

Curs 7 (Prolog)

Cuprins

- 1 Recursie
- 2 Generează și testează
- 3 Structura generală a unui joc / Jocul Nim
- 4 SAT solver

Rekursie

Recursie cu acumulatori

Exercițiu

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

Recursie cu acumulatori

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).
```

Cum putem defini o variantă mai rapidă (păstrând și proprietățile generative)?

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.

Recurсие cu acumulatori

- Varianta inițială:

```
rev([], []).
```

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

- Varianta cu acumulator

```
rev(L, R) :- revac(L, [], R).
```

% la momentul inițial nu am acumulat nimic.

```
revac([], R, R).
```

% cand lista inițială a fost consumată,

% am acumulat rezultatul final.

```
revac([X|T], Acc, R) :- revac(T, [X|Acc], R).
```

% Acc conține inversa listei care a fost deja parcursă.

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

Recursie

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

Recursie

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

```
biglist(0, []).
```

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

```
biglist_tr(0, []).
```

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

Recursie la coadă

- Predicat **fără** recursie la coadă:
`biglist(0,[]).`

`biglist(N,[N|T]) :- N >= 1, M is N-1, biglist(M,T), M=M.`
Apelul recursiv întoarce valoarea găsită în predicatul apelant, această valoare urmând a fi prelucrată.

```
?- use_module(library(statistics)).  
?- 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.

`?- use_module(library(statistics)).`
`?- 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).\`

Exercițiu

- Definiți un predicat `sum/2` care calculează suma elementelor unei liste.

```
sum([],0).
```

```
sum([X|T],R) :- sum(T,S), R is S+X.
```

Soluția de mai sus este corectă, dar va performa mai bine dacă o scriem cu recursie la coadă.

Pentru aceasta folosim acumulatori.

Recursie cu acumulatori

- Varianta inițială:

`sum([],0).`

`sum([X|T],R) :- sum(T,S), R is S+X.`

- Varianta cu acumulator:

Recursie cu acumulatori

- Varianta inițială:

`sum([],0).`

`sum([X|T],R) :- sum(T,S), R is S+X.`

- Varianta cu acumulator:

`sumac(L,R) :- sumaux(L,0,R).`

`sumaux([],S,S).`

`sumaux([X|T],S,R) :- S1 is S+X, sumaux(T,S1,R).`

Recursie cu acumulatori

- Varianta inițială:

```
sum([],0).  
sum([X|T],R) :- sum(T,S), R is S+X.
```

- Varianta cu acumulator:

```
sumac(L,R) :- sumaux(L,0,R).
```

```
sumaux([],S,S).  
sumaux([X|T],S,R) :- S1 is S+X, sumaux(T,S1,R).
```

- Varianta cu acumulator este mai rapidă:

```
?- time((biglist_tr(50000,T), sum(T,R))).  
% 200,002 inferences, 0.016 CPU in 0.021 seconds  
R = 1250025000 .
```

```
?- time((biglist_tr(50000,T), sumac(T,R))).  
% 200,002 inferences, 0.016 CPU in 0.009 seconds  
R = 1250025000 .
```


Generează și testează

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.
```

Generează și testează

Exercițiu: Cel mai lung cuvânt

Acest exemplu este din

Ulle Endriss, *Lecture Notes – An Introduction to Prolog Programming*.

Countdown este un joc de televiziune popular în Marea Britanie în care jucătorii trebuie să găsească un cuvânt cât mai lung cu literele dintr-o mulțime dată de nouă litere.

Concret, vom încerca să găsim o soluție optimă pentru următorul joc:

*Primind nouă litere din alfabet (nu neapărat unice),
trebuie să construim cel mai lung cuvânt format din literele date
(pot rămâne litere nefolosite).*

Vom rezolva jocul pentru cuvinte din limba engleză.

Scorul obținut este lungimea cuvântului găsit.

Countdown

Exercițiu:

Scopul final este de a construi un predicat în Prolog `topsolution/3`: dându-se o listă de litere în primul argument, trebuie să returneze în al doilea argument cuvântul din baza de cunoștințe de lungime maximă care poate fi format cu literele date; lungimea cuvântului este întoarsă în ultimul argument.

```
?- topsolution([r,d,i,o,m,t,a,p,v],Word, Score).  
Word = dioptra,  
Score =7
```

Countdown

Fișierul `words.pl` conține o listă cu peste 350.000 de cuvinte din limba engleză, de la a la zyzzyva, sub formă de fapte. Acest fișier poate fi a descărcat de la adresa <http://tinyurl.com/prolog-words>.
Salvați acest fișier în același director cu fișierul programului vostru și includeți linia `:- include('words.pl').` în programul vostru pentru a putea folosi aceste fapte.

word.pl

```
...  
word(abbotric).      word(abbots).  
word(abbotship).    word(abbotships).  
word(abbott).       word(abbozzo).  
word(abbr).         word(abbrev).  
word(abbreviatable). word(abbreviating).  
...
```

Countdown

Exercițiu:

Predicatul predefinit în Prolog `atom_chars(Atom,CharList)` descompune un atom într-o listă de caractere.

```
word_letters(Word,Letters) :- atom_chars(Word,Letters).
```

```
?- word_letters(hello,X).  
X = [h,e,l,l,o]
```

Observați că puteți folosi acest predicat pentru a găsi cuvinte în engleză de 45 de litere:

```
?- word(Word), word_letters(Word,Letters),  
   length(Letters,45).
```

Countdown

Exercițiu:

Predicatul predefinit în Prolog `atom_chars(Atom,CharList)` descompune un atom într-o listă de caractere.

```
word_letters(Word,Letters) :- atom_chars(Word,Letters).
```

```
?- word_letters(hello,X).  
X = [h,e,l,l,o]
```

Observați că puteți folosi acest predicat pentru a găsi cuvinte în engleză de 45 de litere:

```
?- word(Word), word_letters(Word,Letters),  
   length(Letters,45).
```

```
W = pneumonoultramicroscopicsilicovolcanoconiosis
```

Countdown

Exercițiu:

Scrieți un predicat `cover/2` care, primind două liste, verifică dacă a doua listă "acoperă" prima listă (i.e., verifică dacă fiecare element care apare de k ori în prima listă apare de cel puțin k ori în a doua listă).

De exemplu

```
?- cover([a,e,i,o], [m,o,n,k,e,y,b,r,a,i,n]).  
true
```

```
?- cover([e,e,l], [h,e,l,l,o]).  
false
```

Countdown

Exercițiu:

Scrieți un predicat `cover/2` care, primind două liste, verifică dacă a doua listă "acoperă" prima listă (i.e., verifică dacă fiecare element care apare de k ori în prima listă apare de cel puțin k ori în a doua listă).

```
cover([],_).
cover([Head|Tail], List) :-
    select(Head,List,Result),
    cover(Tail,Result).

/*
select(X,L,R) este true
           daca R se obtine din L eliminand X
*/
```

Countdown

Exercițiu:

Scrieți un predicat `solution/3` care primind o listă de litere ca prim argument și un scor dorit ca al treilea argument, returnează prin al doilea argument un cuvânt cu lungimea egală cu scorul dorit, "acoperit" de lista respectivă de litere.

De exemplu

```
?- solution([g,i,g,c,n,o,a,s,t], Word, 3).  
Word = act
```

Generează și testează

Exercițiu:

Scrieți un predicat `solution/3` care primind o listă de litere ca prim argument și un scor dorit ca al treilea argument, returnează prin al doilea argument un cuvânt cu lungimea egală cu scorul dorit, "acoperit" de lista respectivă de litere.

```
solution(ListLetters, Word, Score) :-  
    word(Word),  
    word_letters(Word, Letters),  
    length(Letters, Score),  
    cover(Letters, ListLetters).
```


Countdown

Exercițiu:

Implementați predicatul `topsolution/3`.

Testați, de exemplu, predicatul definit pe mulțimea de litere:

`[y,c,a,l,b,e,o,s,x]`

Aceasta este una listele de litere folosite în ediția de *Countdown* din 18 Decembrie 2002 din Marea Britanie în care Julian Fell a obținut cel mai mare scor din istoria concursului. Pentru lista de mai sus, el a găsit cuvântul *cables*, câștigând astfel 6 puncte.

Poate programul vostru să bată acest scor?

Countdown

Exercițiu:

Implementați predicatul `topsolution/3`: dându-se o listă de litere în primul argument, trebuie să returneze în al doilea argument cuvântul din baza de cunoștințe de lungime maximă care poate fi format cu literele date; lungimea cuvântului este întoarsă în ultimul argument.

```
topsolution(ListLetters, Word, Score) :-  
    length(ListLetters, X),  
    search_solution(ListLetters, Word, X),  
    atom_length(Word, Score).
```

Countdown

Exercițiu:

Implementați predicatul `topsolution/3`: dându-se o listă de litere în primul argument, trebuie să returneze în al doilea argument cuvântul din baza de cunoștințe de lungime maximă care poate fi format cu literele date; lungimea cuvântului este întoarsă în ultimul argument.

```
topsolution(ListLetters,Word, Score) :-  
    length(ListLetters,X),  
    search_solution(ListLetters,Word,X),  
    atom_length(Word,Score).  
  
search_solution(_,'',0).  
search_solution(ListLetters,Word,X) :-  
    solution(ListLetters,Word,X).  
search_solution(ListLetters,Word,X) :- Y is X-1,  
    search_solution(ListLetters,Word,Y).
```

Countdown

Exercițiu:

Implementați predicatul `topsolution/3`.

Testați, de exemplu, predicatul definit pe mulțimea de litere:

`[y,c,a,l,b,e,o,s,x]`

Aceasta este una listele de litere folosite într-o ediție de *Countdown* din Marea Britanie. Pentru lista de mai sus, concurenții au găsit cuvântul *cables*, care are scorul 6.

Poate programul vostru să bată acest scor?

```
?- topsolution([y,c,a,l,b,e,o,s,x],Word,Score).
```

```
Word = calyxes,
```

```
Score =7.
```

Structura generală a unui joc / Jocul Nim

Joc: structura generală

În cartea *The Art of Prolog* (L.S. Sterling, E.Y. Shapiro, The Art of Prolog, The MIT Press, 1994) este propus un framework general pentru descrierea jocurilor:

```
play(Game) :- initialize(Game,Position,Player),  
               display_game(Position),  
               play(Position,Player).
```

```
play(Position,Player) :- game_over(Position,Player,Result),  
                           !, announce(Result).
```

```
play(Position,Player) :- choose_move(Position,Player,Move),  
                           move(Move,Position,Position1),  
                           display_game(Position1),  
                           next_player(Player,Player1),  
                           !, play(Position1,Player1).
```

Jocul Nim

În continuare vom implementa jocul Nim:

- **Regula:** joacă mai mulți jucători care mută pe rând,
- **Tabla:** mai multe gramezi cu piese,
- **Mutare:** jucatorul ia una sau mai multe piese dintr-o gramadă; poate lua toată grămada, dar piesele trebuie luate din aceeași grămadă),
- **Rezultat:** câștigă jucătorul care a făcut ultima mutare.

Jocul Nim - Exemplu

Tabla: $[2,5,4]$

Un șir de mutări posibile:

Alice ia 4 obiecte din gramada 2: $[2,1,4]$

Bob ia 1 obiect din gramada 3: $[2,1,3]$

Alice ia 2 obiecte din gramada 1: $[1,3]$

Bob ia 1 obiect din gramada 1: $[3]$

Alice ia 3 obiecte din gramada 1: Alice a castigat!

Joc Nim

```
play(Game) :- initialize(Game,Position,Player),  
              display_game(Position),  
              play(Position,Player).  
  
% jucatorii sunt computer si opponent  
% pozitia initiala este [1,3,5,7]  
  
initialize(nim,[1,3,5,7],opponent).  
  
display_game(Position) :- write(Position), nl.
```

Joc Nim: initializarea

```
play(Game) :- initialize(Game,Position,Player),  
              display_game(Position),  
              play(Position,Player).
```

```
initialize(nim,[1,3,5,7],opponent).  
display_game(Position) :- write(Position), nl.
```

```
game_over([],Player,Player).
```

```
announce(computer) :- write('You won! Congratulations. '), nl.  
announce(opponent) :- write('I won. '), nl.
```

```
play(Position,Player) :- game_over(Position,Player,Result),  
                          !,announce(Result).
```

Joc Nim: jucatorii

```
play(Position,Player) :- choose_move(Position,Player,Move),  
                           move(Move,Position,Position1),  
                           display_game(Position1),  
                           next_player(Player,Player1),  
                           !, play(Position1,Player1).
```

```
choose_move(Position,Player,Move):- ...  
move(Move,Position,Position1):- ...
```

```
next_player(computer,opponent).  
next_player(opponent,computer).
```

Joc Nim: efectuarea mutarii

Exercițiu: scrieți un predicat `move(Move, Position, NewPosition)` care să efectueze o mutare; vom reprezenta mutarea a M elemente din grămada K prin perechea (K,M) .

Joc Nim: efectuarea mutarii

Exercițiu: scrieți un predicat `move(Move, Position, NewPosition)` care să efectueze o mutare; vom reprezenta mutarea a M elemente din grămada K prin perechea (K,M) .

`% o mutare este o pereche (K,M): din gramada K iau M`

```
move((1,N), [N|Ns], Ns).
```

```
move((1,M), [N|Ns], [N1|Ns]) :- N > M, N1 is N - M.
```

```
move((K,M), [N|Ns], [N|Ns1]) :- K > 1, K1 is K - 1,  
                                move((K1,M), Ns, Ns1).
```

Joc Nim: mutarea oponentului

Exercițiu: scrieți predicatul `choose_move(Position, opponent, Move)` care citește mutarea oponentului și verifică dacă aceasta este corectă; în caz contrar, îi cere o altă mutare.

Joc Nim: mutarea oponentului

Exercițiu: scrieți predicatul `choose_move(Position, opponent, Move)` care citește mutarea oponentului și verifică dacă aceasta este corectă; în caz contrar, îi cere o altă mutare.

```
% nth1(K,L,X) este true daca X este elementul din pozitia K  
% din lista L, unde prima pozitie este 1
```

```
legal((K,N),Position) :- 0 < K, 0 < N,  
                           nth1(K,Position,M),  
                           N =< M.
```

```
choose_move(Position,opponent,Move) :-  
    writeln(['Please make move']),  
    read(Move),legal(Move,Position).  
choose_move(Position,opponent,Move) :-  
    writeln(['Illegal move!']),  
    choose_move(Position,opponent,Move).
```

Joc Nim: mutarea calculatorului

Exercițiu: scrieți predicatul `choose_move(Position, computer, Move)` care întoarce în `Move` o mutare legală arbitrară.

```
% o mutare este o pereche (K,M): din gramada K iau M
```

```
choose_move(Position,computer,(K,M)) :-  
    length(Position,L),  
    random_between(1,L, K),  
    nth1(K,Position, N),  
    random_between(1,N,M).
```


Joc Nim: alegerea mutărilor

```
% o mutare este o pereche (K,M): din gramada K iau M

choose_move(Position,opponent,Move) :-
    writeln(['Please make move']),
    read(Move),legal(Move,Position).

choose_move(Position,opponent,Move) :-
    writeln(['Illegal move!']),
    choose_move(Position,opponent,Move).

choose_move(Position,computer,(K,M)) :-
    length(Position,L),
    random_between(1,L, K),
    nth1(K,Position, N),
    random_between(1,N,M).

% random_between este predefinit
```

Joc Nim: mutarea calculatorului

```
?- play(nim).
```

```
[1,3,5,7]
```

```
[Please make move]
```

```
|: (3,2).
```

```
[1,3,3,7]
```

```
[1,2,3,7]
```

```
[Please make move]
```

```
|: (4,7).
```

```
[1,2,3]
```

```
[1,2]
```

```
[Please make move]
```

```
|: (2,1).
```

```
[1,1]
```

```
[1]
```

```
[Please make move]
```

```
|: (1,1).
```

```
[]
```

```
You won! Congratulations.
```

```
true .
```

Jocul Nim: strategie

Puteți citi o analiză matematică completă a jocului Nim în articolul:

C.L. Bouton, *Nim, a game with a complete mathematical theory*, Annals of Mathematics, 3 (14): 35–39,

<https://www.jstor.org/stable/1967631>

Pentru acest joc există o strategie câștigătoare: pozițiile sunt clasificate în sigure și nesigure; dacă poziția inițială este sigură, al 2-lea jucător are o strategie care îi asigură victoria, iar dacă poziția inițială este nesigură, primul jucător are o strategie care îi asigură victoria.

În *The Art of Prolog* este prezentată o implementare a acestei strategii.

SAT solver

SAT solver în Prolog

- Definirea formulelor.
- Determinarea formelor normale (nnf și cnf).
- Determinarea formei clauzale.
- Implementarea algoritmului Davis-Putnam.

Exercițiu: definiți limbajul logicii propoziționale clasice în Prolog.

Începeți prin a defini:

- variabilele: `is_var(a).` `is_var(b).` ...
- operatorii (asociativi la dreapta): `nu`, `si`, `sau`, `imp`
 - `:- op(630, xfy, sau).`
 - `:- op(620, xfy, si).`
 - `:- op(640, xfy, imp).`
 - `:- op(610, fy, nu).`

Exemplu:

```
?- X= a si nu b.  
X = a si nu b.
```

Exercițiu: scrieți un predicat care sa întoarcă a true dacă argumentul este o formulă corectă.

Exemplu:

```
?- formula(nu nu a si b sau c).  
true.
```

Atenție! dacă formula nu este sintactic corectă se poate primi răspunsul false sau mesaj de eroare:

```
?- formula(a si sau).  
false.  
?- formula(a si sau a).  
ERROR: Syntax error: Operator expected
```

Practică

catch

```
?- formula(a si sau).  
false.  
?- formula(a si sau a).  
ERROR: Syntax error: Operator expected
```

Pentru a evita mesajele de eroare, putem defini:

```
test :- catch(read(X), Error, false),X.
```

```
?- test.  
|: formula(a imp a).  
true.
```

```
?- test.  
|: formula(a si sau a).  
false.
```


Exercițiu: scrieți un predicat care sa întoarcă a true dacă argumentul este o formulă corectă.

```
formula(X) :- is_var(X).  
formula(nu X) :- formula(X).  
formula(X sau Y) :- formula(X), formula(Y).  
formula(X si Y) :- formula(X), formula(Y).  
formula(X imp Y) :- formula(X), formula(Y).
```

Eliminarea implicației

Exercițiu: scrieți un predicat `inloc_imp(X,Y)` cu următorul efect: `Y` este `X` în care toate implicațiile au fost scrise folosind `sau` și `nu`.

Amintiți-vă că $\varphi \rightarrow \psi \sim \neg\varphi \vee \psi$

```
inloc_imp(X,X) :- is_var(X).
inloc_imp(nu X, nu X1) :- inloc_imp(X,X1).
inloc_imp(X sau Y, X1 sau Y1) :- inloc_imp(X, X1),
                                inloc_imp(Y, Y1).
inloc_imp(X si Y, X1 si Y1) :- inloc_imp(X, X1),
                                inloc_imp(Y, Y1).
inloc_imp(X imp Y, (nu X1) sau Y1) :- inloc_imp(X, X1),
                                      inloc_imp(Y, Y1).
```

```
?- inloc_imp(a imp b imp c, Y).
Y = nu a sau nu b sau c
```

Forma NNF

Fie φ o formulă din calculul propozițional clasic care nu conține implicații.

- Formula φ este în forma **NNF** dacă negația este numai pe variabile.

Example:

$p \wedge \neg q$ este în formă NNF

$\neg(p \vee q) \wedge r$ nu este în formă NNF

- Formula φ poate fi adusă la forma NNF folosind:

- regulile De Morgan

$$\neg(\varphi \vee \psi) \sim \neg\varphi \wedge \neg\psi,$$

$$\neg(\varphi \wedge \psi) \sim \neg\varphi \vee \neg\psi,$$

- principiului dublei negații

$$\neg\neg\psi \sim \psi$$

Forma NNF

Exercițiu: scrieți un predicat `nnf(X,Y)` astfel încât `Y` să fie `X` în formă NNF; vom presupune ca `X` nu conține implicații.

Exemplu:

```
?- nnf(nu nu a, Y).
```

```
Y = a .
```

```
?- nnf(a, Y).
```

```
Y = a .
```

```
?- nnf(a si nu nu b, Y).
```

```
Y = a si b .
```

```
?- nnf(nu (a sau b) si nu nu c, Y).
```

```
Y = (nu a si nu b)si c
```

Forma NNF

Exercițiu: scrieți un predicat `nnf(X,Y)` astfel încât `Y` să fie `X` în formă NNF; vom presupune ca `X` nu conține implicații.

```
nnf(X,X) :- literal(X).  
nnf(nu nu X, X).  
nnf(nu(X sau Y), X1 si Y1) :- nnf(nu X, X1), nnf(nu Y, Y1).  
nnf(nu(X si Y), X1 sau Y1) :- nnf(nu X, X1), nnf(nu Y, Y1).  
nnf(X sau Y, X1 sau Y1) :- nnf(X, X1), nnf(Y, Y1).  
nnf(X si Y, X1 si Y1) :- nnf(X, X1), nnf(Y, Y1).
```

Orice formulă poate fi adusa la FNC (conjuncție de disjuncții de literali) prin urmatoarele transformări:

1. înlocuirea implicațiilor și echivalențelor

$$\begin{aligned}\varphi \rightarrow \psi &\sim \neg\varphi \vee \psi, \\ \varphi \leftrightarrow \psi &\sim (\neg\varphi \vee \psi) \wedge (\neg\psi \vee \varphi)\end{aligned}$$

2. regulile De Morgan

$$\begin{aligned}\neg(\varphi \vee \psi) &\sim \neg\varphi \wedge \neg\psi, \\ \neg(\varphi \wedge \psi) &\sim \neg\varphi \vee \neg\psi,\end{aligned}$$

3. principiului dublei negații

$$\neg\neg\psi \sim \psi$$

4. distributivitatea

$$\begin{aligned}\varphi \vee (\psi \wedge \chi) &\sim (\varphi \vee \psi) \wedge (\varphi \vee \chi), \\ (\psi \wedge \chi) \vee \varphi &\sim (\psi \vee \varphi) \wedge (\chi \vee \varphi)\end{aligned}$$

Exercițiu: scrieți un predicat `cnf(X,Y)` astfel încât `Y` să fie `X` în formă cnf.

Exemplu:

```
?- cnf(a,Y).
```

```
Y = a .
```

```
?- cnf(a imp b,Y).
```

```
Y = nu a sau b .
```

```
?- cnf(a imp b imp c,Y).
```

```
Y = nu a sau nu b sau c .
```

```
?- cnf(a imp b imp (c si d),Y).
```

```
Y = (nu a sau nu b sau c)si(nu a sau nu b sau d)
```

Exercițiu: scrieți un predicat `cnf(X,Y)` astfel încât `Y` să fie `X` în formă `cnf`.

```
cnf(F,C) :- inloc_imp(F,F1), nnf(F1, F2), distribute(F2,C).
```


Exercițiu: scrieți un predicat `cnf(X,Y)` astfel încât `Y` să fie `X` în formă cnf.

```
cnf(F,C) :- inloc_imp(F,F1), nnf(F1, F2), distribute(F2,C).
```

Observăm că `F2` este o format din are negatia pe literali iar ca operatii binare numai disjunctii și conjunctii. Pentru a obține FNC trebuie să aplicăm distributivitatea.

```
distribute(A si B, A1 si B1) :- distribute(A, A1),  
                                distribute(B, B1).
```

```
distribute(A sau B, AB) :- distribute(A, A1),  
                           distribute(B, B1),  
                           distribute_cnf(A1, B1, AB).
```

```
distribute(A, A).
```

FNC

```
cnf(F,C) :- inloc_imp(F,F1), nnf(F1, F2), distribute(F2,C).  
  
distribute(A si B, A1 si B1) :- distribute(A, A1),  
                                distribute(B, B1).  
  
distribute(A sau B, AB) :-  distribute(A, A1),  
                             distribute(B, B1),  
                             distribute_cnf(A1, B1, AB).  
  
distribute(A, A).
```

Observăm că `distribute_cnf (F1,F2,R)` calculează în R forma normală conjunctivă pentru F1 sau F2, unde F1 și F2 sunt forme normale conjunctive.

```
distribute_cnf(A si B,C,D1 si D2) :- distribute_cnf(A, C, D1),  
                                     distribute_cnf(B, C, D2).  
distribute_cnf(C,A si B,D1 si D2) :- distribute_cnf(C, A, D1),  
                                     distribute_cnf(C, B, D2).  
  
distribute_cnf(A, B, A sau B).
```

Forma clauzală a unei FNC

Exercițiu: scrieți un predicat `toclausal(F,L)` astfel încât `L` să fie forma clauzală a lui `F`, unde `F` este **fnc**.

Exemplu:

```
?- toclausal((a sau nu b) si (nu c sau nu a) si d, L).
```

```
L = [[a, nu b], [nu c, nu a], [d]].
```

Forma clauzală a unei FNC

Exercițiu: scrieți un predicat `toclausal(F,L)` astfel încât `L` să fie forma clauzală a lui `F`, unde `F` este **fnc**.

Exemplu:

```
?- toclausal((a sau nu b) si (nu c sau nu a) si d, L).
```

```
L = [[a, nu b], [nu c, nu a], [d]].
```

```
toclausal(X,[X]):-literal(X).
```

```
toclausal(A si B,R) :- toclausal(A,RA), toclausal(B,RB),  
                        union(RA,RB,R).
```

```
toclausal(A sau B,[R]) :- literal(A), toclausal(B,[RB]),  
                          union([A],RB,R).
```

Clauze triviale

Exercițiu: scrieți un predicat `remove_trivial(LC,L)` unde `L` și `LC` sunt forme clauzale, `L` fiind obținută prin eliminarea clauzelor triviale din `LC`.

```
trivial(L) :- select(X,L,_), is_var(X), subset([X,nu X],L).

remove_trivial([],[]).
remove_trivial([C|L], R) :- trivial(C), remove_trivial(L,R).
remove_trivial([C|L], [C|R]) :- remove_trivial(L,R).

?- remove\tu trivial([[a,b, nu a],[c],[b ,d, e, nu d]], L).
L = [[c]]
```

Forma clauzală a unei formule

Exercițiu: combinând predicatele pe care le-ați scris până acum, scrieti un predicat `clausal_form(F,LC)` în care `F` este o formulă corectă (nu neapărat în formă `cnf`) iar `LC` este forma ei clauzală.

```
clausal_form(F,FC):- cnf(F,CNF), toclausal(CNF,LC),  
                      remove_trivial(LC,FC).
```

```
?- clausal_form(a si (b imp (nu a sau c)), LC).  
LC = [[a], [nu b, nu a, c]]
```

Separarea clauzelor

Exercițiu: scrieți două predicate `list1(LC,X,L1)` și `list2(LC,X,L2)` în care: `LC` este o formă clauzală, `X` este o variabila care apare în `LC`, `L1` este lista clauzelor care contin literalul `X`, `L2` este lista clauzelor care contin literalul `nu X`.

```
?- list1([[a],[nu b, nu a, c]],a, L1).  
L1 = [[a]].
```

```
?- list2([[a],[nu b, nu a, c]],a, L2).  
L2 = [[nu b, nu a, c]].
```

Separarea clauzelor

Exercițiu: scrieți două predicate `list1(LC,X,L1)` și `list2(LC,X,L2)` în care: `LC` este o formă clauzală, `X` este o variabilă care apare în `LC`, `L1` este lista clauzelor care conțin literalul `X`, `L2` este lista clauzelor care conțin literalul `nu X`.

```
?- list1([[a],[nu b, nu a, c]],a, L1).  
L1 = [[a]].
```

```
?- list2([[a],[nu b, nu a, c]],a, L2).  
L2 = [[nu b, nu a, c]].
```

```
list1(LC,X, L1):-findall(C,(member(C, LC), member(X,C)), L1).  
list2(LC,X, L2):-findall(C,(member(C, LC), member(nu X,C)), L2).
```


Rezoluția. SAT

Exercițiu: Scrieți un predicat `resolution(FC,Sat)` în care `FC` este o formă clauzală, iar `Sat` este rezultatul algoritmului Davis-Putnam.

Exemplu:

```
?- resolution([[a],[nu a, c], [d], [a, b]], Sat).  
Sat = [] . % lista vida de clauze
```

```
?- resolution([[a],[nu a, c], [nu c]], Sat).  
Sat = [[]]. % lista contine numai clauza vida
```

Rezoluția. SAT

Exercițiu: Scrieți un predicat `resolution(FC,Sat)` în care `FC` este o formă clauzală, iar `Sat` este rezultatul algoritmului Davis-Putnam.

Exemplu:

```
?- resolution([[a],[nu a, c], [d], [a, b]], Sat).  
Sat = [] . % lista vida de clauze
```

```
?- resolution([[a],[nu a, c], [nu c]], Sat).  
Sat = [[]]. % lista contine numai clauza vida
```

Verificarea SATisfiabilității

```
sat(Formula,Sat):- clausal_form(Formula,FC),  
                   resolution(FC, Sat).
```

Rezoluția

Exercițiu: scrieți un predicat `resolution(FC,Sat)` în care `FC` este o formă clauzală, iar `Sat` este rezultatul algoritmului Davis-Putnam.

```
resolution([],[]).  
resolution([[]],[[]]).  
resolution(LC, Sat) :-  
    choose(V,LC),  
    list1(LC, V, L1), list2(LC, V, L2),  
    findall(Rez, resolvent(L1,L2,V,Rez),LR1),  
    remove_trivial(LR1,LR),  
    subtract(LC,L1,LC1), subtract(LC1,L2, LC2),  
    append(LC2,LR, LC3),  
    list_to_set(LC3,LC4),  
    resolution( LC4,Sat).
```

```
choose(V,LC):- ...  
resolvent(L1,L2,V,Rez):- ...
```

Rezoluția (continuare)

Exercițiu: scrieți un predicat `resolution(FC,Sat)` în care `FC` este o formă clauzală, iar `Sat` este rezultatul algoritmului Davis-Putnam.

```
% choose alege o variabila V dintr-o clauza C din LC
```

```
choose(V, LC) :- select(C,LC,_), select(L,C,_),  
                ((L= nu X)-> (V=X);(V=L)).
```

```
% Rez este rezultatul aplicarii regulii rezolutiei pentru  
% variabila X pentru o clauza din L1 si una din L2
```


```
resolvent(L1,L2, X,Rez):- member(C1,L1), member(C2,L2),  
                          subtract(C1, [X], C11),  
                          subtract(C2, [nu X], C22),  
                          append(C11,C22, Rez1),  
                          list_to_set(Rez1,Rez).
```

Verificarea SATisfiabilității

```
sat(Formula,Sat):- clausal_form(Formula,FC),  
                  resolution(FC, Sat).
```

```
?- sat( nu a imp b si nu a, Sat).  
Sat = []
```

```
?- sat(a si nu c si (nu a sau c), Sat).  
Sat = [[]]
```



Testul de Prolog are loc luni, 14.04.2025, ora 14.00!

Succes la testare!