3,!.
range(X,'B'):- X < 6,!.
range(X,'C').
```

- Ce se întâmplă dacă eliminăm ! în programul de mai sus?

```
range(X,'A'):- X < 3,!.
range(X,'B'):- X < 6.
range(X,'C').

?- range(4,Cat).
Cat = 'B' ;
Cat = 'C'
```

Acest tip de utilizare se numește cut roșu, deoarece prezența predicatului ! afectează logica programului. Acest mod de utilizare trebuie evitat.

Probleme cu cut

- Cuts sunt foarte utile pentru a ghida Prolog spre o soluție.
- Totuși, introducând cuts, renunțăm la anumite caracteristici declarative ale Prolog-ului și ne îndreptăm spre un sistem procedural.

Probleme cu cut

- Cuts sunt foarte utile pentru a ghida Prolog spre o soluție.
- Totuși, introducând cuts, renunțăm la anumite caracteristici declarative ale Prolog-ului și ne îndreptăm spre un sistem procedural.

Exemplu

- Predicatul `add/3` inserează un element într-o listă doar dacă acel element nu este deja un membru al listei.
- Elementul pe care dorim să îl inserăm este dat ca prim argument, iar lista ca al doilea argument. Variabila dată ca al treilea argument este rezultatul.

```
?- add(elephant, [dog, donkey, rabbit], List).
List = [elephant, dog, donkey, rabbit]

?- add(donkey, [dog, donkey, rabbit], List).
List = [dog, donkey, rabbit]
```

Probleme cu cut

Exemplu

O soluție posibilă:

```
add(Element, List, List) :-
        member(Element, List), !.

add(Element, List, [Element | List]).
```

- Dacă elementul se află deja în listă, lista soluție este chiar cea inițială. Cum aceasta este unica soluție posibilă, împiedicăm Prolog-ul să caute o altă soluție introducând `!`.
- Altfel, în rezultat adăugam elementul la începutul listei de intrare.

Probleme cu cut

- Acesta este un exemplu în care cut creează probleme.
- Când predicatul add/3 este folosit cu o variabilă în al treilea argument funcționează corect.
- Totuși, dacă folosim acest predicat cu o listă instanțiată în al treilea argument, răspunsul Prolog-ului nu este neapărat cel așteptat.

Exemplu

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

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

?- add(a, [a, b, c, d], [a, b, a, c, d]).
false
```

Probleme cu cut

Exemplu

O soluție alternativă:

```
add(Element, List, Result) :-
        member(Element, List), !,
        Result = List.

add(Element, List, [Element | List]).
```

- Din punct de vedere declarativ, cele două soluții sunt echivalente, dar procedural se comportă diferit.

Atenție cum folosiți cut!

Negarea unui predicat: `\+ pred(X)`

Răspunsurile din Prolog

- 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, răspunsul `true` la o țintă nu înseamnă doar că ținta este adevarată, ci și că este demonstrabilă.
- Astfel, un răspuns `false` nu înseamnă neapărat că ținta nu este adevărată, ci doar că Prolog nu a reușit să găsească o demonstrație.

Răspunsurile din Prolog

Exemplu

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

?- animal(cat).
false
```

Răspunsurile din Prolog

Exemplu

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

?- animal(cat).
false
```

- Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- Totuși, dacă specificăm complet o problemă (adică specificăm toate cazurile posibile), atunci noțiunile de nedemonstrabil și fals coincid. Atunci un `false` e chiar un fals.

Operatorul \+

- Câteodată poate dorim să negăm o țintă.
- Negarea unei ținte se poate defini astfel:

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

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

Operatorul \+

- Câteodată poate dorim să negăm o țintă.
- 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 foloseste pentru a nega un predicat.
- O țintă `\+ Goal` reușește dacă Prolog nu găsește o demonstrație pentru `Goal`.
- Semantica operatorului \+ se numește negation as failure.
- Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.

Operatorul \+

- Câteodată poate dorim să negăm o țintă.
- 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 foloseste pentru a nega un predicat.
- O țintă `\+ Goal` reușește dacă Prolog nu găsește o demonstrație pentru `Goal`.
- Semantica operatorului \+ se numește negation as failure.
- Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.

"nevinovat până la proba contrarie"

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.

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește nemonoton.

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

  Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește nemonoton.

- Sistemele logice pe care le-am sudiat până acum (calculul cu propoziții clasic, logica de ordinul I, logica clauzelor Horn) sunt monotone:

  dacă din $\Gamma \vdash \varphi$ și $\Gamma \subseteq \Sigma$ atunci $\Sigma \vdash \varphi$.

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește nemonoton.

Raționamente nemonotone

- În exemplul anterior extindem baza de cunoștințe astfel:

```
married(peter, lucy). married(paul, mary).
married(bob, juliet). married(harry, geraldine).
married(john, anne).

single(P) :- \+ married(P, _), \+ married(_, P).

?- single(anne).
false
```

  Observăm că largind mulțimea de ipoteze (baza de cunoștințe) putem demonstra mai puțin! Acest tip de raționament se numește nemonoton.

- Sistemele logice pe care le-am sudiat până acum (calculul cu propoziții clasic, logica de ordinul I, logica clauzelor Horn) sunt monotone:

  dacă din $\Gamma \vdash \varphi$ și $\Gamma \subseteq \Sigma$ atunci $\Sigma \vdash \varphi$.

Lista tuturor soluțiilor

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].
```

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`

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

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

  - definiția concatenării este:

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

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

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.

Definirea unei gramatici în Prolog

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

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

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

Acestă 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;
...
```