1. Limbajul Prolog

- Limbajul Prolog (PROgrammation en LOGique) a fost elaborat la Universitatea din Marsilia în jurul anului 1970, ca instrument pentru programarea şi rezolvarea problemelor ce implicau reprezentări simbolice de obiecte şi relaţii dintre acestea.

- Prolog are un câmp de aplicaţii foarte larg: baze de date relaţionale, inteligenţă artificială, logică matematică, demonstrarea de teoreme, sisteme expert, rezolvarea de probleme abstracte sau ecuaţii simbolice, etc.

- Există standardul ISO-Prolog.

- Nu există standard pentru programare orientată obiect în Prolog, există doar extensii: TrincProlog, SWI-Prolog.

- Vom studia implementarea SWI-Prolog – sintaxa e foarte apropiată de cea a standardului ISO-Prolog.

    - Turbo Prolog, Visual Prolog, GNUProlog, Sicstus Prolog, Parlog, etc.

- SWI-Prolog – 1986
    - oferă o interfață bidirecțională cu limbajele C și Java
    - folosește XPCE – un sistem GUI orientat obiect
    - *multithreading* – bazat pe suportul *multithreading* oferit de limbajul standard C.

Program Prolog

- caracter descriptiv: un program Prolog este o colecţie de definiţii ce descriu relaţii sau funcţii de calculat – reprezentări simbolice de obiecte și relații între obiecte. Soluţia problemelor nu se mai vede ca o execuţie pas cu pas a unei secvenţe de instrucţiuni.

    - program – colecție de declarații logice, fiecare fiind o clauză Horn de forma *p*, $p \rightarrow q$, $p_1 \wedge p_2 ... \wedge p_n \rightarrow q$
    - **concluzie** de demonstrat – de forma $p_1 \wedge p_2 ... \wedge p_n$

- **Structura de control** folosită de interpretorul Prolog

    - Se bazează pe declarații logice numite **clauze**
        - **fapt** – ceea ce se cunoaște a fi adevărat
        - **regulă** - ce se poate deduce din fapte date (indică o concluzie care se știe că e adevărată atunci când alte concluzii sau fapte sunt adevărate)
    - **Concluzie** ce trebuie demonstrată - GOAL
        - Prolog folosește *rezoluția* (liniară) pentru a demonstra dacă concluzia (teorema) este adevărată sau nu, pornind de la ipoteza stabilită de faptele și regulile definite (axiome).
        - Se aplică raționamentul înapoi pentru a demonstra concluzia
        - Programul este citit de sus în jos, de la dreapta la stânga, căutarea este în adâncime (*depth-first*) și se realizează folosind **backtracking.**

- $p \rightarrow q$ se transcrie în Prolog folosind clauza q :- p. (q if p.)

- $\wedge$ se transcrie în Prolog folosind ","
  - $p_1 \wedge p_2 ... \wedge p_n \rightarrow q$ se transcrie în Prolog folosind clauza q :- $p_1, p_2, ..., p_n$.
- $\vee$ se transcrie în Prolog folosind ";" sau o clauză separată.
  - $p_1 \vee p_2 \rightarrow q$ se transcrie în Prolog
    - folosind clauza q :- $p_1; p_2$.

      sau
    - folosind 2 clauze separate

      q :- p_1.

      q :- p_2.

Exemple

- **Logică**

  $\forall x\, p(x) \wedge q(x) \rightarrow r(x)$

  $\forall x\, w(x) \vee s(x) \rightarrow p(x)$

  $\forall x\, t(x) \rightarrow s(x) \wedge q(x)$

  $t(a)$

  $w(b)$

  Concluzie

  $r(a)$

  $p(a)$

  (SWI-)Prolog

  $r(X)$:- $p(X), q(X)$.

  $p(X)$:- $w(X)$.

  $p(X)$:- $s(X)$.

  $s(X)$:- $t(X)$.

  $q(X)$:- $t(X)$.

  $t(a)$.

  $w(b)$.

  Goal

  ? $r(a)$.

  true

  ? $p(b)$.

  false

- **Logică**

  $\forall x\, s(x) \rightarrow p(x) \vee q(x)$

  Prolog

  ????

2. Elemente de bază ale limbajului SWI-Prolog

1. **Termen**
   - **SIMPLU**
     a. constantă
     - simbol (*symbol*)
       - secvență de litere, cifre, _
       - începe cu **literă mică**
     - număr =întreg, real (*number*)

- șir de caractere (*string*): ‘text’ (caracter: ‘c’, ‘\t’,...)

ATOM = SIMBOL + STRING +ȘIR-DE-CARACTERE-SPECIALE + [ ] (lista vidă)

- caractere speciale + * / < > = : . & _ ~

b. <u>variabilă</u>

- secvență de litere, cifre, _
- începe cu **literă mare**
- variabila anonimă este reprezentată de caracterul underline (_).

- **COMPUS** (a se vedea Secțiunea 13).
  - listele (*list*) sunt o clasă specială de termeni compuși

2. Comentariu

% Acesta este un comentariu
/* Acesta este un comentariu */

3. Predicat

a). standard (ex: **fail, number,** ...)
b). utilizator

- *nume* [(*obiect*[, *obiect*....)]

numele simbolic al relației

Tipuri

1. **number** (integer, real)
2. **atom** (symbol, string, șir-de-caractere-speciale)
3. **list** (secvență de elemente) specificat ca **list**=tip_de_bază*

ex. listă (omogenă) formată din numere întregi [1,2,3]

% definire tip: el=integer list=el*

!!! lista vidă [] este singura listă care e considerată în Prolog atom.

<u>Convenții</u>.

- În SWI-Prolog nu există declarații de predicate, nici declarații de domenii/tipuri (ex. ca în Turbo-Prolog).
- *Specificarea* unui predicat
  - % definire tipuri, dacă e cazul
  - % *nume* [(param$_1$:tip$_1$[,param$_2$:tip$_2$...)]
  - % modelul de flux al predicatului (i, o, ...) - vezi Secțiunea 4
  - % param$_1$ - semnificația parametrului 1

- % $param_2$ - semnificația parametrului 2
  ….

4. **Clauza**
   - fapt
     - relație între obiecte
     - *nume_predicat* [(*obiect* [, *obiect*....)]
   - regula
     - permite deducere de fapte din alte fapte

Exemplu:

fie predicatele

tata(X, Y) reprezentând relația “Y este tatăl lui X”
mama(X, Y) reprezentând relația “Y este mama lui X”

și următoarele fapte corespunzătoare celor două predicate:

```
mama(a,b).
mama(e,b).
tata(c,d).
tata(a,d).
```

Se cere: folosind definițiile anterioare să se definească predicatele

parinte(X, Y) reprezentând relația “Y este părintele lui X”
frate(X, Y) reprezentând relația “Y este fratele lui X”

Clauze în SWI-Prolog

```
parinte(X,Y) :-tata(X,Y).
parinte(X,Y) :-mama(X,Y).
frate(X,Y) :-parinte(X,Z),parinte(Y,Z),X\=Y.
```

5. **Intrebare** (goal)

- e de forma *predicat*$_1$ [(*obiect* [, *obiect*....)], *predicat*$_2$ [(*obiect* [, *obiect*....)].... **.**
- **true**, **false**
- **CWA – Closed World Assumption**

Folosind definițiile anterioare, formulăm următoarele întrebări:

```
?- parinte(a,b).
true.
```

```
?- parinte(a,X).
X=d;
X=b.
```

```
? - parinte(a,f).
false.
```

```
?- frate(a,X).
X=c;
X=e.
```

```
?- frate(a,_).
true.
```

3. “Matching”. Cum îşi primesc valori variabilele?

Prolog nu are instrucţiuni de atribuire. Variabilele în Prolog îşi primesc valorile prin **potrivire** cu constante din fapte sau reguli.

Până când o variabilă primeşte o valoare, ea este **liberă** (free); când variabila primeşte o valoare, ea este **legată** (bound). Dar ea stă legată atâta timp cât este necesar pentru a obţine o soluţie a problemei. Apoi, Prolog o dezleagă, face backtracking şi caută soluţii alternative.

Observaţie. Este important de reţinut că nu se pot stoca informaţii prin atribuire de valori unor variabile. Variabilele sunt folosite ca parte a unui proces de potrivire, nu ca un tip de stocare de informaţii.

Ce este o potrivire?

Iată câteva reguli care vor explica termenul 'potrivire':

1. Structuri identice se potrivesc una cu alta
   - p(a, b) se potriveste cu p(a, b)
2. De obicei o potrivire implică variabile libere. Dacă X e liberă,
   - p(a, X) se potriveste cu p (a, b)
   - X este legat la b.
3. Dacă X este legat, se comportă ca o constantă. Cu X legat la b,
   - p(a, X) se potriveste cu p(a,b)
   - p(a, X) NU se potriveste cu p(a, c)
4. Două variabile libere se potrivesc una cu alta.
   - p(a, X) se potriveste cu p(a, Y)

Observaţie. Mecanismul prin care Prolog încearcă să ‘potrivească’ partea din întrebare pe care doreste să o rezolve cu un anumit predicat se numeste **unificare**.

4. Modele de flux

În Prolog, legările de variabile se fac în două moduri: la intrarea în clauză sau la ieşirea din clauză. Direcţia în care se leagă o valoare se numeşte ‘**model de flux**’. Când o variabilă este dată la intrarea într-o clauză, aceasta este un parametru de intrare (i), iar când o variabilă este dată la ieşirea dintr-o clauză, aceasta este un parametru de ieşire (o). O anumită clauză poate să aibă mai multe modele de flux. De exemplu clauza

factorial (N, F)

poate avea următoarele modele de flux:

- (i,i) - verifică dacă N! = F;
- (i,o) - atribuie F := N!;
- (o,i) - găseşte acel N pentru care N! = F.

Observaţie. Proprietatea unui predicat de a funcţiona cu mai multe modele de flux depinde de abilitatea programatorului de a programa predicatul în mod corespunzător.

5. Sintaxa regulilor

Regulile sunt folosite în Prolog când un fapt depinde de succesul (veridicitatea) altor fapte sau succesiuni de fapte. O regulă Prolog are trei părti: capul, corpul şi simbolul if (:-) care le separă pe primele două.

Iată sintaxa generică a unei reguli Turbo Prolog:

```
capul regulii :-
        subgoal,
        subgoal,
        ...,
        subgoal.
```

Fiecare subgoal este un apel la un alt predicat Prolog. Când programul face acest apel, Prolog testează predicatul apelat să vadă dacă poate fi adevarat. Odată ce subgoal-ul curent a fost satisfăcut (a fost găsit adevărat), se revine şi procesul continuă cu următorul subgoal. Dacă procesul a ajuns cu succes la punct, regula a reuşit. Pentru a utiliza cu succes o regulă, Prolog trebuie să satisfacă toate subgoal-urile ei, creând o mulţime consistentă de legări de variabile. Dacă un subgoal eşuează (este găsit fals), procesul revine la subgoal-ul anterior şi caută alte legări de variabile, şi apoi continuă. Acest mecanism se numeşte **backtracking**.

6. Operatori de egalitate

X=Y — verifică dacă X și Y pot fi unificate

- Dacă X este variabilă liberă și Y legată, sau Y este variabilă liberă și X e legată, propoziţia este satisfăcută unificând pe X cu Y.
- Dacă X și Y sunt variabile legate, atunci propoziţia este satisfăcută dacă relaţia de egalitate are loc.

```
?- [a,b]=[a,b].
true.

?- [X,Y]=[a,b].
X = a,
Y = b.

?- [a,b]= [X,Y].
X = a,
Y = b.
```

X\=Y
\+ X=Y — verifică dacă X și Y nu pot fi unificate

```
?- [X,Y,Z]\ =[a,b].
true.

?- [X,Y]\=[a,b].
false.

?- [a,b]\= [X,Y].
false.

?- \+ a=a.
false.

?- \+ [X,Y]=[a,b].
false.

?- \+ [a,b]= [X,Y,Z].
true.
```

X ==Y verifică dacă X și Y sunt legate la aceeași valoare.

?- [2,3]==[2,3].
true.

?- a==a.
true.

?- R==1.
false.

X \== Y verifică dacă X și Y nu au fost legate la aceeași valoare.

?- [2,3]\==[3,2].
true.

?- a\==a.
false.

?- R\==1.
true.

7. Operatori aritmetici

!!! Important

- 2+4 e doar o structură, utilizarea sa nu efectuează adunarea
- Utilizarea 2+4 nu e aceeași ca utilizarea lui 6.

Operatori aritmetici

=, \=, ==, \== A se vedea secțiunea 6.

?- 2+4=6.
false.

?- 2+4\=6.
true.

?- 6==6.
true.

?-6\=7.
true.

?- 6==2+4.
false.

?- 2+4=2+4.
true.

?- 2+4=4+2.
false.

?- X= 2+4-1.
X=2+4-1.

=:=

- testează egalitatea aritmetică
- forțează evaluarea aritmetică a ambelor părți
- operanzii trebuie să fie numerici
- variabilele sunt LEGATE

=\= testează operatorul aritmetic "diferit"

?- 2+4=:=6.
true.

?- 2+4=\=7.
true.

?- 6=:=6.
true.

is

- partea dreaptă este LEGATĂ și numerică
- partea stângă trebuie să fie o variabilă
- dacă variabila este legată, verifică egalitatea numerică (ca și =:=)
- dacă variabila nu este legată, evaluează partea dreaptă și apoi variabila este legată de rezulatul evaluării

?- X is 2+4-1.
X=5

?- X is 5.
X=5

Inegalități

<	mai mic
=<	mai mic sau egal
>	mai mare
>=	mai mare sau egal

- evaluează ambele părți
- variabile LEGATE

?- 2+4=<5+2.	?- 2+4=\=7.	?- 6=:=6.
true.	**true.**	**true.**

Câteva funcţii aritmetice predefinite SWI-Prolog

X mod Y mod(X, Y)	întoarce restul împărţirii lui X la Y
X div Y div(X, Y)	întoarce câtul împărţirii lui X la Y
abs(X)	întoarce valoarea absolută a lui X
sqrt(X)	întoarce rădăcina pătrată a lui X
round(X)	întoarce valoarea lui X rotunjită spre cel mai apropiat întreg (round(2.56) este 3, round (2.4) este 2)

...

8. Predicate predefinite

var(X)	= adevărat dacă X e liberă, fals dacă e legată
number(X)	= adevărat dacă X e legată la un număr
integer(X)	= adevărat dacă X e legată la un număr întreg
float(X)	= adevărat dacă X e legată la un număr real
atom(X)	= adevărat dacă X e legată la un atom
atomic(X)	= atom(X) or number(X)

....

9. Predicatul „findall“ (determinarea tuturor soluțiilor)

Prolog oferă o modalitate de a găsi toate soluţiile unui predicat în acelaşi timp: predicatul **findall**, care colectează într-o listă toate soluţiile găsite.

findall (arg1, arg2, arg3)

Acesta are următoarele argumente:

- primul argument specifică argumentul din predicatul considerat care trebuie colectat în listă;

- al doilea argument specifică predicatul de rezolvat;
- al treilea argument specifică lista în care se vor colecta soluţiile.

10. Negație - “not”, “\+”

not(*subgoal*(Arg1, ..., ArgN)) adevărat dacă *subgoal* eșuează(nu se poate demonstra că este adevărat)

\+ *subgoal*(Arg1, ..., ArgN)

?- \+ (2 = 4).
true.

?- not(2 = 4).
true.

11. Predicatul ! (cut) – “tăietura”

Turbo Prolog conţine predicatul cut (!) folosit pentru a preveni backtracking-ul. Când se procesează predicatul !, apelul reuşeşte imediat şi se trece la subgoalul următor. O dată ce s-a trecut peste o tăietură, nu este posibilă revenirea la subgoal-urile plasate înaintea ei şi nu este posibil backtracking-ul la alte reguli ce definesc predicatul în execuţie.

Există două utilizări importante ale tăieturii:

1. Când ştim dinainte că anumite posibilităţi nu vor duce la soluţii, este o pierdere de timp să lăsăm sistemul să lucreze. În acest caz, tăietura se numeşte **tăietură verde**.
2. Când logica programului cere o tăietură, pentru prevenirea luării în consideraţie a subgoal-urilor alternative, pentru a evita obţinerea de soluţii eronate. În acest caz, tăietura se numeşte **tăietură roşie**.

Prevenirea backtracking-ului la un subgoal anterior

În acest caz tăietura se utilizează astfel: r1 :- a, b, !, c.

Aceasta este o modalitate de a spune că suntem multumiţi cu primele soluţii descoperite cu subgoal-urile a şi b. Deşi Prolog ar putea găsi mai multe soluţii prin apelul la c, nu este autorizat să revină la a şi b. De-asemenea, nu este autorizat să revina la altă clauză care defineşte predicatul r1.

Prevenirea backtracking-ului la următoarea clauză

Tăietura poate fi utilizată pentru a-i spune sistemului Prolog că a ales corect clauza pentru un predicat particular. De exemplu, fie codul următor:

```
r(1) :- !, a, b, c.
r(2) :- !, d.
r(3) :- !, e.
r(_) :- write("Aici intră restul apelurilor").
```

Folosirea tăieturii face predicatul r determinist. Aici, Prolog apelează predicatul r cu un argument întreg. Să presupunem că apelul este r(1). Prolog caută o potrivire a apelului. O

găseşte la prima clauză. Faptul că imediat după intrarea în clauză urmează o tăietură, împiedică Prolog să mai caute şi alte potriviri ale apelului r(1) cu alte clauze.

Observaţie. Acest tip de structură este echivalent cu o instrucţiune de tip *case* unde condiţia de test a fost inclusă în capul clauzei. La fel de bine s-ar fi putut spune şi

```
r(X) :- X = 1, !, a, b, c.
r(X) :- X = 2, !, d.
r(X) :- X = 3, !, e.
r(_) :- write("Aici intra restul apelurilor").
```

Notă. Deci, următoarele secvenţe sunt echivalente:

```
case X of
        v1: corp1;
        v2: corp2;
        ...
        else corp_else.
```

```
predicat(X) :- X = v1, !, corp1.
predicat(X) :- X = v2, !, corp2.
...
predicat(X) :- corp_else.
```

De asemenea, următoarele secvenţe sunt echivalente:

```
if cond1 then
        corp1
else if cond2 then
        corp2
...
else
        corp_else.
```

```
predicat(...) :-
        cond1, !, corp1.
predicat(...) :-
        cond2, !, corp2.
...
predicat(...) :-
        corp_else.
```

12. Predicatul "fail"

Valoarea lui *fail* este eşec. Prin aceasta el încurajează backtracking-ul. Efectul lui este acelaşi cu al unui predicat imposibil, de genul 2 = 3. Fie următorul exemplu:

```
predicat(a, b).
predicat(c, d).
predicat(e, f).
toate :-
        predicat(X, Y),
        write(X),write(Y),nl,
        fail.
toate1 :-
        predicat(X, Y),
        write(X),write(Y),nl.
```

Predicatele **toate** şi **toate1** sunt fără parametri, şi ca atare sistemul va trebui să răspundă dacă există X şi Y astfel încât aceste predicate să aibă loc.

```
?- toate.
ab
cd
ef
false.
```

```
?-toate1.
ab
true;
cd
true;
ef
true.
```

```
?-predicat(X,Y).
X = a,
Y = b ;
X = c,
Y = d ;
X = e,
Y = f.
```

Faptul că apelul predicatului **toate** se termină cu *fail* (care eşuează întotdeauna) obligă Prolog să înceapă backtracking prin corpul regulii **toate**. Prolog va reveni până la ultimul apel care poate oferi mai multe soluţii. Predicatul **write** nu poate da alte soluţii, deci revine la apelul lui **predicat**.

Observaţii.

- Acel **false** de la sfârşitul soluţiilor semnifică faptul că predicatul 'toate' nu a fost satisfăcut.
- Dupa *fail* nu are rost să puneţi nici un predicat, deoarece Prolog nu va ajunge să-l execute niciodată.

Notă. Secvenţele următoare sunt echivalente:

```
cât timp condiţie execută
        corp
```

```
predicat :-
        condiţie,
        corp,
        fail.
```

13. Obiecte simple şi obiecte compuse

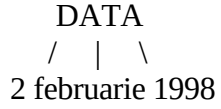
Obiecte simple

Un obiect simplu este fie o variabilă, fie o constantă. O constantă este fie un caracter, fie un număr, fie un atom (simbol sau string).

Variabilele Prolog sunt locale, nu globale. Adică, dacă două clauze conţin fiecare câte o variabilă numită X, cele două variabile sunt distincte şi, de obicei, nu au efect una asupra celeilalte.

Obiecte compuse si functori

Obiectele compuse ne permit să tratăm mai multe informaţii ca pe un singur element, într-un astfel de mod încât să-l putem utiliza şi pe bucăţi. Fie, de exemplu, data de *2 februarie 1998*. Constă din trei informaţii, ziua, luna şi anul, dar e util să o tratăm ca un singur obiect cu o structură arborescentă:



Acest lucru se poate face scriind obiectul compus astfel:

data(2, “februarie”, 1998)

Aceasta seamănă cu un fapt Prolog, dar nu este decât un obiect (o dată) pe care îl putem manevra la fel ca pe un simbol sau număr. Din punct de vedere sintactic, începe cu un nume (sau **functor**, în acest caz cuvântul **data**) urmat de trei argumente.

Notă. Functorul în Prolog nu este acelaşi lucru cu funcţia din alte limbaje de programare. Este doar un nume care identifică un tip de date compuse şi care ţine argumentele laolaltă.

Argumentele unei date compuse pot fi chiar ele compuse. Iată un exemplu:

naştere(persoana(“Ioan”, “Popescu”), data(2, “februarie”, 1918))

Unificarea obiectelor compuse

Un obiect compus se poate unifica fie cu o variabila simplă, fie cu un obiect compus care se potriveşte cu el. De exemplu,

data(2, “februarie”, 1998)

se potriveşte cu variabila liberă X şi are ca rezultat legarea lui X de data(...). De asemenea, obiectul compus de mai sus se potriveşte şi cu

data(Zi, Lu, An)

şi are ca rezultat legarea variabilei Zi de valoarea 2, a variabilei Lu de valoarea “februarie” şi a variabilei An de valoarea 1998.

Observaţii

- Convenim să folosim următoarea declarație pentru *specificarea* unui domeniu cu alternative

  % domeniu = alternativa$_1$(dom, dom, ..., dom);
  % alternativa$_2$(dom, dom, ..., dom);
  % ...
- Functorii pot fi folosiți pentru controla argumentele care pot avea tipuri multiple

  % element = i(integer); r(real); s(string)

14. Optimizarea prin recursivitate de coadă (tail recursion)

Recursivitatea are o mare problemă: consumă multă memorie. Dacă o procedură se repetă de 100 ori, 100 de stadii diferite ale execuţiei procedurii (cadre de stivă) sunt memorate.

Totuşi, există un caz special când o procedură se apelează pe ea fară să genereze cadru de stivă. Dacă procedura apelatoare apelează o procedură ca ultim pas al sau (după acest apel urmează punctul). Când procedura apelată se termină, procedura apelatoare nu mai are altceva de făcut. Aceasta înseamnă că procedura apelatoare nu are sens să-şi memoreze stadiul execuţiei, deoarece nu mai are nevoie de acesta.

Funcţionarea recursivităţii de coadă

Iată două reguli depre cum să faceţi o recursivitate de coadă:

1. Apelul recursiv este ultimul subgoal din clauza respectivă.
2. Nu există puncte de backtracking mai sus în acea clauză (adică, subgoal-urile de mai sus sunt deterministe).

Iată un exemplu:

```
tip(N) :-
        write(N),
        nl,
        Nou is N + 1,
        tip(Nou).
```

Această procedură foloseşte recursivitatea de coadă. Nu consumă memorie, şi nu se opreşte niciodată. Eventual, din cauza rotunjirilor, de la un moment va da rezultate incorecte, dar nu se va opri.

Exemple greşite de recursivitate de coadă

Iată cateva reguli despre cum să NU faceţi o recursivitate de coadă:

1. Dacă apelul recursiv nu este ultimul pas, procedura nu foloseşte recursivitatea de coadă.

Exemplu:

```
tip (N) :-
        write(N),
        nl,
        Nou is N + 1,
        tip (Nou),
        nl.
```

2. Un alt mod de a pierde recursivitatea de coadă este de a lăsa o alternativă neîncercată la momentul apelului recursiv.

Exemplu:

```
tip(N) :-
        write(N),
        nl,
        Nou is N + 1,
        tip(Nou).
tip(N) :-
        N < 0,
        write('N este negativ.').
```

Aici, prima clauză se apelează înainte ca a doua să fie încercată. După un anumit număr de paşi intră în criză de memorie.

3. Alternativa neîncercată nu trebuie neaparat să fie o clauza separată a procedurii recursive. Poate să fie o alternativă a unei clauze apelate din interiorul procedurii recursive.

Exemplu:

```
tip (N) :-
        write(N),
        nl,
        Nou is N + 1,
        verif(Nou),
        tip(Nou).
verif(Z) :- Z >= 0.
verif(Z) :- Z < 0.
```

Dacă N este pozitiv, prima clauză a predicatului **verif** a reuşit, dar a doua nu a fost încercată. Deci, **tip** trebuie să-şi pastreze o copie a cadrului de stivă.

Utilizarea tăieturii pentru păstrarea recursivităţii de coadă

A doua şi a treia situaţie de mai sus pot fi înlăturate dacă se utilizează tăietura, chiar dacă există alternative neîncercate.

Exemplu la situaţia a doua:

```
tip (N) :-
        N>= 0,
        !,
        write(N),
        nl,
        Nou = N + 1,
        tip(Nou).
tip(N) :-
        N < 0,
        write("N este negativ.").
```

Exemplu la situaţia a treia:

```
tip(N) :-
        write(N),
        nl,
        Nou = N + 1,
        verif(Nou),
        !,
        tip(Nou).
verif(Z) :- Z >= 0.
verif(Z) :- Z < 0.
```