F.I.A. **Cătălin Stoean**
Laboratorul numărul 3

Unificarea şi recursivitatea

Unificarea

Unificarea reprezintă modul în care Prologul realizează potrivirile între termeni. La prima vedere, procesul de unificare pare explicat de fraza *obiectul ţintă poate fi potivit obiectului sursă*. Presupunerea implicită este că numai obiectul ţintă este modificat. Dar nu se întâmplă astfel: se încearcă atât modificarea obiectului ţintă, cât şi a obiectului sursă.

De exemplu, dacă se încearcă potrivirea termenului **autor(mihai, X)** cu termenul **autor(Y, eminescu)** am fi tentaţi să credem că unica substituţie care se face este *X/eminescu*. Unificarea realizează însă ambele substituţii: *X/eminescu* şi *Y/mihai*. În urma acestui proces, ambii termeni arată astfel: **autor(mihai, eminescu)**.

<u>Concluzie</u>: procesul de unificare se realizează în ambele sensuri.

Exemple:

X = marian.	întoarce rezultat pozitiv (*Yes*).
marian = andrei	întoarce *No* pentru că nu pot fi potriviţi doi atomi distincţi.
X = marian, X = Y.	X = marian, Y = marian; întoarce *Yes*.
X = barbat(marian).	întoarce *Yes*.
X = Y.	întoarce *Yes*, iar daca mai târziu una din variabile ia o valoare şi cealaltă o va avea.

Interogări

Introduceţi următoarele interogări şi asiguraţi-vă că înţelegeţi de ce unificarea în unele cazuri reuşeşte, iar în unele nu:

- `2 + 1 = 3.`
- `f(X, a) = f(a, X).`
- marian = marian.
- place(maria, X) = place(X, andrei).
- f(X, Y) = f(P, P).

Recursivitatea

Programarea în Prolog depinde foarte mult de această tehnică numită *recursivitate*. În principiu, recursivitatea implică definirea unui predicat în funcţie de el însuşi. Cheia care ne asigură că această tehnică are sens constă în aceea că întotdeauna trebuie să definim predicatul la o scală mai mică. Recursia de la nivelul algoritmilor este echivalentă cu *demonstrarea prin inducţie* din matematică.

O definiţie recursivă (în orice limbaj, nu numai în Prolog) trebuie să aibă întotdeauna cel puţin două părţi: o condiţie elementară şi o parte recursivă.

Condiția elementară defineşte un caz simplu, care ştim că este întotdeauna adevărat. *Partea recursivă* simplifică problema eliminând inițial un anumit grad de complexitate şi apoi apelându-se ea însăşi. La fiecare nivel, condiția elementară este verificată: dacă s-a ajuns la ea, recursivitatea se încheie; altfel, recursivitatea continuă.

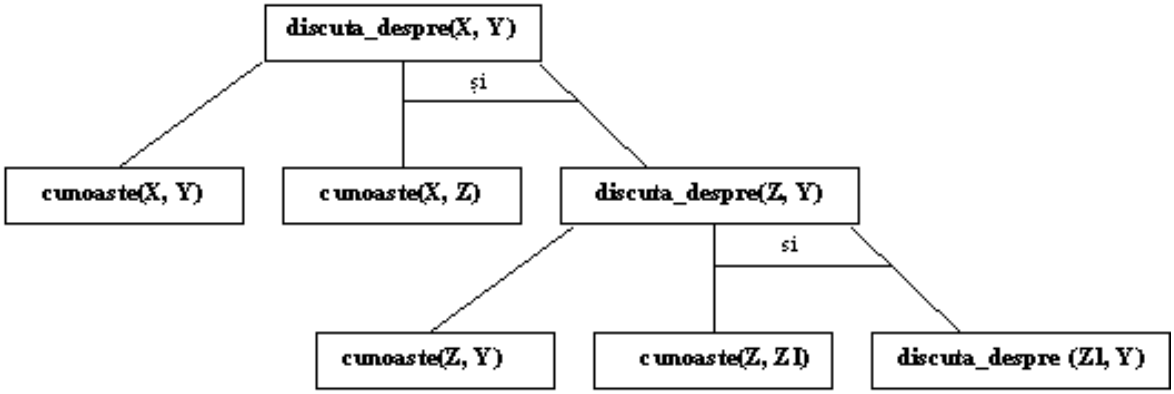
Vom ilustra recursivitatea prin următorul exemplu:

```
discuta_despre(A, B) :- cunoaste(A, B).
discuta_despre(X, Y) :- cunoaste(X, Z), discuta_despre(Z, Y).
```

Aceasta o putem explica într-o frază astfel:

Poți discuta despre cineva dacă o cunoşti pe persoana respectivă sau daca cunoşti pe cineva care discută despre ea.

Ajungerea la o soluție pentru o problemă dată depinde de posibilitatea de a opri recursivitatea la un anumit punct. Pentru a înțelege mai bine cum funcționează recursivitatea putem să ne imaginăm că pentru exemplul de mai sus, se formează următorul arbore care are clauze ca noduri:



Iata programul Prolog pe care îl puteți testa:

```
cunoaste(maria, ana).
cunoaste(ana, mircea).
cunoaste(mircea, mihai).

discuta_despre(A, B) :- cunoaste(A, B).
discuta_despre(X, Y) :- cunoaste(X, Z), discuta_despre(Z, Y).
```

Folosiți următoarea interogare:

```
? - discuta_despre(X, Y), write(X), write(' discuta despre '), write(Y), nl, fail.
```

Explicați rezultatul acestei interogări.

Exerciții rezolvate

1. **Calculați factorialul unui număr.**

Vom rezolva această problemă folosind următoarea funcție recursivă:

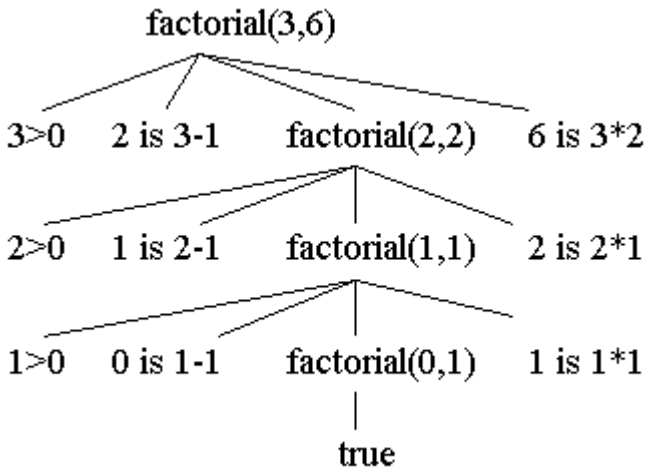
$$fact(x) = \begin{cases} 1, \text{ dacă } x = 0; \\ fact(x - 1) * x, \text{ altfel} \end{cases}$$

În exemplul de față, putem recunoaște *condiția elementară* ca fiind fact(0) = 1; aceasta este condiția care va stopa recursivitatea. Cunoscând această condiție, putem scrie programul:

```
factorial(0,1).
factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1), F is N * F1.
```

Pentru a calcula factorialul numărului 3, vom face o interogare după cum urmează:
? – factorial(3, F).
Răspunsul va fi: F = 6.

Următorul arbore este construit pentru interogarea *factorial(3, F)*. Ca noduri are clauze care nu conțin variabile libere, ci instanțe ale acestora:



O alta posibilitate de a realiza acest program este aceea de a folosi un predicat cu trei argumente astfel:

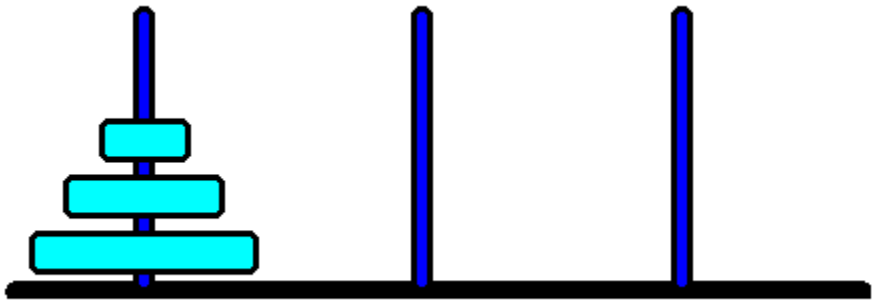
```
factorial(0,F,F).
factorial(N,A,F) :- N > 0, A1 is N*A, N1 is N -1, factorial(N1,A1,F).
```

Interogarea o vom realiza în felul următor:
? – factorial(3, 1, F).
Vom primi răspunsul F = 6.

Iată cum funcționează acest program: al doilea argument este cel în care calculăm rezultatele parțiale; el pornește de la valoarea 1 (cea dată în cadrul interogării). În final, variabila care reprezintă ultimul argument, va fi *unificată* cu variabila din cel de-al doilea argument, variabilă care va conține rezultatul final.

2. Rezolvați problema turnurilor din Hanoi.

Scopul acestui puzzle este de a muta n discuri de pe bara din stânga pe bara din dreapta folosind bara din centru ca pe una auxiliară. Important este că un disc mai mare nu poate fi aşezat pe un disc mai mic si la un moment dat poate fi mutat numai unul. Imaginea următoare arată care este configurația de pornire pentru n = 3.



În continuare, programul recursiv care rezolvă acest puzzle:

```
muta(1, X, Y, _) :- write('Muta discul din '), write(X), write(' in '), write(Y), nl.
muta(N,X,Y,Z) :- N > 1, M is N - 1, muta(M,X,Z,Y), muta(1,X,Y,Z), muta(M,Z,Y,X).
```

Interogarea va arăta astfel:

```
? – muta(3, stanga, dreapta, centru).
Muta discul din stanga in dreapta
Muta discul din stanga in centru
Muta discul din dreapta in centru
Muta discul din stanga in dreapta
Muta discul din centru in stanga
Muta discul din centru in dreapta
Muta discul din stanga in dreapta
Yes
```

Prima clauză din program se referă la mutarea unui singur disc. Cea de a doua arată cum poate fi obținută recursiv o soluție. Să vedem mai bine cum raționează sistemul atunci când avem interogarea *muta(3, stanga, dreapta, centru)*.

Avem N = 3, X = stanga, Y = dreapta, Z = centru:

muta(3, stanga, dreapta, centru) are loc dacă:

- muta(2, stanga, centru, dreapta) şi (notez clauza cu *)
- muta(1, stanga, dreapta, centru) şi
- muta(2, centru, dreapta, stanga). (notez clauza cu **)

Mai departe, aceasta se poate scrie desfăşurat astfel:

muta(3, stanga, dreapta, centru) are loc dacă:

- muta(1, stanga, dreapta, centru) şi
- muta(1, stanga, centru, dreapta) şi *
- muta(1, dreapta, centru, stanga) şi
- muta(1, stanga, dreapta, centru) şi
- muta(1, centru, stanga, dreapta) şi
- muta(1, centru, dreapta, stanga) şi **
- muta(1, stanga, dreapta, centru).

Exerciţii

1. Definiţi un predicat Prolog care să calculeze suma primelor n numere naturale nenule.
2. Scrieţi un predicat Prolog care să calculeze pentru un n dat valoarea elementului x_n din şirul:

$$x_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

3. Următorul predicat calculează valorile funcţiei lui Ackerman:

$$ac: N \times N \to N,\ ac(m, n) = \begin{cases} n + 1 \text{ , dacă } m = 0; \\ ac(m-1, 1) \text{ , dacă } n = 0; \\ ac(m - 1, ac(m, n - 1)), \text{ altfel} \end{cases}$$

   Scrieţi un predicat care să calculeze valoarea acestei funcţii într-un anumit punct.

4. Scrieţi un predicat Prolog care să calculeze cel mai mare divizor comun dintre două numere.
5. Definiţi un predicat Prolog care să calculeze valoarea funcţiei lui Fibonacci într-un punct.(f(1) = f(2) = 1; f(n) = f(n - 2) + f(n - 1) pentru n >= 3).
6. Scrieţi un predicat Prolog care să calculeze n^m, unde n şi m sunt numere naturale.
7. Să se verifice dacă un număr k este divizor al unui număr n. Construiţi apoi un predicat Prolog care să afişeze toţi divizorii unui număr natural n.

Notă: predicatul predefinit *mod* ne dă restul împărţirii primului operand la cel de-al doilea.