Operații aritmetice

1 Obiective

Lucrarea de față îl familiarizează pe student cu principalele recurențe matematice și cu tipurile de recursivitate.

2 Considerații teoretice

2.1 Operatorul „is”

În Prolog expresiile matematice sunt evaluate **doar** folosind operatorul „is”.

Exemple:

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


?- X is 1+2.
X = 3.
```

Expresia matematică se scrie în dreapta operatului „is” și nu trebuie să conțină variabile neinițializate.

Exemple:

```
?- X is (1+2*3)/7-1.
X = 0.

?- X is sqrt(4).
X = 2.0.

?- Y = 10, X is Y mod 2.
X = 0.

?- X is Y+1.
Arguments are not sufficiently instantiated

?- X is 1+2, X is 3+4.
false.
```

2.2 Cel mai mare divizor comun (CMMDC)

Pentru a calcula cel mai mare divizor comun vom folosi algoritmul lui Euclid. Pseudocod varianta 1 (pentru numere pozitive):

cmmdc(a,a) = a
cmmdc(a,b) = cmmdc(a-b, b), dacă a>b
cmmdc(a,b) = cmmdc(a, b-a), dacă b>a

Pseudocod varianta 2 (mai eficientă pentru numere mari, numere nenule):

cmmdc(a,0) = a
cmmdc(a,b) = cmmdc(b, a mod b)

Deoarece în Prolog, predicatele returnează doar adevărat/fals, trebuie să adăugam rezultatul în lista de parametrii a predicatului.

```
% Varianta 1
cmmdc1(X,X,X). % parametrul 3 este rezultatul la cmmdc
cmmdc1(X,Y,Z) :- X>Y, Diff is X-Y, cmmdc1(Diff,Y,Z).
cmmdc1(X,Y,Z) :- X<Y, Diff is Y-X, cmmdc1(X,Diff,Z).

% Varianta 2
cmmdc2(X,0,X). % parametrul 3 este rezultatul la cmmdc
cmmdc2(X,Y,Z) :- Rest is X mod Y, cmmdc2(Y,Rest,Z).
```

Exemplu de urmărire a execuției (vezi Laboratorul 1 pentru **trace**):

```
[trace] 3 ?- cmmdc1(3,3,R).
   Call: (7) cmmdc1(3, 3, _G1614) ?     % se unifică cu prima clauză
   Exit: (7) cmmdc1(3, 3, 3) ?          % iese cu succes
R = 3 ;                                 % repetăm întrebarea
   Redo: (7) cmmdc1(3, 3, _G1614) ?     % se unifică cu clauza 2
   Call: (8) 3>3 ?                      % primul apel din clauza 2
   Fail: (8) 3>3 ?                      % eșuează și trece la următoarea clauză
   Redo: (7) cmmdc1(3, 3, _G1614) ?     % se unifică cu clauza 3
   Call: (8) 3<3 ?                      % primul apel din clauza 3
   Fail: (8) 3<3 ?                      % eșuează
   Fail: (7) cmmdc1(3, 3, _G1614) ?         % eșuează (nu mai există altă
                                            %clauză cu care să se unifice)
false.
```

```
[trace] 6 ?- cmmdc1(3,5,R).
   Call: (7) cmmdc1(3, 5, _G1614) ?      % se unifică cu clauza 2
   Call: (8) 3>5 ?                       % primul apel din clauza 2
   Fail: (8) 3>5 ?                       % eșuează și trece la următoarea clauză
   Redo: (7) cmmdc1(3, 5, _G1614) ?      % se unifică cu clauza 3
   Call: (8) 3<5 ?                       % primul apel din clauza 3
   Exit: (8) 3<5 ?                       % succes
   Call: (8) _G1693 is 5-3 ?             % al doilea apel din clauza 3
   Exit: (8) 2 is 5-3 ?                  % succes
   Call: (8) cmmdc1(3, 2, _G1614) ?      % apelul recursiv din clauza 3
   Call: (9) 3>2 ?                       % primul apel din clauza 2
   Exit: (9) 3>2 ?                       % succes
   Call: (9) _G1696 is 3-2 ?             % al doilea apel din clauza 2
   Exit: (9) 1 is 3-2 ?                  % succes
   Call: (9) cmmdc1(1, 2, _G1614) ?      % apelul recursiv din clauza 3
   Call: (10) 1>2 ?                      % etc.
   Fail: (10) 1>2 ?
   Redo: (9) cmmdc1(1, 2, _G1614) ?
   Call: (10) 1<2 ?
   Exit: (10) 1<2 ?
   Call: (10) _G1699 is 2-1 ?
   Exit: (10) 1 is 2-1 ?
   Call: (10) cmmdc1(1, 1, _G1614) ?
   Exit: (10) cmmdc1(1, 1, 1) ?
   Exit: (9) cmmdc1(1, 2, 1) ?
   Exit: (8) cmmdc1(3, 2, 1) ?
   Exit: (7) cmmdc1(3, 5, 1) ?
R = 1
```

Urmărește execuția la:

```
?- cmmdc1(30,24,X).
?- cmmdc1(15,2,X).
?- cmmdc1(4,1,X).
```

Este important să reținem următoarele:

- Predicatele în Prolog au return de tip boolean (*true/false*), acest return este folosit de către interpretorul Prolog pentru a procesa secvențial comenzile. Gândiți-vă la un *și logic* – acesta este operatorul "**,**" în Prolog – ca un predicat să returneze *true* înseamnă ca *toate* apelurile subpredicatelor trebuie sa fie adevărate. Dacă cel putin unul returnează false, rezultatul este false.

- Nu întotdeauna vrem Prolog sa returneze true sau false, există cazuri in care dorim să calculeze o valoare. În acest caz, ne gândim la C pentru a face o analogie. C permite un singur return, când vrem să avem acces la mai multe -> putem să folosim parametrii unei funcții prin operatorul de adresă (&). Într-un mod similar, output-ul din predicatele Prolog este unul dintre argumente. În cazul prezentat mai sus, al predicatului cmmdc/1, este al treilea argument.
    - În consecință, nu putem să folosim niciuna dintre următoarele pentru a primi output-ul unui predicat Prolog:
        - cmmdc1(...) = R
        - R = cmmdc1(...)
        - cmmdc1(...) is R
        - R is cmmdc1(...)
    - Modul corect este de a ne folosi de parametrii *cmmdc1(X,Y,R)*, unde X si Y sunt variabile instanțiate iar R este o variabilă neinstanțiată care ajunge instanțiată la finalul apelului.

2.3 Factorial

Recurența matematică a factorialului este:

```
factorial(0) = 1
factorial(n) = n * factorial(n-1), pentru n>0
```

2.3.1 Recursivitatea înapoi (backwards)

În cadrul recursivității înapoi, rezultatul este construit la întoarcerea din recursivitate. Ceea ce înseamnă că inițializarea rezultatului se face după apelul recursiv.

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

Urmărește execuția la:

```
?- fact_bwd(6, 720).
?- N=6, fact_bwd(N, 120).
?- fact_bwd(6, F).
?- fact_bwd(N,720).
?- fact_bwd(N,F).
```

Dacă se repetă întrebarea (prin **;** în *SWI* sau **Next** în *SWISH*), va rezulta într-o buclă infinită – deoarece condiția de oprire va fi ignorată (prin backtracking). Pentru a primi un singur rezultat, va trebui să ne asigurăm că N nu ajunge să fie

negativ în a doua clauză (pentru că prima clauză va fi ignorată la repetarea întrebării). Clauza a doua din predicat ar trebui să fie:

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

2.3.2 Recursivitate înainte (forward)

În cadrul recursivității înainte, rezultatul este construit într-un **acumulator**. Un acumulator este un *argument adițional* (în acest caz, al doilea) necesar pentru recursivitatea înainte. Valoarea acumulatorului trebui asignată rezultatului în ultimul pas de recursivitate (clauza de terminare). Valoarea noului acumulator se va calcula înainte de apelul recursiv.

```
fact_fwd(0,Acc,F) :- F = Acc.
fact_fwd(N,Acc,F) :- N > 0, N1 is N-1, Acc1 is Acc*N, fact_fwd(N1,Acc1,F).
```

Acumulatorul trebuie instanțiat la fiecare întrebare cu aceeași valoare. Fiind un factorial, vom folosi elementul neutru al înmulțirii (**1**). Având de scris un argument adițional la fiecare întrebare poate fi deranjant, pentru a ușura procesul se poate implementa un *wrapper*.

```
fact_fwd(N,F) :- fact_fwd(N,1,F). % acumulatorul este inițializat cu 1
```

Observație. Semnătura unui predicat Prolog este determinată de nume si de numărul de argumente, astfel *fact_fwd/2* și *fact_fwd/3* pot fi distinse ca fiind predicate diferite de către interpretorul Prolog.

2.3.3 Recursivitate înapoi (backward) vs înainte (forward)

Vom face o urmărire rudimentară a celor două tipuri de recursivitate, folosind întrebarea unui factorial de 3:

- Înapoi / Backwards:

```
?- fact_bwd(3, F).

fact_bwd(3, Fa).
        N1(2) is N(3) - 1.
        fact_bwd(2, Fb).
                N1(1) is N(2) - 1.
                fact_bwd(1, Fc).
                        N1(0) is N(1) - 1.
                        fact_bwd(0, Fd).
```

```
% în acest punct, predicatul se poate unifica cu condiția de oprire (clauza 1);
% REMARCAȚI faptul că se folosește doar de 0 pentru a se opri
% În acest caz doar primul argument pentru a ajunge la condiția de oprire
% și am putea sa îl denominăm ca argumentul de oprire
% Pe când, al doilea argument (1) este defapt folosit pentru a instanţia acest argument

% REMARCAȚI faptul că nicio operație nu a fost făcută până la acest punct
% pentru a calcula factorialul, operațiile sunt făcute la întoarcerea
% apelurilor recursive, de unde si numele: recursivitate înapoi

                        fact_bwd(0, 1).
                        F is F1(1) * N(1)
                fact_bwd(1, 1) % al doilea argument este F-ul anterior
                F is F1(1) * N(2)
        fact_bwd(2, 2) % al doilea argument este F-ul anterior
        F is F1(2) * N(3)
fact_bwd(3, 6).
```

- Înainte / Forwards:

După cum a fost menționat anterior, acumulatorul (al doilea argument) trebuie instanțiat la fiecare întrebare:

```
?- fact_fwd(3, 1, F).

fact_fwd(3, 1, F).
        N1 is N(3) – 1
        Acc1 is Acc(1) * N

        fact(2, 3, F)
                N1 is N(2) – 1
                Acc1 is Acc(3) * N

                fact(1, 6, F).
                        N1 is N(1) – 1
                        Acc1 is Acc(6) * N

                        fact_fwd(0, 6, F).

% în acest punct, predicatul se unifică cu condiția de oprire
% (prima clauză), prin primul argument.

% Al doilea argument fiind o variabilă instanțiată și
% al treilea o varibilă neinstanțiată – care a fost transferată până la
```

```
% acest punct dintr-un apel recursiv în altul – pot fi unificate iar
% ambele vor conține aceeași valoare, în acest caz 6

                        fact_fwd(0, 6, 6).
                fact_fwd(1, 6, 6).
        fact_fwd(2, 3, 6).
fact_fwd(3, 1, 6).

% După cum puteți observa la finalul apelurilor recursive, nicio operație nu este facută,
% toate operațiile au fost făcute înaintea apelurilor recursive, de unde si numele:
% recursivitate înainte
```

Este important sa observăm următoarele:

- În primul rand, urmăriți ce se întâmplă cu acumulatorul la revenirea din fiecare apel recursiv: revine în starea din acel apel. Acest fapt este primul motiv pentru care trebuie să folosim output-ul (al treilea argument) pentru a salva valoarea finală
- Al doilea motiv este faptul că fără să avem o variabilă neinstanțiată (al treilea argument) nu putem sa returnăm o valoare – ar returna doar true sau false. Încercați acest lucru, stergeți ultimul argument din acest predicat și încercați următoarea întrebare:
  ?- fact(3, 1).
- Al treilea motiv care justifică folosirea unui acumulator este faptul că nu am putea face operațiile matematice fără. Încercați acest lucru, stergeți al doilea argument și înlocuiți cu al treilea în operații. Ce credeți că se va întâmpla? Ați înmulți un număr (**N**) cu o valoarea neinstanțiată (**F**) care va rezulta intr-o eroare.

Observație. În cadrul recursivității înapoi urmăriți output-ul (ultimul argument în ambele variante), acesta se schimbă la fiecare apel recursiv, pe când în varianta de recursivitate înainte se păstrează; acest fapt este o trăsătură inerentă a cum diferă cele două procese de recursivitate.

2.4 Bucla FOR

Chiar dacă structurile de control repetitive nu sunt specifice programării Prolog, acestea pot fi implementate ușor. Ne vom uita peste un exemplu al buclei *for*:

```
for(Inter,Inter,0).
for(Inter,Out,In):-
  I>0,
```

```
NewIn is In-1,
<modificați_ Inter_pentru_a_obține_Intermediate>
for(Intermediate,Out,NewIn).
```

Observație1. Remarcați ce se întâmplă în condiția de oprire (clauza 1 din predicatul *for/3*). Ce fel de recursivitate este? Dacă v-ați gândit la recursivitate înainte, ați avut dreptate. Primul argument functionează ca și acumulator.

Observație2. Remarcați că pentru acest predicat, output-ul este dat de al doilea argument (nu ultimul). În general, argumentul dorit este ultimul datorită convenienței și lizibilității, nu din obligație.

Observație3. Linia bold-uită din clauza a doua (I>0), care este scopul acesteia? Cea mai bună variantă de a afla este de a urmări execuția predicatului *cu* si *fără* această linie. Retineți că: operațiile ca suma, au un singur răspuns matematic.

3 Exerciții

1. Scrieți predicatul *cmmmc/3* care calculează cel mai mic multiplu comun (CMMMC).

   Sugestie: cmmmc între două numere naturale este raportul dintre produsul lor și cmmdc.

% *cmmc(X, Y, Z).* - calculează cel mai mic multiplu comun dintre X și Y și returnează rezultatul în Z
?- cmmmc(12, 24, Z).
Z=24;
false.

2. Scrieți predicatul care calculează ridicarea unui număr la o putere aleasă folosind:
   - recursivitate înainte: *power_fwd/3*
   - recursivitatea înapoi: *power_bwd/3*

% *power(X, Y, Z).* – va calcula X la puterea Y și va returna rezultatul în Z
?- power_fwd(2, 3, Z).
Z=8;
false.

?- power_bwd(2, 3, Z).
Z=8;
false.

3. Scrieți predicatul *fib/2* care calculează al n-lea număr din șirul lui Fibonacci.

   Formula de recurență este:

   fib(0) = 0
   fib(1) = 1
   fib(n) = fib(n-1) + fib(n-2), pentru n>0

% *fib(N, X).* – va calcula al N-lea număr din șirul lui Fibonacci și va returna rezultatul în X
?- fib(5, X).
X=5;
false.

?- fib(8, X).
X=21;
false.

4. Scrieți predicatul anterior folosind doar un singur apel recursiv.

% *fib1(N, X).* – va calcula al N-lea număr din șirul lui Fibonacci și va returna rezultatul în X
?- fib1(5, X).
X=5;
false.

?- fib1(8, X).
X=21;
false.

5. Scrieți predicatul *triangle/3* care verifică dacă cei trei parametrii pot reprezenta lungimile laturilor unui triunghi. Suma oricăror două laturi trebuie să fie mai mare decât a treia latură.

% *triangle(A, B, C).* – va verifica dacă A,B,C ar putea fi laturile unui triunghi și va returna
% true sau false
?- triangle(3, 4, 5).
true.

?- triangle(3, 4, 8).
false.

6. Scrieți predicatul *solve/4* care să rezolve ecuația de gradul doi ($a*x^2+b*x+c = 0$). Predicatul trebuie să aibă 3 parametri de intrare (*A,B,C*) și un parametru de ieșire (*X*). La repetarea întrebării trebuie să se obțină a doua soluție (distinctă) dacă există.

% *solve(A, B, C, X).* – va rezolva ecuația pătratică $A*x^2+B*x+C = 0$
% și va returna rezultatul(ele) în X sau false altfel
?- solve(1, 3, 2, X).
X=-1;
X=-2;
false.

?- solve(1, 2, 1, X).
X=-1;
false.

?- solve(1, 2, 3, X).
false.

7. **For:** Scrieți predicatul *for/3* după specificațiile date în secțiunea 2.4 care calculează suma tuturor numerelor mai mici decât un număr dat.

```
?- for(0, S, 9).
S=45;
false.
```

8. **For:** Scrieți predicatul *for_bwd/2* folosind recursivitate înapoi care calculează suma tuturor numerelor mai mici decât un număr dat.

```
% for_bwd(In,Out). – va calcula suma toturor valorilor între 0 și In
% și va returna rezultatul în Out
?- for_bwd(9, S).
S=45;
false.
```

9. **While:** Scrieți predicatul *while/3* care simulează o buclă *while* și returnează suma tutoror numerelor între două numere date.

   Sugestie. Structura unei astfel de bucle este:

   ```
   while <some condition>
         <do something>
   end while
   ```

```
% while(Low,High,Sum). – va calcula suma toturor valorilor între Low și High
% și va returna rezultatul în Sum
?- while(0, 5, S).
S=10;
false.

?- while(2, 2, S).
S=0;
false.
```

10. **Repeat….until:** Scrieți predicatul *dowhile/3* care simulează o buclă *repeat…until* și returnează suma tutoror numerelor între două numere date.

    Sugestie. Structura unei astfel de bucle este:

    ```
    repeat
          <do something>
    until <some condition>
    ```

% *dowhile(Low,High,Sum).* – va calcula suma toturor valorilor între Low și High
% și va returna rezultatul în Sum
?- *dowhile*(0, 5, S).
S=10;
false.

?- *dowhile*(2, 2, S).
S=2;
false.