2. Module şi predicate predefinite pentru prelucrarea obiectelor SWI-Prolog.

2.1 Obiecte şi structuri de date în SWI-Prolog

În figura 2.1 prezentăm o clasificare a obiectelor din mediul de dezvoltare SWI-Prolog.

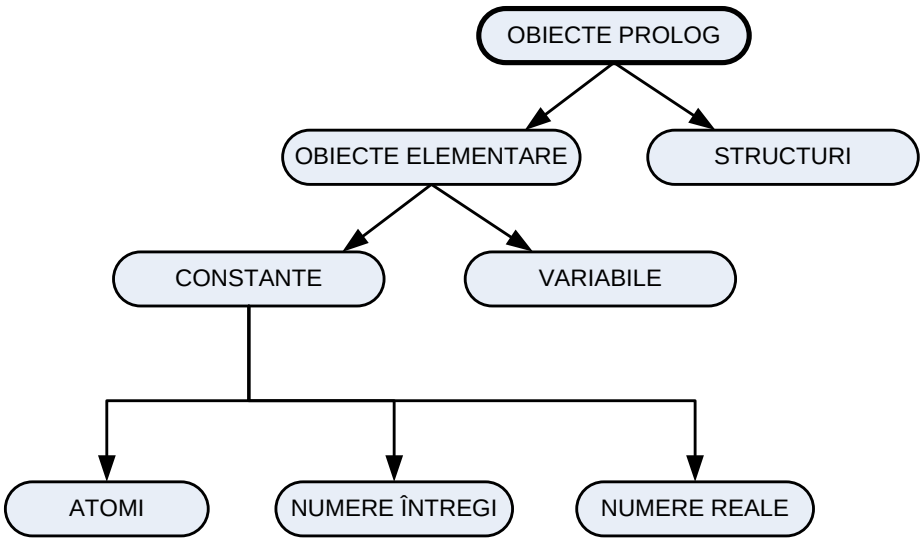

Figura 2.1 *Taxonomia obiectelor SWI-Prolog*

Sistemul Prolog recunoaşte tipul obiectelor din program după forma sintactică, deoarece sintaxa Prolog impune diferite forme pentru fiecare tip de obiect. În SWI-Prolog există o mulțime de *predicate predefinite*, unele fiind *predicate standard*, iar altele depinzând de implementare. De fapt, un program Prolog este alcătuit din *predicate*. Fiecare predicat este definit de *numele* său şi de *aritatea* (număr fix de argumente ale predicatului) sa. În Prolog, două predicate cu acelaşi nume, dar număr diferit de argumente, se consideră că sunt predicate diferite. Argumentele predicatelor Prolog, prin analogie cu logica predicatelor de ordinul I, se numesc *termeni*, şi pot fi constante, variabile sau structuri. În capitolul 1 al acestei cărți, am arătat cum se definesc *constantele* (încep întotdeauna cu literă mică) şi *variabilele* (încep întotdeauna cu literă mare).

2.1.1 Constante

O constantă (atom) este un identificator ce poate fi construită în trei feluri:

- Poate începe cu literă mică şi se continuă cu orice înşiruire de litere mici, litere mari, cifre sau caracterul "_" (underscore): *ana, nil, x25, x_25, x_25AB, x_, x__y, procedura_alfa_beta, dna_Maria, domnul_Goe*, etc.
- Poate începe cu şiruri de caractere speciale: <>, <...>, =, ==, ==>, ..., etc.
- Există, totuşi, şi posibilitatea ca atomul să înceapă cu literă mare sau să conțină caractere speciale. Limbajul Prolog permite acest lucru prin încadrarea atomului între caractere apostrof (ex. *tata('Alexe Ion', daniel), 'Bogdan', 'Romania',* etc.).

În SWI-Prolog, *numerele* pot fi reprezentate ca: întregi, reale. Sintaxa pentru *numere întregi* este cât se poate de simplă: 1, 1313, 0, -97, +32. Numerele reale, în general (depind de tipul de Prolog utilizat), sunt reprezentate într-o formă similară celor din C, C++: *3.14, 0.56, -0.0035, 100.2, 25.78*. Numerele reale nu sunt foarte utilizate în Prolog. Motivul este că limbajul Prolog este orientat pe simboluri, pe calcul non-numeric (numerele întregi, în schimb, sunt utilizate frecvent pentru a contoriza numărul de elemente dintr-o listă, etc.).

2.1.2 Variabile

Din punct de vedere sintactic, variabilele sunt tot atomi, însă au o altă semnificație față de constante. Variabilele sunt şiruri de caractere, cifre sau caracterul underscore şi încep cu literă mare sau caracterul underscore: X, Rezultat, Obiect1, Lista_participanti, _x24, _25.

În Prolog există situații în care o variabilă apare o singură dată într-o regulă, caz în care nu avem nevoie de un nume pentru ea, deoarece nu este referită decât într-un singur loc. De exemplu, în cazul în care dorim să scriem o regulă care ne spune dacă cineva este fiul cuiva, o posibilă implementare ar fi:

este_fiu(Fiu):- parinte(Y, Fiu).

În relația (regula) de mai sus am denumit cu Y un părinte anonim. În acest caz, nu interesează cine apare pe post de părinte. Pentru a nu încărca regulile cu nume (dacă situația permite acest lucru) care pot distrage atenția

şi îngreuna citirea programelor, se pot utiliza variabile ***anonime***, notate cu "_" (underscore). Prin urmare, regula anterioară se poate rescrie astfel:

este_fiu(Fiu):- parinte(_, Fiu).

De cele mai multe ori folosim o variabilă anonimă când această variabilă apare în program o singură dată şi astfel simplificăm procesul de unificare şi câştigăm timp.

Variabilele din Prolog *nu sunt identice* ca semnificație cu variabilele C, C++, fiind mai curând similare cu variabilele în sens matematic. **O variabilă, odată ce a primit o valoare, nu mai poate fi modificată.** Acest lucru elimină efectele laterale care sunt permise în limbajele procedurale.

De reținut că, în SWI-Prolog, o variabilă neinstanțiată semnifică ceva necunoscut.

2.1.3 Structuri

Structurile sunt folosite pentru a grupa sub acelaşi nume, mai multe date de acelaşi tip sau de tipuri diferite. Obiectele structurate (numite pe scurt *structuri***)**, sunt obiecte formate din mai multe componente. *Componentele*, pot fi, la rândul lor, *structuri*. De exemplu, *data calendaristică* poate fi văzută ca o structură cu trei componente: zi, luna, an. Deşi formate din câteva componente, structurile sunt tratate în Prolog ca obiecte singulare. Pentru a mixa componentele într-un singur obiect avem nevoie de un ***functor*** *(elementul care uneşte componentele)*.

Pentru exemplul din figura 2.2, functorul potrivit este: ***data(1,martie, 2009).***

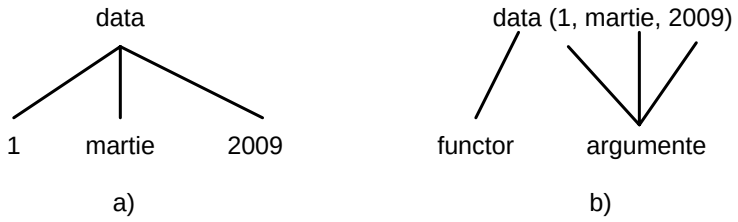

Figura 2.2 *Exemplu de structură de date: a) reprezentare sub formă de arbore, b) reprezentare cod Prolog*

Toate componentele din acest exemplu sunt *constante* (doi întregi şi un atom), iar elementul care le unește (liantul) este functorul *data*. Precizăm faptul că functorul poate fi orice atom, iar componentele pot fi orice obiect Prolog.

Aşadar, în Prolog, o structură se defineşte prin specificarea:

- numelui structurii (*functorul structurii*);
- elementelor structurii (*componentele structurii*).

Structurile Prolog pot fi utilizate pentru reprezentarea structurilor de date, de exemplu liste sau arbori. Toate obiectele structurate pot fi desenate ca nişte arbori (figura 2.2). Rădăcina arborelui este functorul, iar fii săi sunt componentele. Dacă o componentă este, de asemenea, o structură, atunci ea este un subarbore al unui arbore care corespunde întregului obiect structurat.

2.2 Operatori

Limbajul Prolog pune la dispoziția programatorului trei categorii de operatori:

- operatori aritmetici,
- operatori relaționali,
- operatori definiți de utilizator.

Operatorii aritmetici sunt utilizați pentru operații matematice de bază, cum ar fi:

+	adunare,
-	scădere,
*	înmulțire,
/	împărțire,
mod	modulo

La întrebarea:

?- X = 1 + 2.

Prolog va răspunde cu:

X = 1 + 2

şi nu X = 3, aşa cum ne-am aştepta (ținând cont de limbajele clasice procedurale C, Pascal). Motivul pentru care Prolog răspunde astfel este că în expresia 1 + 2, termenul + este *functorul,* iar 1 şi 2 sunt *argumentele* acestuia. Deci nu există nimic în acest scop care să activeze operația de adunare. Operatorul special predefinit pentru operația de adunare este ***is***. Operatorul predefinit *infixat is* va forța evaluarea expresiei. Prin urmare, modul corect de a cere răspunsul la operația aritmetică de adunare este:

?- X is 1 + 2.

Acum Prolog va răspunde cu (variabila X este instanțiată cu 3):

X = 3

Operatorii relaționali sunt predicate predefinite infixate. Aceştia sunt:

= *operatorul de egalitate*. Predicatul (operatorul) de egalitate funcționează ca şi cum ar fi definit prin următorul fapt: *X=X.*,
iar încercarea de a satisface un scop de tipul **X = Y** se face prin încercarea de a unifica **X** cu **Y**. Din aceasta cauză, dându-se un scop de tipul **X = Y**, regulile de decizie care indică dacă scopul se îndeplineşte sau nu sunt următoarele:

- Dacă **X** este variabilă neinstanțiată, iar **Y** este instanțiată la orice obiect Prolog, atunci scopul reuşeşte. Ca efect lateral, **X** se va instanția la aceeaşi valoare cu cea a lui **Y**. De exemplu:

  ?- *merg(studenti, universitate) = X.*

  este un scop care reuşeşte şi X se instanțiază la merg(studenti, universitate).
- Dacă atât **X** cât şi **Y** sunt variabile neinstanțiate, scopul **X = Y** reuşeşte, variabila **X** este legată la **Y** şi reciproc. Aceasta înseamnă că ori de câte ori una dintre cele două variabile se instanțiază la o anumită valoare, cealaltă variabilă se va instanția la aceeaşi valoare.
- Atomii şi numerele sunt întotdeauna egali cu ei înşişi. De exemplu, următoarele scopuri au rezultatul marcat drept comentariu:

  ana = ana *% este satisfăcut*
  dulce = amar *% eşuează*
  10 = 10 *% reuşeşte*
  1100 = 1011 *% eşuează*
- Două structuri sunt egale dacă au acelaşi functor, acelaşi număr de componente şi fiecare componentă dintr-o structură este egală cu componenta corespunzătoare din cealaltă structură. De exemplu, scopul:

  data(1, 3, 2009, timp(12, 20, 33)) = data(1, 3, X, timp(12, Y, 33)).

  este satisfăcut, iar ca efect lateral se fac instanțierile:

  X = 2009 *şi* ***Y = 20****.*

Scopul:

autor(emil, X) = autor(Y, cioran).

este satisfăcut, iar ca efect lateral se fac instanțierile:

X = cioran *şi* ***Y = emil****.*

În urma acestui proces, ambii termeni arată astfel:

autor(emil, cioran).

Aşadar, unificarea (modul în care Prolog realizează potrivirile între termeni) presupune atât modificarea obiectului țintă, cât şi a obiectului sursă.

= = *testează echivalența a două variabile*. El consideră cele două variabile egale doar dacă ele sunt deja partajate. **X = = Y** reuşeşte ori de câte ori **X = Y** reuşeşte, dar reciproca este falsă:

```
?- X = = X.
yes          %variabilă neinstanțiată

?- X= =Y.
no
```

\= *operatorul de inegalitate* reprezintă un predicat opus celui de egalitate. Scopul **X \= Y** reuşeşte dacă scopul **X = Y** nu este satisfăcut şi eşuează dacă **X = Y** reuşeşte.

X > Y X este mai mare decât Y

X < Y X este mai mic decât Y

X >= Y X este mai mare sau egal cu Y

X =< Y X este mai mic sau egal cu Y

X =\= Y returnează adevărat dacă X şi Y au valori diferite (face evaluare aritmetică dar nu instanțiază nimic).

X =:= Y returnează adevărat dacă X şi Y au aceleaşi valori (face evaluare aritmetică dar nu instanțiază nimic).

Exemplu:

```
?- 1 + 2 =:= 2 + 1.  % se face evaluarea aritmetică a expresiei
yes
?- 1 + 2 = 2 + 1.    % se compară obiectele
no
```

Notă:

Operatorii definiți de utilizator vor fi prezentați în detaliu în capitolul 5 al acestei lucrări.

2.3 Predicate pentru prelucrarea obiectelor SWI-Prolog

- **findall(X, Scop, Lista)** – are ca efect obținerea tuturor termenilor **X** care satisfac predicatul **Scop** în baza de cunoștințe a programului și cumularea acestor termeni în lista **Lista**. **Scop** trebuie să fie instanțiat la un predicat Prolog în care, în mod normal, trebuie să apară **X**. Dacă **Scop** nu reușește, **findall** reușește, instanțiind **Lista** la lista vidă (*această ultimă caracteristică este specifică mediului SWI-Prolog*).
- **bagof(X, Scop, Lista)** – are ca efect colectarea în **Lista** a tuturor termenilor **X** care satisfac **Scop**, ținând cont de diversele instanțieri posibil diferite ale celorlalte variabile din **Scop**. **Scop** este un argument care trebuie să fie instanțiat la un scop Prolog. Dacă **Scop** nu conține alte variabile în afara lui **X**, atunci efectul predicatului **bagof** este identic cu cel al lui **findall**. Dacă **Scop** nu reușește cel puțin o dată atunci **bagof** eșuează.
- **setof(X, Scop, Lista)** – are același efect cu predicatul **bagof**, cu excepția faptului că valorile cumulate în lista **Lista** sunt sortate crescător și se elimină aparițiile multiple de valori, deci **Lista** devine o mulțime ordonată. Dacă **Scop** nu reușește cel puțin o dată, atunci **setof** eșuează.

Exemplu de utilizare a celor 3 predicate:

```
parinte(mihai, carmen).
parinte(andreea, carmen).
parinte(ion, elena).
parinte(cristina, elena).
```

```
26 ?- findall(X, parinte(X, C), L).
L = [mihai, andreea, ion, cristina].
```

```
27 ?- bagof(X, parinte(X, C), L).
C = carmen,
L = [mihai, andreea] ;
C = elena,
L = [ion, cristina].
```

```
28 ?- setof(X, parinte(X, C), L).
C = carmen,
L = [andreea, mihai] ;
C = elena,
L = [cristina, ion].
```

2.4 Recursivitate în SWI-Prolog

Programarea în Prolog depinde foarte mult de această tehnică numită *recursivitate*. În principiu, recursivitatea implică definirea unui predicat în funcție 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. Recursivitatea de la nivelul algoritmilor este echivalentă cu *demonstrarea prin inducție* din matematică.

O definiție recursivă (în orice limbaj procedural sau declarativ) trebuie să aibă întotdeauna cel puțin două părți:

- *condiție 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 pe ea însăşi.

La fiecare nivel, condiția elementară este verificată. Dacă s-a ajuns la ea, recursivitatea se încheie, altfel, recursivitatea continuă.

O definiție recursivă cuprinde o serie de noțiuni matematice, cum ar fi factorialul unui număr, permutări, aranjamente, combinări, mulțimea submulțimilor unei mulțimi, numerele lui Fibonacci, turnurile din Hanoi, etc.

Controlul recursivității prin predicatele CUT şi FAIL.

Predicatul **cut**, notat cu atomul special **!**, este un predicat standard, fără argumente, care se îndeplineşte (este adevărat) întotdeauna şi nu poate fi resatisfăcut. Acest predicat are următoarele *efecte laterale*:

- La întâlnirea predicatului **cut** toate selecțiile făcute între scopul antet de regulă şi **cut** sunt "înghețate", deci marcajele de satisfacere a scopurilor sunt eliminate, ceea ce duce la eliminarea oricăror altor soluții alternative pe această porțiune. O încercare de resatisfacere a unui scop între scopul antet de regulă şi scopul curent va eşua.
- Dacă clauza în care s-a introdus predicatul **cut** reuşeşte, toate clauzele cu acelaşi antet cu aceasta, care urmează clauzei în care a apărut **cut** vor fi ignorate. Ele nu se mai folosesc în încercarea de resatisfacere a scopului din antetul clauzei care conține **cut**.

Pe scurt, comportarea predicatului **cut** este următoarea:

(C1) **B :- C, D, !, E, F.**

(C2) **B :- G, H.**

(C3) **B :- K, L.**

Presupune că scopul curent este **B**. Dacă **C** şi **D** sunt satisfăcute, ele nu mai pot fi resatisfăcute datorită lui **cut**. Dacă **C** şi **D** sunt satisfăcute, C_2 şi C_3 nu vor mai fi utilizate pentru resatisfacerea lui **B**. Resatisfacerea lui **B** se poate face numai prin resatisfacerea unuia din scopurile **E** şi **F**, dacă acestea au mai multe soluții.

Mai exact, în momentul execuției predicatului **cut**, calculatorul "uită" alternativele descoperite înaintea sau după apelul acestei clauze.

```
afiseaza(1):-!, write('unu').
afiseaza(2):-!, write('doi').
afiseaza(3):-!, write('trei').
afiseaza(_):- write('altceva!').
```

```
 35 ?- afiseaza(2).
 doi
true.
```

În exemplul de mai sus, atâta vreme cât *afiseaza* este determinist, nu contează dacă predicatul **cut** este scris înainte sau după predicatul **write**. Oricum, programele sunt mai uşor de interpretat dacă **cut** apare cât mai devreme posibil.

Aşadar, predicatul **cut** poate fi util în cazul în care se doreşte eliminarea unor paşi din deducție care nu conțin soluții sau eliminarea unor căi de căutare care nu conțin soluții. El permite exprimarea în Prolog a unor structuri de control de tipul:

dacă condiție **atunci** acțiune$_1$
altfel acțiune$_2$

astfel:

daca_atunci_altfel(Cond, Act1, Act2) **:-** Cond, **!**, Act1.

daca_atunci_altfel(Cond, Act1, Act2) **:-** Act2.

Se observă însă că există două contexte diferite în care se poate utiliza predicatul **cut**:

- într-un context predicatul **cut** se introduce numai pentru creşterea eficienței programului, caz în care el se numeşte "***cut verde" (green cut)***;
- în alt context utilizarea lui **cut** modifică semnificația procedurală a programului, caz în care el se numeşte „***cut roşu" (red cut)***.

În exemplul de mai sus avem un **cut roşu.** Utilizarea predicatului **cut** în definirea predicatului asociat structurii de control **daca_atunci_altfel** introduce un **cut roşu** deoarece efectul programului este total diferit dacă se schimbă ordinea clauzelor. Introducerea unui **cut roşu** modifică corespondența dintre semnificația declarativă şi semnificația procedurală a programelor Prolog.

În cele ce urmează prezentăm un exemplu de **cut verde**. Adăugarea lui **cut** nu face decât să crească eficiența programului, dar semnificația procedurală este aceeaşi, indiferent de ordinea în care se scriu cele cinci clauze.

```
afiseaza(1):-!, write('unu').
afiseaza(2):-!, write('doi').
afiseaza(3):-!, write('trei').
afiseaza(X):-X<1, !, write('mai mic decat unu').
afiseaza(X):-X>3, !, write('mai mare decat trei').
```

Un alt exemplu de utilizare a predicatului cut (în cele două variante: verde şi roşu) este pentru aflarea minimului dintre două numere:

```
min_verde(X, Y, X) :- X =< Y, !.              % cut verde
min_verde(X, Y, Y) :- X > Y.

min_rosu(X, Y, X) :- X =< Y, !.              % cut rosu
min_rosu(X, Y, Y).
```

În definiția predicatului **min_verde** se utilizează un **cut** verde. Acesta este plasat pentru creşterea eficienței programului, dar ordinea clauzelor de definire a lui **min_verde** poate fi schimbată fără nici un efect asupra rezultatului programului. În cazul predicatului **min_roşu** se utilizează un **cut roşu**, asemănător structurii **daca_atunci_altfel**. Dacă se schimbă ordinea clauzelor de definire a predicatului **min_roşu**:

min2(X, Y, Y).
min2(X, Y, X) **:-** X =< Y, !.

atunci rezultatul programului va fi evident incorect pentru valori **X < Y**.

Limbajul Prolog permite exprimarea directă a eşecului unui scop cu ajutorul predicatului **fail**.

Fail este un predicat:

- ✓ standard,
- ✓ fără argumente,
- ✓ care eşuează întotdeauna.

Datorită acestui lucru introducerea unui predicat **fail** într-o conjuncție de scopuri, de obicei la sfârşit, determină intrarea în procesul de backtracking.

De reținut că după **fail** nu se mai poate satisface nici un scop. Dacă **fail** se întâlneste după predicatul **cut**, atunci nu se mai face backtracking.

De exemplu, enunțul:

"Un individ este rău dacă nu este bun.",

se poate exprima astfel:

```
bun(andrei).
bun(ion).
bun(gheorghe).

rau(X) :- bun(X), !, fail.
rau(X).
```

Exemplu de execuție a programului:

```
3 ?- rau(andrei).
false.

4 ?- rau(ion).
false.

5 ?- rau(catalin).
true.
```

În exemplul de mai sus predicatul **fail** este folosit pentru a determina eşecul. Acesta este, de obicei, precedat de predicatul **cut**, deoarece procesul de backtracking pe scopurile care îl preced este inutil, scopul eşuând oricum datorită lui **fail**.

Observație:

Combinația **!, fail** poate fi utilizată cu rol de negație.

2.5 Probleme rezolvate

1. Problema unicornului (*Lewis Carroll – "Alice în țara minunilor"*). În cele ce urmează prezentăm premisele de la care se pleacă:
   - Leul minte luni, marți şi miercuri şi spune adevărul în toate celelalte zile.

- Unicornul minte joi, vineri şi sâmbătă şi spune adevărul în toate celelalte zile.
- Astăzi leul zice: "Ieri a fost una din zilele în care eu mint".
- Tot astăzi unicornul zice: "Ieri a fost una din zilele în care eu mint".

Întrebarea (concluzia) este: **Ce zi este astăzi?**

Soluție:

Codul sursă (SWI-Prolog) al programului este următorul:

```
% precizăm succesiunea zilelor săptămânii
urmeaza(luni, marti).
urmeaza(marti, miercuri).
urmeaza(miercuri, joi).
urmeaza(joi, vineri).
urmeaza(vineri, sambata).
urmeaza(sambata,duminica).
urmeaza(duminica, luni).

% Premisa 1
minte(leu, luni).
minte(leu, marti).
minte(leu, miercuri).

% Premisa 2
minte(unicorn, joi).
minte(unicorn, vineri).
minte(unicorn, sambata).

% Premisele 3 şi 4
zice(Animal, Astazi) :-
                 minte(Animal, Ieri),
                 urmeaza(Ieri, Astazi).
posibil(Animal, Astazi) :-
                 zice(Animal, Astazi),
                 not(minte(Animal, Astazi)).
posibil(Animal, Astazi) :-
                 minte(Animal, Astazi),
                 not(zice(Animal, Astazi)).
% Conchidem
azi(Astazi) :-
           posibil(leu, Astazi), posibil(unicorn, Astazi).
```

La interogarea:

```
?- azi(Astazi).
```

răspunsul sistemului va fi:

```
Astazi = joi;
false.
```

2. În cele ce urmează prezentăm un program Prolog ce rezolvă ecuația de gradul al doilea. Vom declara următoarele predicate:
   - delta – se calculează discriminantul ecuației de gradul 2.
   - ec_grad2 – primește ca argumente coeficienții ecuației și, în funcție de valoarea coeficienților și a discriminantului, realizează următoarele acțiuni:
     - ✓ Afișează soluțiile în cazul în care acestea sunt numere întregi sau reale.
     - ✓ Afișează mesajul "Ecuația nu este de gradul 2", în cazul în care coeficientul lui x^2 este 0.
     - ✓ Afișează mesajul "Ecuația nu are soluții reale", în cazul în care discriminantul este negativ.

Codul sursă (SWI-Prolog) al programului este următorul:

```
ec_grad2(A,B,C):-
                delta(A,B,C,D),
                solutie(A,B,C,D).
delta(A,B,C,D):-
                D is (B*B)-(4*A*C).
solutie(A,_,_,_):-
                A = 0,
                writeln('Ecuatia nu este de gradul 2').
solutie(_,_,_,D):-
                D < 0,
                writeln('Ecuatia nu are solutii reale').
solutie(A,B,_,D):-
                D >= 0,
                X1 is (-B-sqrt(D))/(2*A),
                X2 is (-B+sqrt(D))/(2*A),
                write('Solutiile  sunt:  X1='),  write(X1),
write(', X2= '),writeln(X2).
```

La interogarea:

```
?- ec_grad2(1,4,2).
```

răspunsul sistemului va fi:

```
Solutiile sunt: X1=-3.41421, X2= -0.585786
true.
```

3. În cele ce urmează prezentăm un program Prolog care calculează factorialul unui număr *n*. Știm că *n! = 1·2·3·…·n* și *0! = 1*. În acest exemplu identificăm *condiția elementară* ca fiind *factorial(0) = 1* (aceasta va fi condiția care va opri recursivitatea).

   Codul sursă (SWI-Prolog) al aplicației este prezentat în figura 2.7.

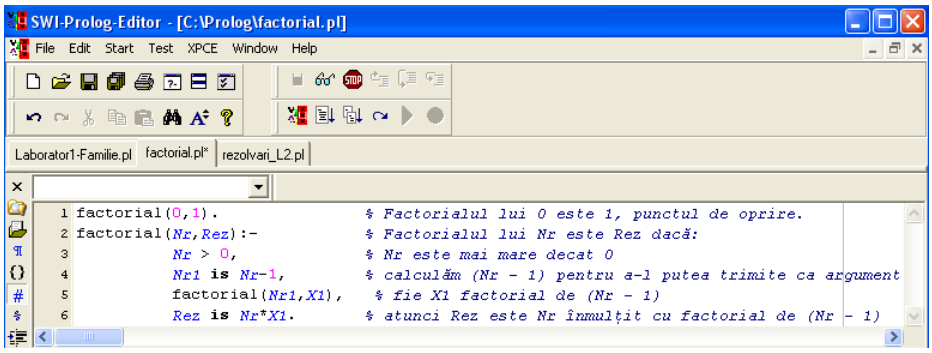

Figura 2.7 *Implementare Prolog pentru factorial (n)*

Spre exemplu, pentru a afla factorialul numărului 3, interogarea este următoarea:

```
?- factorial(3, Rez).
```

Răspunsul sistemului este:

```
Rez = 6 ;
```

4. Problema turnurilor din Hanoi. Se dau trei tije *(stânga, mijloc, dreapta)* şi *n discuri* de diferite dimensiuni, aranjate pe tija *stanga* în ordine descrescătoare a dimensiunilor lor.

   Cele *n* discuri de pe tija din *stanga* trebuie mutate pe tija din dreapta astfel încât acestea să fie ordonate ca la început. Mutările se fac cu următoarele restricții:

   - La fiecare mişcare se poate muta doar un disc.
   - Un disc cu diametrul mai mare nu poate fi mutat peste unul cu diametrul mai mic.
   - Tija din *mijloc* poate fi utilizată ca tijă intermediară.

   Codul sursă (SWI-Prolog) al aplicației este următorul:

```
solutie(N):-hanoi(N, stanga, dreapta, mijloc).
hanoi(0, _ , _ , _ ).
hanoi(N, A, B, C) :-
        N > 0, N1 is N - 1,
        hanoi(N1, A, C, B),
        write('Muta discul din '), write(A), write(' in
        '), write(B), nl,
        hanoi(N1, C, B, A).
```

La interogarea:

```
?- solutie(3).
```

răspunsul sistemului va fi:

```
Muta discul din stanga in dreapta
Muta discul din stanga in mijloc
Muta discul din dreapta in mijloc
Muta discul din stanga in dreapta
Muta discul din mijloc in stanga
Muta discul din mijloc in dreapta
Muta discul din stanga in dreapta
true
```

Rezultatul rezolvării problemei pentru N = 4, 5, 6 este prezentat în lucrarea [9].

2.6 Probleme propuse

1. Se vor studia problemele rezolvate (problemele prezentate pe parcursul acestui laborator), încercând găsirea altor posibilități de soluționare a acestora. Utilizați şi alte scopuri (interogări) pentru a testa definițiile predicatelor introduse. Se atrage atenția asupra faptului că toate cunoştințele din acest capitol vor fi necesare şi în derularea celorlalte capitole.
2. Se vor rezolva următoarele probleme propuse şi se va urmări execuția lor corectă.
   - Definiți relația *max(X, Y, Max)* astfel încât *Max* este mai mare decât cele două numere *X* şi *Y*.
   - Definiți relația *max(X, Y, Max)* astfel încât *Max* este cel mai mare număr din cele două *X* şi *Y* (Ex: max(23, 34, Max) rezulta Max=34).
   - Definiți relația *divide(A, B)* care testează dacă un număr este divizibil cu altul.
   - Definiți procedura *intre(N1, N2, X)* care, pentru două numere întregi date *N1* şi *N2*, generează prin backtracking toți întregii *X* care satisfac condiția: $N1 \leq X \leq N2$.
   - Să se scrie un program Prolog care calculează termenul *n* din şirul lui Fibonacci. Şirul lui Fibonacci este o secvență de numere în care fiecare număr se obține din suma precedentelor două din şir, conform relației:

$$F_n = \begin{cases} 0 & daca \ \ n = 0 \\ 1 & daca \ \ n = 1 \\ F_{n-1} + F_{n-1} & daca \ \ n > 1 \end{cases}$$

- Să se scrie un program Prolog care calculează cel mai mare divizor comun dintre două numere. **Indicație:** Se foloseşte definiția recursivă a lui Euclid:

  Dacă **a** şi **b** două numere întregi pozitive.
  dacă **b** = 0
  atunci **cmmdc(a, b) = a**;
  altfel **cmmdc(a, b) = cmmdc(b, r)**, unde **r** este restul împărțirii lui **a** la **b**.
- Să se scrie un program Prolog care testează dacă un număr introdus de la tastatură este prim.
- Să se scrie un program Prolog care să calculeze suma primelor *N* numere naturale nenule.