Rodica POTOLEA • Camelia LEMNARU
Eugen Richard ARDELEAN • Raluca Laura PORTASE

PROGRAMARE LOGICĂ

Îndrumător de laborator

UTPRESS
Cluj-Napoca, 2024
ISBN 978-606-737-707-1

Rodica POTOLEA
Camelia LEMNARU
Eugen Richard ARDELEAN
Raluca Laura PORTASE

Programare Logică

Îndrumător de laborator

UTPRESS
Cluj-Napoca, 2024
ISBN 978-606-737-707-1

Editura UTPRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Prof.dr.ing. Alin Suciu

Conf.dr.ing. Radu-Răzvan Slăvescu

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-707-1

CUPRINS

L1. Introducere în Prolog

1 Obiective

Lucrarea de față introduce principalele elemente din limbajul Prolog: fapte, reguli și întrebări, precum și tipurile de date utilizate în Prolog și regulile de unificare.

1.1 Utilizări ale Prolog-ului

Limbajul Prolog a fost creat în 1972 de către *Alain Colmerauer* și *Philippe Roussel*, fiind bazat pe interpretarea procedurală a clauzelor de tip Horn. Popularitatea limbajului a crescut în anii ’80 și ’90 în Europa, USA și Japonia, fiind utilizat pentru a implementa o serie de sisteme, cum ar fi: sisteme expert (pentru îngrijirea animalelor – *Auspig*, distribuția apei – *WADNES* și *SERPES*, consultant sportiv – *Perfect Pitch*), sisteme ce țin de mediul înconjurător (predicția vremii – *MM4 Weather Modeling System*), precum si o suita de aplicații pentru procesarea limbajului natural. Această creștere a fost datorată posibilității dezvoltării incrementale și capabilităților crescute de prototipizare în rezolvarea problemelor prin tehnici de Inteligență Artificială (IA). Alte arii de utilizare includ: demonstrarea teoremelor, planificare automată.

În zilele noastre, câteva din utilizările Prolog pot fi găsite în aplicații industriale, medicale și comerciale:

- Sisteme expert care rezolvă probleme fără ajutor uman (e.g. planificare automata, monitorizare, control și depanare sisteme complexe)
- Sisteme de suport a deciziilor (e.g. în diagnosticul medical)
- Sisteme de suport pentru clienți

Totodată, Prolog a fost utilizat într-un sistem capabil să răspundă la întrebări formulate în limbaj natural, dezvoltat de IBM – denumit Watson. În mod specific, a fost utilizat în implementarea de soluții bazate pe potrivire de șabloane pentru a construi diferiți arbori de parsare.

1.2 De ce ar trebui să înveți Prolog?

Învățarea Prolog-ului este extrem de benefică pentru profesioniști din mai multe motive. În primul rând, îi introduce în paradigma programării logice, oferindu-le o perspectivă unică asupra rezolvării problemelor, distinctă de abordările convenționale (procedurale). Acest lucru stimulează abordarea logică, în

defavoarea controlului explicit specific din abordarea imperativă, și deci o înțelegere mai profundă a logicii și a raționamentului deductiv, abilități esențiale aplicabile în diverse domenii. Suportul său pentru *programarea logică cu constrângeri* îmbogățește și mai mult abilitățile de rezolvare a problemelor, în special în problemele de satisfacere a constrângerilor. În plus, proeminența Prolog-ului în domeniul Inteligenței Artificiale (IA) și al sistemelor expert îi expune pe profesioniști la tehnologii de vârf, esențiale pentru roluri în cercetarea și dezvoltarea IA. Prolog facilitează, de asemenea, interpretabilitatea modelelor de *Machine Learning* (ML), oferind o abordare transparentă și bazată pe reguli pentru rezolvarea problemelor. Cu Prolog, se introduce o paradigmă de gândire nouă, în care soluțiile sunt derivate din reguli logice și fapte în loc de instrucțiuni explicite. În plus, suportul Prolog-ului pentru recursivitate îl face ușor de înțeles și de implementat pentru algoritmi recursivi, facilitând înțelegerea si aplicarea conceptelor de recursivitate învățate în cursurile de structuri de date și algoritmi. Natura sa bazată pe interpretor permite execuția și testarea imediată a „ideii” (fără necesitatea de a defini și declara structuri, variabile, etc), facilitând totodată dezvoltarea iterativă. În cele din urmă, învățarea Prolog-ului nu numai că îmbunătățește abilitățile de rezolvare a problemelor, dar oferă și un rezumat al structurilor de date și algoritmilor într-o paradigmă nouă și captivantă.

2 Considerații teoretice

2.1 Paradigma logică

Paradigma de programare este un stil fundamental de programare care dictează:

- Modul de **reprezentare** a datelor (ex: fapte, variabile, clase etc.)
- Modul de **prelucrare** a datelor (ex: comparații, evaluări etc.)

Paradigma logică face parte din paradigma declarativă. Un limbaj de programare declarativ răspunde la întrebarea **CE** trebuie să facă un program. Să considerăm următorul exemplu în limbajul declarativ SQL:

```
SELECT nume, prenume FROM Studenți WHERE an = 3
```

Instrucțiunea de mai sus nu îi spune interpretorului SQL cum să realizeze căutarea, ci îi spune doar ce condiție trebuie să respecte rezultatul. Prolog este un limbaj de programare logic, deci implicit și declarativ.

În opoziție, limbajele imperative (ex: C, Pascal, C++, C#, Java etc.) sunt focalizate pe control si ca urmare NECESITA explicitarea răspunsului la întrebarea „**CUM** trebuie să rezolve programul o anumită problemă”.

2.2 Concepte Prolog

Un program sursă Prolog este format dintr-o serie de predicate. Un predicat este o colecție de fapte și reguli. După ce programul sursă Prolog a fost consultat de interpretor, se pot pune întrebări. Răspunsul la întrebări se determină pe baza faptelor și a regulilor, folosind o căutare în adâncime cu backtracking (mecanismul standard de execuție Prolog). Tot ceea ce nu este cunoscut sau nu poate fi demonstrat este considerat fals.

Comentariile:

- *de linie* încep cu simbolul **%**
- *bloc* se încadrează între simbolurile **/*** și ***/** (similar comentariilor din limbajul de programare C)

În cadrul acestui laborator, există două opțiuni de interpretoare Prolog:

2.2.1 (Opțiunea1) SWI Prolog (necesită instalare) & Notepad++

Programul sursă este scris în Notepad++ într-un fișier de format *.pl și va fi consultat în interpretorul SWI Prolog prin menu bar: *File -> Consult*. Pentru încărcările ulterioare ale aceluiași fișier se poate folosi *File -> Reload modified files*.

2.2.2 (Opțiunea2) SWISH Prolog – platformă online (fără instalare)

Platforma online SWISH Prolog[1] încorporează atât editorul cât și interpretorul și permite folosirea ambelor prin intermediul unei interfețe simple.

2.3 Tipuri de date

Singurul tip de date în Prolog este termenul. Termenii pot să fie: atomi, numere, variabile sau termeni compuși (structuri). Figura 1.1 prezintă o clasificare a tipurilor de date în Prolog:
Simple:

- constante
    - numere (ex: 47, 6.3 etc.)
        - În contrast cu C – Prolog nu necesită declararea unui tip de date ca ‚int’ sau ‚float’

[1] https://swish.swi-prolog.org/

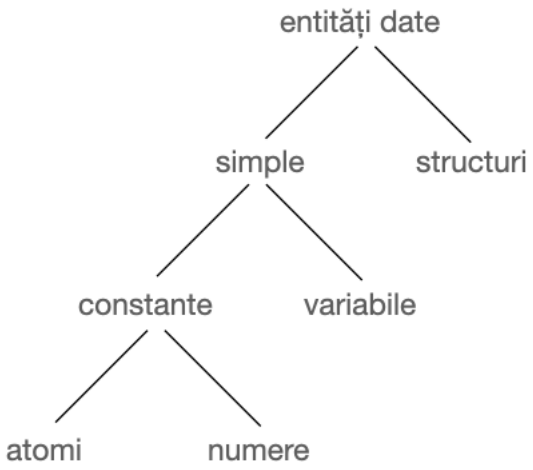


Figura 1.1 – Tipuri de date in Prolog

- simboluri/atom (ex: girafa, 'Romania', 'antilopa Gnu' etc.)
  - Dacă literalul conține caractere speciale trebuie înconjurat de ghilimele simple

- variabile
  - Încep cu **literă mare** sau cu **_** (ex: X, Aux, _12 etc.)
  - Un singur **_** reprezintă o variabilă anonimă/liberă

Compuse:

- structuri (ex: t(1, t(-2, nil, nil), t(8,nil,nil)) etc.)
  - liste (ex: [], [1, 2, 3], [1, 2 | _], [1, 'ana', [3,a]] etc.)
    - [] = lista vidă
    - Lista [1,2,3] este reprezentată intern prin '.'(1, '.'(2, '.'(3, [])))
    - Șablonul **[H|T]** împarte lista în H (= elementul din capul listei) și T (= **lista** ce conține restul elementelor)
  - string-uri (ex: "Hello World")

2.3.1 Exerciții pe Tipuri de Date

1. Ce tip de date sunt următorii termeni:

a.	X	d.	hello	g.	[a, b, c]
b.	'X'	e.	Hello	h.	[A, B, C]
c.	_138	f.	'Hello'	i.	[Ana, are, 'mere']

2. Ce fac următoarele predicate predefinite în Prolog:

```
var(Term),
nonvar(Term),
number(Term),
atom(Term),
atomic(Term)
```

2.3.2 Informații adiționale despre liste in Prolog

Șablonul **[H|T]** poate fi folosit pentru a separa o listă în cap (H) și coadă (T – coada este restul listei și este o listă de sine stătătoare):

[1,2,3] poate fi scris ca [1 | [2,3]]

Întrebare: Parcurgerea unei liste se face prin folosirea șablonului, dar care este tehnica de programare folosită?

Răspuns: Recursivitate. Primul element este separat de coadă și un apel recursiv este aplicat pe coadă, astfel se poate procesa primul element.

Vizualizând șablonul într-o manieră recursivă, lista [1,2,3] poate fi scrisă ca:
[1 | [2 | [3 | []]]]

2.4 Fapte

Faptele sunt predicate care sunt **tot timpul** adevărate. Se mai numesc și axiome (din matematică). Pot avea zero, unu sau mai multe argumente. **Aritatea** unui predicat reprezintă numărul de argumente ale acelui predicat.

Exemple:

```
we_are_in_room_108.
animal(elefant).
animal('antilopa Gnu').
height(girafa, 5.5). % înălțimea în metri
tree( t(1, t(-2, nil, nil), t(8,nil,nil)) ).
```

2.5 Întrebări

Întrebările pot fi văzute ca și scopuri ale programului Prolog. Răspunsul la o întrebare poate fi afirmativ sau negativ. Dacă folosim variabile în întrebare putem obține și alte informații.

Exemple:

```
?- we_are_in_room_108.
true.

?- animal('antilopa Gnu').
true.
?- animal(X).
X = elefant ; % repetă întrebarea cu ; sau n sau spațiu
X = 'antilopa Gnu' ;
false. % nu mai există un alt animal definit în program
```

2.6 Unificare

Simbolul „=” realizează unificarea dintre cei 2 termeni.
Valoare este orice tip de dată care nu este/nu conține o variabilă neinstanțiată.

Termen 1	**Termen 2**	**Operație**
valoare (/variabila instanțiată)	valoare (/variabila instanțiată)	Comparație
valoare (/variabila instanțiată)	variabila neinstanțiată	Asignare
variabila neinstanțiată	valoare(/variabila instanțiată)	Asignare
variabila neinstanțiată	variabila neinstanțiată	Variabilele devin sinonime (aceeași adresă)

Observație: Unificarea din Prolog nu funcționează ca atribuirea din C.

2.6.1 Exerciții folosind unificarea

Testează următoarele unificări

a. ?- a = a.
b. ?- a = b.
c. ?- 1 = 2.
d. ?- 'ana' = 'Ana'.
e. ?- X = 1, Y = X.
f. ?- X = 3, Y = 2, X = Y.
g. ?- X = 3, X = Y, Y = 2.
h. ?- X = ana.
i. ?- X = ana, Y = 'ana', X = Y.
j. ?- a(b,c) = a(X,Y).
k. ?- a(X,c(d,X)) = a(2,c(d,Y)).
l. ?- a(X,Y) = a(b(c,Y),Z).
m. ?- tree(left, root, Right) = tree(left, root, tree(a, b, tree(c, d, e))).
n. ?- k(s(g),t(k)) = k(X,t(Y)).

o. ?- father(X) = X.
p. ?- loves(X,X) = loves(marsellus,mia).
q. ?- [1, 2, 3] = [a, b, c].
r. ?- [1, 2, 3] = [A, B, C].
s. ?- [abc, 1, f(x) | L2] = [abc|T].
t. ?- [abc, 1, f(x) | L2] = [abc, 1, f(x)].

Observație1: O listă în formatul [1,2,3] este echivalentă cu [1,2,3 | []] dar și cu [1 | [2 | [3 | []]]].

Observație2: Șablonul **[H|T]** poate fi extins la [H1, H2|T] sau [H1, H2, H3|T], etc. Este important să reținem că primul element nu poate lipsi, cu toate că T poate să fie listă vidă ([]). Se poate infera astfel că șablonul de tip [H1, H2, ..., Hn|T] poate să fie unificat doar cu o listă cu minim *n* elemente. Acest fapt înseamnă că este necesar să avem la fel de multe condiții de oprire ca numărul de capuri de listă *(Hn)* pentru a adresa toate cazurile posibile.

Observație3: (Rețineți) Unificarea din Prolog este procesul prin care doi termeni se încearcă a fi făcuți egali, indiferent de nivelul de instanțiere (completă, parțială sau deloc) în care se află oricare dintre aceștia la momentul unificării. Chiar data se utilizează același simbol ca în limbaje imperative („=”), semantica este diferita.

2.7 Reguli

Un predicat este format din una sau mai multe clauze *Horn*, având forma:

p(…) dacă p11(…) și p12(…) și … și p1n(…).

p(…) = capul clauzei (maxim un predicat)
p11(…) și … = corpul clauzei (zero, unu sau mai multe predicate)

fapt = clauză fără corp
întrebare = clauză fără cap

Simboluri speciale (operatori) în Prolog:
:- = „dacă”
, = „și”
; = „sau” (poate fi realizat și prin scrierea mai multor clauze pentru același predicat)

Exemple:

```
larger_height(X,Y) :- height(X,Hx), height(Y,Hy), Hx>Hy.
% prima data luam înălțimea lui X, apoi luam înălțimea lui Y
% și în final verificăm inegalitatea

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Intermediar), path(Intermediar,Y).
% drumul între nodurile X și Y poate să fie conexiunea directă
% între cele 2 noduri SAU daca nu există conexiune directă ne
% folosim de o conexiune indirectă printr-un nod intermediar
```

2.8 SWISH *vs* SWI

În imaginea de mai jos se prezintă configurația de lucru bazată pe SWI Prolog & Notepad++.

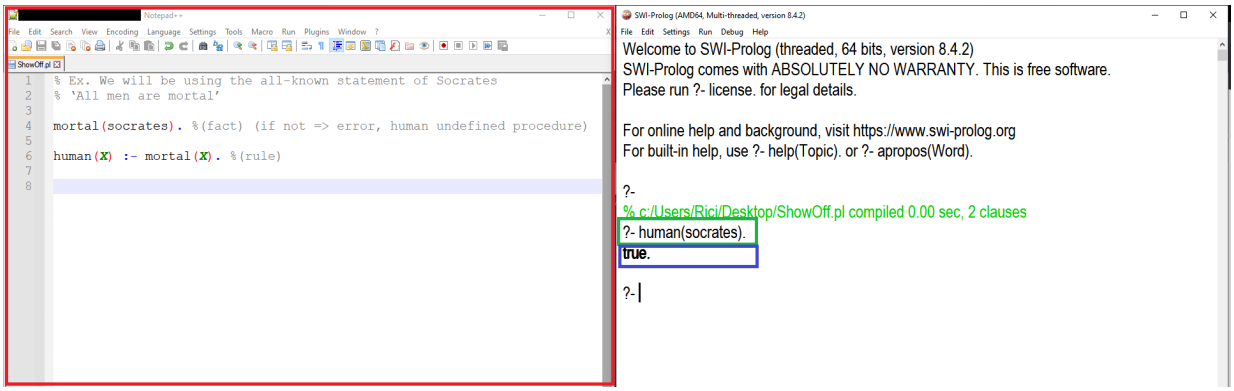


Figura 1.2 – SWI Prolog & Notepad++

Figura 1.3 prezintă site-ul SWISH Prolog. Încadrarea prin culori a fost ajustată părților corespunzătoare celor două elemente.

2.9 Debugging în Prolog

Debugging-ul în Prolog se numește trasare, iar *‚trace'* este comanda Prolog care permite debugging-ul codului Prolog.
O trasare în versiunea SWI este prezentat în Figura 1.4.

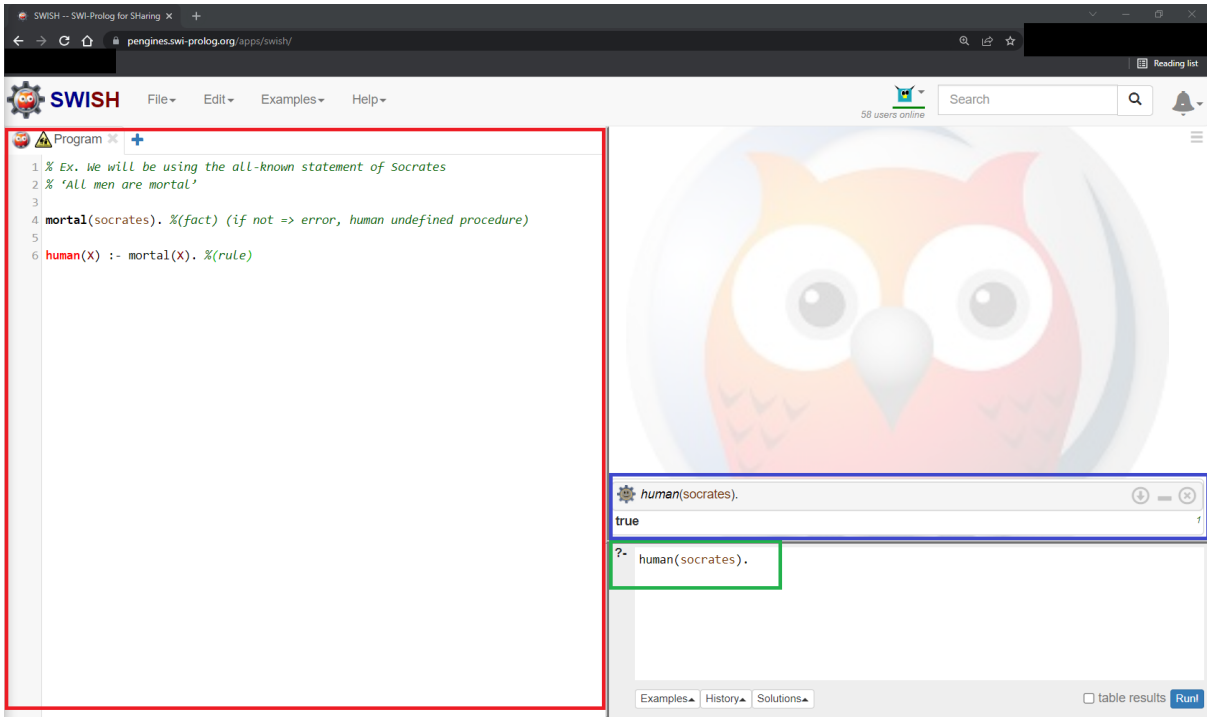


Figura 1.3 – SWISH Prolog

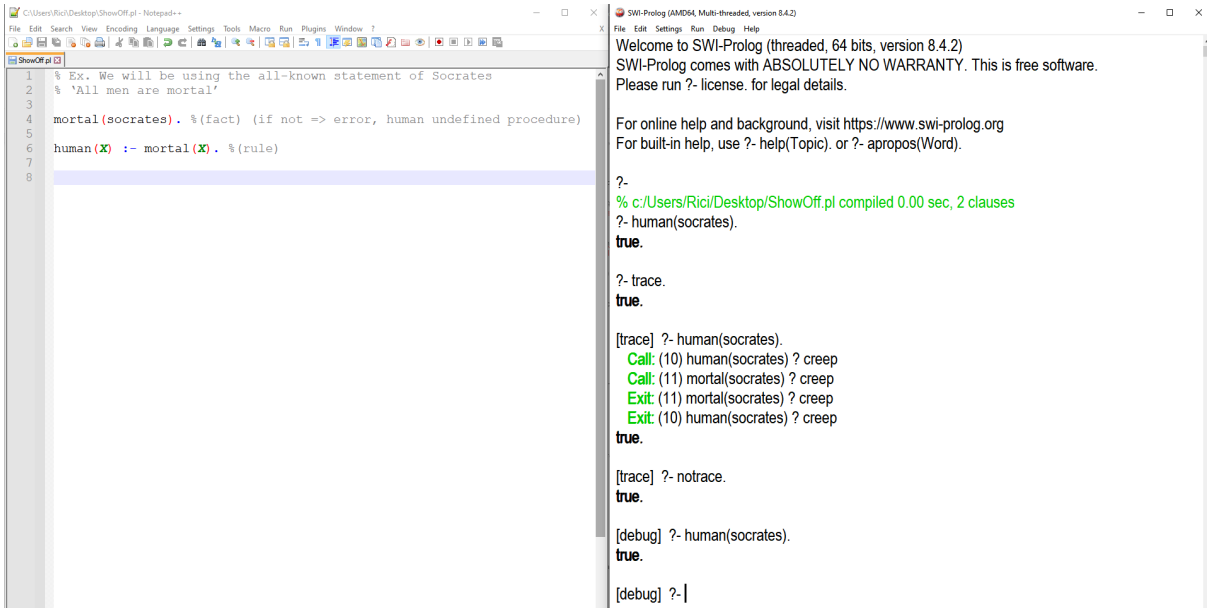


Figura 1.4 – Exemplu trasare în SWI Prolog

Este necesar să activăm opțiunea de trace prin rularea comenzii ‚trace' înainte de întrebare. Poate fi dezactivat prin folosirea comenzii ‚notrace'.

Trasarea în SWISH necesită scrierea comenzii ‚trace' înaintea întrebării (vezi Figura 1.5).

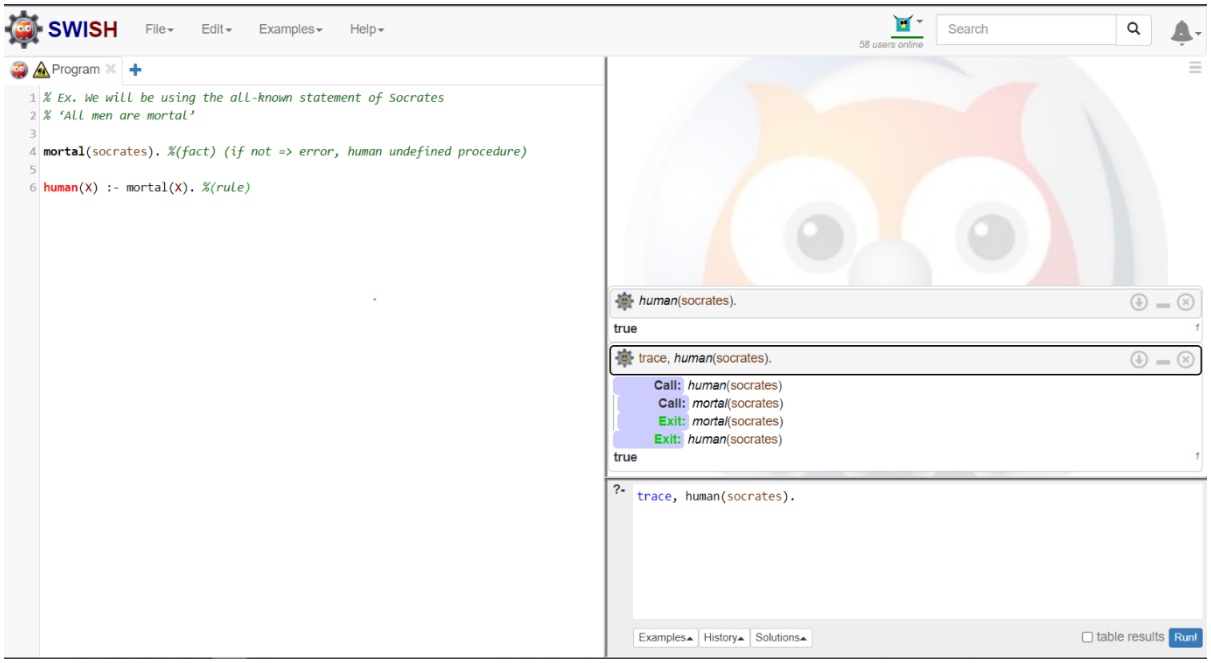


Figura 1.5 – Exemplu trasare în SWISH

2.10 Backtracking

Backtracking-ul reprezintă o funcționalitate implicită a mecanismului de execuție Prolog. Răspunsul la întrebările puse în Prolog este găsit prin efectuarea unei căutări în adâncime, cu backtracking – în spațiul de căutare dat de clauzele din program. Mecanismul de backtracking se declanșează – implicit – în cazul unui eșec la suprapunerea unei întrebări cu un cap de clauză, sau – explicit – la repetarea întrebării (prin **;** sau **n**).

Vom implementa un program Prolog pentru a decide cine poate să țină o petrecere prin parcurgerea unei liste de cerințe:

```
% hold_party/1 este o regulă care depinde de execuția reușită
% a faptelor birthday/1 și happy/1
hold_party(X):-
    birthday(X),
    happy(X).

% o serie de fapte de tip birthday/1 și happy/1
birthday(alex).
birthday(maria).
birthday(adriana).
happy(ana).
happy(george).
happy(adriana).
```

Observație1. Rețineți că "*/x*" reprezintă aritatea unui predicat = numărul de argumente.

Observație2. În general, vrem ca faptele să fie specifice -> după cum puteți vedea, conțin constante. Simultan, vrem ca regulile să fie generale -> folosesc variabile. De ce? Când punem o întrebare în Prolog, vrem sa găsim argumente care satisfac anumite constrângeri specificate. În cazul în care mai multe tuple satisfac acele constrângeri, Prologul le poate identifica implicit, **fără** a explicita mecanismul de backtracking (acesta fiind încorporat în paradigma logică).

Vom urmări execuția predicatului *hold_party/1* în SWISH Prolog:

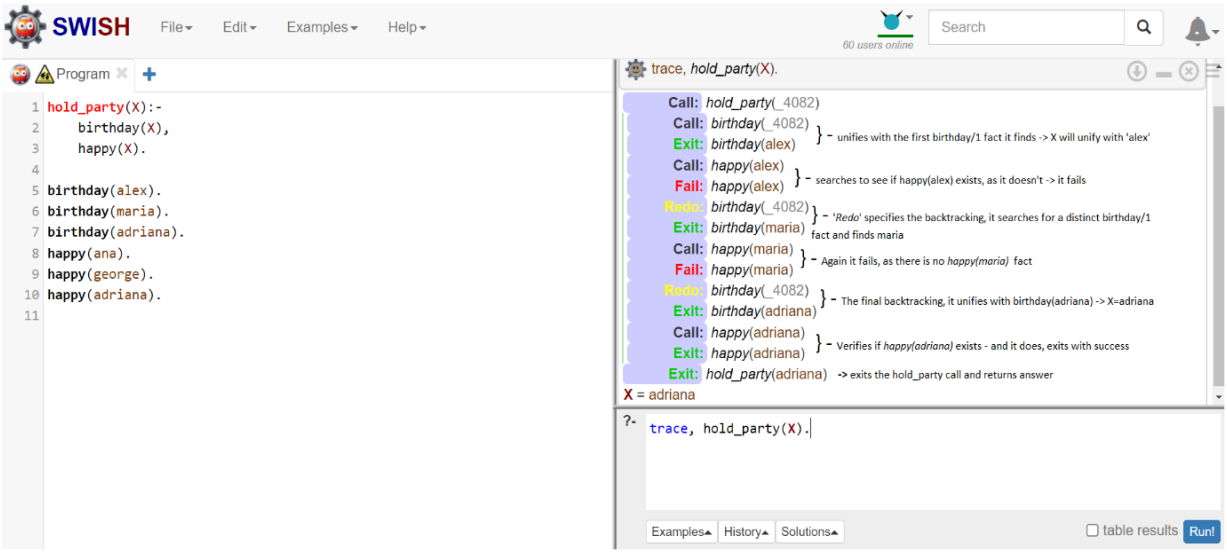


Figura 1.6 – Trasarea întrebării hold_party(X) în SWISH

2.11 Recursivitate

Fiind un limbaj declarativ, Prologul nu conține structuri de control, iterative. Prin urmare, recursivitatea este singurul mecanism prin care se poate implementa un comportament repetitiv în Prolog.
Vom considera de exemplu, un student care a fost acasă după sesiunea de examene și încearcă să ajungă înapoi pentru începerea semestrului.

```
% fapta on_route/1
on_route(camin).
% regula on_route/1 – o regulă recursivă
on_route(Place):-
    move(Place, Method, NewPlace),
    on_route(NewPlace).
```

```
% fapta move/3
move(acasa, taxi, gara).
move(gara, tren, cluj).
move(cluj, bus, camin).
```

Vom urmări execuția predicatului *on_route/1* în SWI-Prolog:

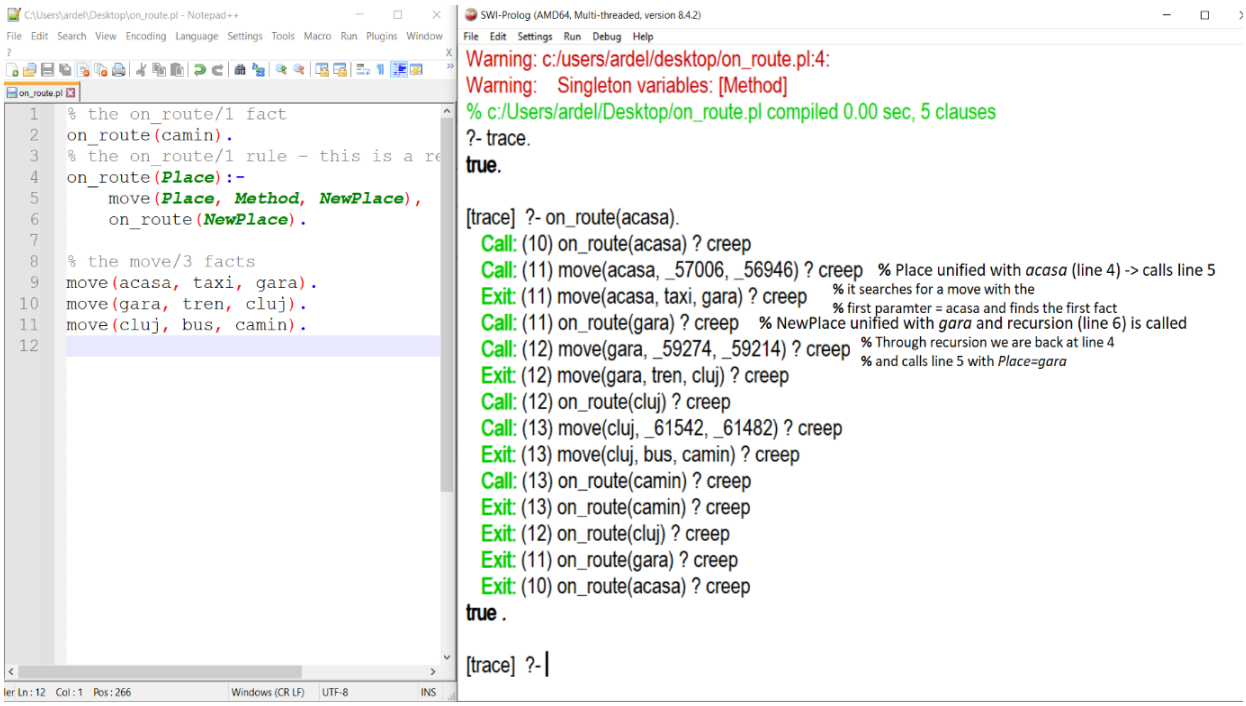


Figura 1.7 – Trasarea întrebării on_route(acasa) în SWI

Observație1. Nu uitați să încărcați fișierul sursă în SWI, folosind *File->Consult*.

Observație2. Nu uitați să activați ‚trace'-ul.

Exercițiu final

Se dă arborele genealogic din Figura 1.8 de mai jos, si o implementare parțială a acestuia (relațiile *man/1*, *woman/1*, *parent/1*, *mother/2*). Se cere să implementați noi predicate pentru relațiile de rudenie cerute.

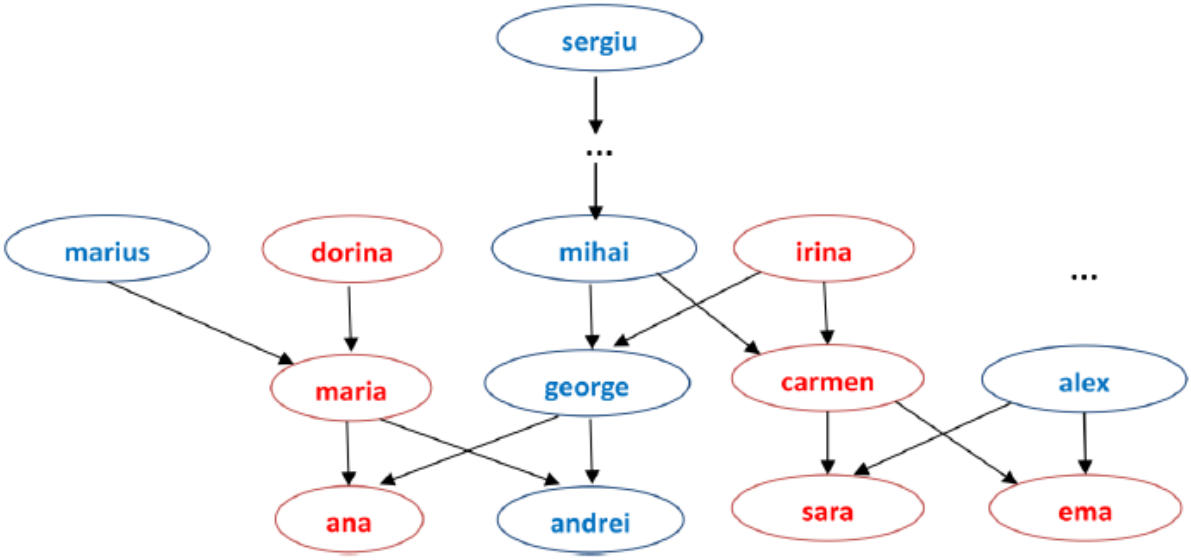


Figura 1.8 – Arbore genealogic

```
% Predicatul woman/1
woman(ana).
woman(sara).
woman(ema).
woman(maria). % … adăugați restul faptelor acestui predicat

% Predicatul man/1
man(andrei).
man(george).
man(alex). % … adăugați restul faptelor acestui predicat

% Predicatul parent/2
parent(maria, ana). % maria este părintele anei
parent(george, ana). % george este părintele anei
parent(maria, andrei).
parent(george, andrei). % … adăugați restul faptelor acestui predicat

% Predicatul mother/2
% X este mama lui Y, daca X este femeie și X este părintele lui Y
mother(X,Y) :- woman(X), parent(X,Y).
```

Pentru SWI Prolog (varianta instalată):

Programul sursă (ex: genealogy.pl) poate fi consultat folosind meniul din interpretor: *File -> Consult* sau se poate scrie în interpretor următorul predicat predefinit:

```
?- consult('C:/Users/student/Desktop/genealogy.pl').
true. % Dacă nu există nici o eroare va returna true
```

1.1. Testați următoarele întrebări:

Rețineți că*: întrebările sunt precedate de operatorul **'?-', *care este deja scris*, restul liniilor reprezintă răspunsurile oferite de Prolog întrebărilor precizate):

```
?- man(george).      % este george bărbat?
true.
?- man(X).           % cine este bărbat?
X = andrei ? ;       % repetăm întrebarea cu ; sau n sau spațiu
X = george ? ;
X = alex ? ;
false.

?- parent(X, andrei). % Cine sunt părinții lui andrei?
X = maria ? ;
X = george ? ;
false.
?- parent(maria, X). % Cine sunt copii mariei?
X = ana ? ;
X = andrei ? ;
false.

?- mother(ana, X). % Cine sunt copii anei?
false.
?- mother(X, ana). % Care este mama anei?
X = maria ? ; % repetăm întrebarea = mai are ana o alta mamă?
false.
```

1.2. Scrieți predicatul *father/2*.

1.3. Completați predicatele *man/1, woman/1* și *parent/2* pentru a acoperi arborele genealogic de mai sus.

1.4. Testați următoarele întrebări:

```
?- father(alex, X).
?- father(X, Y).
?- mother(dorina, maria).
```

1.5. Testați următoarele predicate:

```
% Predicatul sibling/2
% X și Y sunt frați/surori dacă au cel puțin un părinte în comun
% și X diferit de Y
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\=Y.

% Predicatul sister/2
% X este sora lui Y dacă X este femeie și X și Y sunt frați/surori
sister(X,Y) :- sibling(X,Y), woman(X).

% Predicatul aunt/2
% X este mătușa lui Y daca este sora lui Z și Z este părintele lui Y
aunt(X,Y) :- sister(X,Z), parent(Z,Y).
```

1.6. Scrieți predicatele *brother/2*, *uncle/2*, *grandmother/2* și *grandfather/2*.

1.7. Urmăriți pașii pentru găsirea răspunsului la următoarele întrebări (prin utilizarea *trace*-ului):

```
?- aunt(carmen, X).
?- grandmother(dorina, Y).
?- grandfather(X, ana).
```

1.8. Scrieți predicatul *ancestor/2*, unde X este strămoșul lui Y dacă X este legat de Y printr-o serie (indiferent de număr) de relații de tip părinte.

L2. Operații aritmetice

1 Obiective

Lucrarea *L2.Operatii aritmetice* introduce principalele recurențe matematice și tipuri de recursivitate în Prolog.

2 Considerații teoretice

2.1 Operatorul „is”

În Prolog expresiile matematice sunt evaluate **doar la cerere explicită**, 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 neinstanțiate.

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
```

2.2 Cel mai mare divizor comun (CMMDC)

Pentru a calcula cel mai mare divizor comun vom folosi algoritmul lui Euclid. O prima variantă de algoritm (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

Varianta a doua, mai eficientă pentru numere mari (pentru 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 parametri 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) :- Y\=0, Rest is X mod Y, cmmdc2(Y,Rest,Z).
```

Mai jos se prezintă două exemple de urmărire a execuției (vezi Lucrarea 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 apelurilor:

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

Este important să reținem următoarele:

- Predicatele în Prolog returnează valoare de adevăr (*adevărat/fals*). Orice regulă Prolog are în corp o conjuncție de întrebări; pentru a satisface cu succes o suprapunere *întrebare-cap de regulă*, trebuie ca întreaga conjuncție de întrebări din corpul acelei reguli să se evalueze cu succes; la prima întrebare din corp care eșuează, se va declanșa implicit mecanismul de backtracking.

- Atunci când valoarea booleană este insuficientă, deoarece este necesară calcularea unor valori (e.g. numerice, structuri noi, rezultatul unor procesări), în paradigma logica se adaugă câte un argument pentru fiecare termen care reprezintă o valoare „de ieșire” a predicatului. În exemplul prezentat mai sus, al predicatului *cmmdc1*/3, argumentul rezultat este al treilea argument.

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 (backward)

În cadrul recursivității înapoi, rezultatul este construit la întoarcerea din recursivitate. Prin urmare, inițializarea rezultatului se va face pe condiția de oprire.

```
%fact_bwd/2 (intrare, rezultat)

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

Urmărește execuția apelurilor:
?- 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*), execuția va intra într-o buclă infinită – deoarece condiția de oprire va fi ignorată (pe 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(0,1).
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 cazul recursivității înainte, rezultatul este calculat prin acumularea produsului valorilor anterioare, într-un **acumulator**. Acumulatorul trebuie inițializat la apel (pentru a putea permite acumularea valorilor ulterior în el). Pentru a face disponibilă valoarea finală a rezultatului la nivelul apelului inițial, aceasta trebuie asignată unei variabile neinstanțiate în ultimul pas de recursivitate (clauza de terminare). Acea variabila trebuie transmisa de la apelul inițial, pe fiecare apel recursiv, nemodificată.

```
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).

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

Pentru a masca necesitatea inițializării acumulatorului la apel, se poate implementa un predicat wrapper.

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 a celor două tipuri de recursivitate, folosind întrebarea corespunzătoare calculului factorial de 3:

- înapoi / backward:

```
?- 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, întrebarea se poate unifica cu condiția de oprire (clauza 1);
% al doilea argument (1) este folosit pentru a inițializa rezultatul
% execuția se oprește (suprapunere cu un fapt)
```

```
% REMARCAȚI faptul că nicio operație nu a fost făcută până la acest punct
% pentru a calcula factorialul, operațiile sunt făcute la revenirea
% 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 / forward:

După cum a fost menționat anterior, acumulatorul (al doilea argument) trebuie inițializat la apel:

```
?- 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, întrebarea se unifică cu condiția de oprire
% (prima clauză), prin primul argument.
% al doilea argument fiind o variabilă instanțiată și
% al treilea o variabilă neinstanțiată - care a fost transferată până la
% acest punct dintr-un apel recursiv în altul - pot fi unificate

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

```
% După cum puteți observa, nicio operație nu este efectuată după apelurile
% recursive; toate operațiile au fost efectuate înaintea apelurilor recursive,
% de unde si numele: recursivitate înainte
```

Este important sa observăm următoarele:

- În primul rând, 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 argumentul rezultat (al treilea argument) pentru a salva valoarea finală
- Fără a avea o variabilă neinstanțiată (al treilea argument), nu putem să returnăm o valoare nouă – ar returna doar *true* sau *false*. Încercați acest lucru, ștergeți 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 acesta: evaluarea expresiilor matematice impune ca toți operanzii sa fie complet instanțiati.

Observație. În cazul recursivității înapoi, urmăriți ieșirea – acesta se schimbă la fiecare apel recursiv, pe când în varianta de recursivitate înainte este aceeași variabila; acest fapt este o trăsătură inerentă a modalităților diferite de construire a rezultatului în 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. Vom analiza un exemplu de implementare a buclei *for*:

```
for(Inter,Inter,0).
for(Inter,Out,In):-
  In>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 funcționează ca și acumulator.

Observație2. Remarcați că pentru acest predicat, rezultatul este dat de al doilea argument (nu ultimul). În general, argumentul dorit este ultimul ca urmare a bunelor practici, acest lucru nu reprezintă o regulă.

Observație3. Care este scopul liniei îngroșate din clauza a doua (I>0)? Cea mai bună variantă de a afla este de a urmări execuția predicatului *cu* si *fără* această linie.

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 *triangle/3* care verifică dacă cei trei parametrii pot reprezenta lungimile laturilor unui triunghi. (*Hint*: implementați inegalitatea triunghiului).

```
% 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.
```

3. 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.

4. 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.

5. 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.

6. 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.

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.

L3. Operații pe liste

1 Obiective

Lucrarea *L3. Operații pe liste* introduce operațiile fundamentale pe liste: adăugarea, ștergerea, căutarea și înlocuirea de elemente. Aceste predicate sunt deja implementate în Prolog, din acest motiv implementarea noastră va conține și "1" în denumire.

2 Considerații teoretice

2.1 Reprezentarea unei liste

Lista vidă este reprezentată prin simbolul **[].**

O listă care conține cel puțin un element poate fi reprezentată prin șablonul **[H|T]**, unde H este capul listei și poate fi de orice tip, iar T este coada listei și trebuie să fie la rândul ei, o listă.

Observație. Șablonul **[H|T]** nu permite unificarea cu o listă vidă. Încercați întrebarea următoare: *?- [H|T]=[]*. Veți vedea că rezultatul returnat este **false.**

Exemple:

```
?- L=[1,2,3].
?- [1,2,3]=[1|[2|[3]]].
?- L=[a,b,c].
?- L=[a, [b, [c]]].
?- L=[a, [b], [[c]]].
```

Observație. Șablonul **[H|T]** poate face **descompunere** sau **concatenare** în funcție de necesitate. Regula este următoarea. Când *H* și *T* sunt neinstanțiate șablonul *[H|T]* face descompunere, pe când dacă sunt instanțiate face concatenare. Încercați întrebarea următoare:

```
?- [H|T]=[1,2,3], X=3, L=[X|T].
```

2.2 Predicatul „member”

Predicatul *member(X,L)* verifică dacă un elementul *X* este inclus lista *L*. Recurența matematică poate fi formulată în felul următor:

$$x \in L \Leftrightarrow x = primul(L) \vee x \in coada(L)$$

În Prolog se va scrie:

```
member1(X, [H|T]) :- H=X.
member1(X, [H|T]) :- member1(X, T).
```

Simplificarea predicatului:

- Prima clauză a predicatului *member1/2* poate fi simplificată prin înlocuirea lui *H* din capul clauzei cu *X* (member1(X, [X|T]).)
- În a doua clauză, variabila *H* nu este folosită și atunci o putem înlocui cu _ (simbolul pentru o variabilă anonimă).
- Aceeași înlocuire o putem face și pentru *T* din prima clauză.

Astfel predicatul *member1/2* devine:

```
member1(X, [X|_]).
member1(X, [_|T]) :- member1(X, T).
```

Exemplu de urmărire a execuției (cu *trace*):

```
[trace] 3 ?- member1(3,[1,2,3,4]).
   Call: (7) member1(3, [1,2,3,4]) ?      % se unifică cu clauza 2
   Call: (8) member1(3, [2,3,4]) ?        % apelul recursiv din clauza 2
   Call: (9) member1(3, [3, 4]) ?         % se unifică cu clauza 1
   Exit: (9) member1(3, [3, 4])           % succes și iese din apelul recursiv
   Exit: (8) member1(3, [2, 3, 4]) ?
   Exit: (7) member1(3, [1, 2, 3, 4]) ?
true ;                                    % repetăm întrebarea
   Redo: (9) member1(3, [3, 4]) ?         % ultima unificare (cea cu clauza 1) se
                                          % va relua și se va unifica cu clauza 2
   Call: (10) member1(3, [4]) ?           % se unifică cu clauza 2
   Call: (11) member1(3, []) ?            % apelul recursiv din clauza 2
   Fail: (11) member1(3, []) ?            % eșuează (nu există clauză care să
                                          % conțină în al doilea argument
                                          % lista vidă)
   Fail: (10) member1(3, [4]) ?
   Fail: (9) member1(3, [3, 4]) ?
   Fail: (8) member1(3, [2, 3, 4]) ?
   Fail: (7) member1(3, [1, 2, 3, 4]) ?
```

false.

Urmăriți execuția la:
?- member1(a,[a, b, c, a]).
?- X=a, member1(X, [a, b, c, a]).
?- member1(a, [1,2,3]).

Exemplu de comportament nedeterminist la predicatul *member*:
?- member1(X, [1,2,3]).
X = 1 ; **% la repetarea întrebării se alege următoarea soluție posibila**
X = 2 ;
X = 3 ;
false. **% nu mai exista alte soluții**

?- member1(1, L).
L = [1|_] ;
% prima soluție este o listă care începe cu 1, continuată de orice
L = [_, 1|_] ;
% a doua soluție este o listă care are 1 pe poziția 2, continuată de orice
L = [_, _, 1|_]
% etc.

2.3 Predicatul „append”

Predicatul *append(L1, L2, R)* realizează concatenarea listelor *L1* și *L2* și pune rezultatul în parametrul *R*. Recurența matematică poate fi formulata în felul următor:

$$L_1 \oplus L_2 = R \Leftrightarrow primul(R) = primul(L_1) \wedge coada(R) = coada(L_1) \oplus L_2$$

În cazul în care L1 este o lista vidă atunci R va fi egal cu L2. În Prolog, se va scrie ca:

```
append1([], L2, R) :- R=L2.
append1([H|T], L2, R) :- append1(T, L2, CoadaR), R=[H|CoadaR].
```

Putem simplifica predicatul prin înlocuirea argumentului *R* din capul celor 2 clauze cu ultima unificare.

```
append1([], L2, L2).
append1([H|T], L2, [H|CoadaR]) :- append1(T, L2, CoadaR).
```

Observație. Ce fel de recursivitate avem aici? Înainte sau înapoi? Explicați.

Observație. Nu uitați că aceste două variante (cu unificare explicită și implicită) sunt *echivalente*. Chiar dacă nu pare, ambele folosesc același tip de recursivitate și funcționează în același fel. Schimbările aduse in cele două clauze ale predicatului sunt identice, în loc să specificăm explicit unificarea, ea se face implicit.

Exemplu de urmărire a execuției (cu trace):

```
[trace] 6 ?- append1([a,b],[c,d],R).
   Call: (7) append1([a, b], [c, d], _G1680) ?     % se unifică cu clauza 2 -> apoi apel recursiv
   Call: (8) append1([b], [c, d], _G1762) ?        % se unifică cu clauza 2 -> apoi apel recursiv
   Call: (9) append1([], [c, d], _G1765) ?         % se unifică cu clauza 1
   Exit: (9) append1([], [c, d], [c, d]) ?   % succes și iese din apelul recursiv
   Exit: (8) append1([b], [c, d], [b, c, d]) ?
   Exit: (7) append1([a, b], [c, d], [a, b, c, d]) ?
R = [a, b, c, d]. % nu mai exista o alta soluție
```

Observație. Urmăriți ce se întâmplă. Prima listă este traversată până când ajunge să fie vidă, moment în care (la condiția de oprire) rezultatul este instanțiat cu a doua listă (R=[c, d]). Prin revenirea din apelurile recursive succesive, prima listă este concatenată la începutul rezultatului, întâi devine R==[b,c,d] și apoi R=[a,b,c,d]. Prin urmare, este recursivitate **înapoi**. Ce se întâmplă dacă schimbăm recursivitatea înapoi cu una înainte?

Exemplu de comportament nedeterminist la predicatul *append*:

```
?- append1(L1, L2, [1,2,3]).
L1 = [],
L2 = [1, 2, 3] ;              % prima soluție

L1 = [1],
L2 = [2, 3] ;                 % a doua soluție

L1 = [1, 2],
L2 = [3] ;                    % a treia soluție

L1 = [1, 2, 3],
L2 = [] ;                     % nu mai există alte soluții

false.
```

În termeni simpli, punem Prolog-ului următoarea întrebare: "Care două liste concatenate rezultă în [1,2,3]?" și ne returnează toate soluțiile posibile. Astfel, ideea clasică de ce reprezintă o intrare și o ieșire nu se aplică aici.

Urmăriți execuția apelurilor:
?- append1([1, [2]], [3|[4, 5]], R).
?- append1(T, L, [1, 2, 3, 4, 5]).
?- append1(_, [X|_], [1, 2, 3, 4, 5]).

Vă amintiți observația despre șablonul [H|T] si inabilitatea de a se unifica cu lista vidă []? Inversați ordinea clauzelor predicatului *append* și urmăriți execuția întrebărilor de mai sus. Ce diferențe apar în comportamentul predicatului?

2.4 Predicatul „delete"

Predicatul *delete(X, L, R)* șterge prima apariție a elementul X din lista L și pune rezultatul în R. Dacă X nu există în L atunci R va fi egal cu L. Recurența matematică poate fi formulată în felul următor:

$$R = L - \{x\} = \begin{cases} \{\}, & L = \{\} \\ coada(L), & x = primul(L) \\ \{primul(L)\} \oplus (coada(L) - \{x\}), & altfel \end{cases}$$

În Prolog se va scrie:

```
delete1(X, [X|T], T).
% șterge prima apariție și se oprește
delete1(X, [H|T], [H|R]) :- delete1(X, T, R).
% altfel iterează peste elementele listei
delete1(_, [], []).
% daca a ajuns la lista vida înseamnă că elementul nu a fost găsit
% și putem returna lista vidă
```

Observație1. Observați că predicatul *delete/3* are de fapt 2 condiții de oprire (prima si a treia clauză). Prima clauză se unifică când elementul a fost găsit și șterge elementul, pe când a treia clauză se unifică doar dacă elementul nu apare în listă sau se dă ca listă de intrare lista vidă.
Observație2. Aici avem recursivitate înapoi. Al doilea argument (lista de intrare), in a treia clauză, se unifică cu lista vidă și astfel se oprește parcurgerea listei, pe când al treilea argument - instanțiază rezultatul.

Urmărește execuția apelurilor:
?- delete1(3, [1, 2, 3, 4], R).

?- X=3, delete1(X, [3, 4, 3, 2, 1, 3], R).
?- delete1(3, [1, 2, 4], R).
?- delete1(X, [1, 2, 4], R).

Observație. Ce se întâmplă cu aceste întrebări dacă scoatem a treia clauză a predicatului?

2.5 Predicatul „delete_all”

Predicatul *delete_all(X, L, R)* va șterge toate aparițiile lui X din lista L și va pune rezultatul în R. Recurența matematică poate fi formulata în felul următor:

$$R = L - \{x\} = \begin{cases} \{\}, & L = \{\} \\ coada(L) - \{x\}, & x = primul(L) \\ \{primul(L)\} \oplus (coada(L) - \{x\}), & altfel \end{cases}$$

Predicatul *delete_all* diferă față de predicatul *delete* doar la prima clauză.

```
delete_all(X, [X|T], R) :- delete_all(X, T, R).
% dacă s-a șters prima apariție se va continua și pe restul elementelor
delete_all(X, [H|T], [H|R]) :- delete_all(X, T, R).
delete_all(_, [], []).
```

Observație. Observăm că diferența dintre *delete_all/3* și *delete/3* apare când ajungem la elementul pe care vrem să îl ștergem (prima clauză). Restul predicatului rămâne la fel. În predicatul *delete/3*, prima clauză este o condiție de oprire și astfel șterge doar prima apariție a elementului respectiv. Prin modificarea într-o clauza recursivă, va șterge toate aparițiile acestui element.

Clarificare Suplimentară. Putem extinde predicatul *delete_all/3* pentru a arăta de ce este în continuare recursivitate înapoi și faptul că unificarea se face implicit.

```
delete_all(X, [X|T], R) :- delete_all(X, T, R1), R=R1.
delete_all(X, [H|T], R):- delete_all(X, T, R1), R=[H|R1].
delete_all(_, [], []).
```

Adițional, prima clauză a predicatului *detele_all/3* conține încă o simplificare (unificare implicită):

```
delete_all(X, [H|T], R) :- X=H, delete_all(X, T, R1), R=R1.
```

Veți observa că acest predicat permite backtracing, o opțiune de reducere a ramurilor de backtracing este de a adăuga condiția opusă în a doua clauză:

```
delete_all(X, [H|T], R):- not(X=H), delete_all(X, T, R1), R=[H|R1].
```

Urmăriți execuția apelurilor:

```
?- delete_all(3, [1, 2, 3, 4], R).
?- X=3, delete_all(X, [3, 4, 3, 2, 1, 3], R).
?- delete_all(3, [1, 2, 4], R).
?- delete_all(X, [1, 2, 4], R).
```

3 Exerciții

1. Scrieți predicatul *add_first(X,L,R)* care adaugă X la începutul listei L și pune rezultatul în R. *Sugestie*: simplificați pe cât de mult posibil acest predicat.

```
% add_first(X,L,R). – adaugă X la începutul listei L și pune rezultatul în R
?- add_first(1,[2,3,4],R).
R=[1,2,3,4].
```

2. Scrieți predicatul *append3/4* care să realizeze concatenarea a 3 liste. *Sugestie*: Nu folosiți append-ul a două liste.

```
% append3(L1,L2,L3,R). – va realiza concatenarea listelor L1,L2,L3 în R
?- append3([1,2],[3,4,5],[6,7],R).
R=[1,2,3,4,5,6,7] ;
false.
```

3. Scrieți un predicat care realizează suma elementelor dintr-o lista dată.

```
% sum(L, S). – calculează suma elementelor din L și returnează suma în S
?- sum_bwd([1,2,3,4], S).
R=10.

?- sum_fwd([1,2,3,4], S).
R=10.
```

4. Scrieți un predicat care separă numerele pare de cele impare. (*Întrebare*: de ce avem nevoie pentru recursivitate înainte?)

```
?- separate_parity([1, 2, 3, 4, 5, 6], E, O).
E = [2, 4, 6]
O = [1, 3, 5] ;
false
```

5. Scrieți un predicat care să șteargă toate elementele duplicate dintr-o listă.

```
?- remove_duplicates([3, 4, 5, 3, 2, 4], R).
R = [3, 4, 5, 2] ; % păstrează prima apariție
false
```

SAU

```
R = [5, 3, 2, 4] ; % păstrează ultima apariție
false
```

6. Scrieți un predicat care să înlocuiască toate aparițiile lui X în lista L cu Y și să pună rezultatul în R.

```
?- replace_all(1, a, [1, 2, 3, 1, 2], R).
R = [a, 2, 3, a, 2] ;
false
```

7. Scrieți un predicat care șterge tot al k-lea element din lista de intrare.

```
?- drop_k([1, 2, 3, 4, 5, 6, 7, 8], 3, R).
R = [1, 2, 4, 5, 7, 8] ;
false
```

8. Scrieți un predicat care șterge duplicatele consecutive fără a modifica ordinea elementelor din listă. *Sugestie*: căutați observația despre șablonul extins în *Lucrarea nr. 1*.

```
?- remove_consecutive_duplicates([1,1,1,1, 2,2,2, 3,3, 1,1, 4, 2], R).
R = [1,2,3,1,4,2] ;
false
```

9. Scrieți un predicat care adăugă duplicatele consecutive într-o sub-listă fără a modifica ordinea elementelor din listă.

```
?- pack_consecutive_duplicates([1,1,1,1, 2,2,2, 3,3, 1,1, 4, 2], R).
R = [[1,1,1,1], [2,2,2], [3,3], [1,1], [4], [2]];
false
```

L4. Tăierea de backtracking

1 Obiective

În lucrarea *L4. Taierea de backtracking* vom introduce o metodă de îmbunătățire a performanței la execuția unui predicat, prin reducerea spațiului de căutare. De asemenea, vom continua cu operații pe liste folosind cele două tipuri de recursivitate: înainte și înapoi.

2 Considerații teoretice

2.1 Operatorul „!”

Operatorul **!** de tăiere este o proprietate a Prolog-ului care șterge ramurile alternative de căutare a soluției, astfel acele ramuri nu sunt explorate de backtracking. O dată ce s-a trecut peste operator, apelurile din fața operatorului și predicatul inițial *NU* mai pot căuta o altă clauză cu care să se unifice, deci nu mai pot căuta o altă soluție.

Acest operator poate îmbunătăți eficiența programelor Prolog prin evitarea căutării de soluții alternative atunci când (i) se știe că nu există sau (ii) nu ne interesează soluții alternative, chiar daca acestea există.

În caz general putem scrie o clauză care folosește operatorul de tăiere în felul următor:

p :- a_1, a_2, …, a_k, !, a_{k+1}, …, a_n.

Daca a_{k+1} eșuează, nu se va mai căuta o altă clauză care să se unifice cu a_k, a_{k-1}, …, a_1 și p. Un caz simplu de utilizare ar fi atunci când avem 2 clauze cu apeluri inițiale mutual exclusive.

Exemplu:

```
% Varianta 1 fără !
p(X) :- X>0, a, … .
p(X) :- X=<0, b, … .

% Varianta 2 cu !
p(X) :- X>0, !, a, … .
p(X) :- b, … .
```

În concluzie, nodul p și nodurile a1, …, ak nu pot să facă backtracking. Ce trebuie să rețineți despre operatorul de tăiere de backtracking este:

- Tăierea de backtracking se execută mereu cu succes, nu face backtracking
- Nicio clauză din corpul curent, apelată înainte de acest operator (la stânga lui), nu va fi reconsiderată
- Nicio clauză ulterioară regulii curente nu va fi reconsiderată
- Orice variabilă unificată înainte de operator nu va putea lua alte valori

O potențială aplicație a operatorului „!” reiese din acestea. Amintiți-vă exercițiul din laboratorul anterior. Vom folosi ca exemplu predicatul *separate_parity/3*:

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   separate_parity(T, E, O).
```

Dacă interogăm acest predicat, primul răspuns va fi corect, dar următoarele răspunsuri vor muta elemente pare în lista celor impare, prin backtracking. De exemplu, la prima întrebare a *?-separate_parity([1,2,3,4], E, O)*, 4 a trecut prin *0 is H mod 2* și a fost evaluat la *true*. Dacă repetăm întrebarea, prin backtracking, acel apel se va unifica cu a treia clauză, unde *4* va fi adăugat la lista celor impare.

```
?- separate_parity([1,2,3,4], E, O).
E = [2, 4],
O = [1, 3];
E = [2],
O = [1, 3, 4];
…
```

Cum putem opri acest comportament? O primă opțiune este să adăugăm condiția *opusă* pe a treia clauză negarea lui *0 is H mod 2*, care în acest caz ar fi 1 is H mod 2.

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   1 is H mod 2,
   separate_parity(T, E, O).
```

?- separate_parity([1,2,3,4], E, O).
E = [2, 4],
O = [1, 3];
false

Prin adăugarea acestei condiții, primul răspuns va rămâne cel corect, iar la repetarea întrebării vom primi drept răspuns *false* (comportament corect), întrucât clauzele predicatului devin mutual exclusive.

Același efect poate fi obținut prin adăugarea operatorului „!" dupa condiția *0 is H mod 2* (condiția opusă nemaifiind necesară în a treia clauză) :

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2, !,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   separate_parity(T, E, O).
```

?- separate_parity([1,2,3,4], E, O).
E = [2, 4],
O = [1, 3].

Urmează să reluăm predicate din laboratorul anterior: *member/2* și *delete/3*. După cum ați observat anterior, ambele predicate permit comportament non-determinist. Prin adăugarea operatorului „!", putem să le transformăm în predicate deterministe.

2.1.1 Predicatul „member" – varianta deterministă

Versiunea deterministă a predicatului *member/2*:

```
member1(X, [X|_]) :- !.
member1(X, [_|T]) :- member1(X, T).
```

Exemplu de urmărire a execuției (cu trace):
[trace] ?- member1(X,[1,2,3]).
   Call: (7) member1(_G1657, [1, 2, 3]) ?
   Exit: (7) member1(1, [1, 2, 3]) ?
X = 1. **% nu mai putem repeta întrebarea pentru că nu mai poate să caute o altă clauză cu care să se unifice**

După cum se poate observa din urmărirea apelului, tăierea de backtracking nu permite nodului corespunzător apelului de *member1(_G1657,[1, 2, 3])* să fie

unificat cu alte clauze și variabila *_G1657* să fie unificată cu altă valoare. Astfel, când repetăm întrebarea, returnează *false*.

Urmăriți execuția apelurilor:
?- X=3, member1(X, [3, 2, 4, 3, 1, 3]).
?- member1(X, [3, 2, 4, 3, 1, 3]).

2.1.2 Predicatul „delete" cu tăiere de backtracking

Predicatul *delete/3* din lucrarea anterioară ștergea o apariție (defapt, chiar prima) a unui element (primul argument) la un apel. Cum am face dacă am vrea versiunea deterministă acestui predicat, predicatul care șterge prima și doar prima apariție a unui element în listă? Această versiune a predicatului delete este prezentată mai jos:

```
delete1(X, [X|T], T) :- !.
delete1(X, [H|T], [H|R]) :- delete1(X, T, R).
delete1(_, [], []).
```

Exemplu de urmărire a execuției (cu trace):
[trace] ?- delete1(3,[4,3,2,3,1],R).
  Call: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% se unifică cu clauza 2 și apoi apel recursiv**
  Call: (8) delete1(3, [3, 2, 3, 1], _G1786) ? **% se unifică cu clauza 1**
  Exit: (8) delete1(3, [3, 2, 3, 1], [2, 3, 1]) ? **% succes și iese din apelul recursiv**
  Exit: (7) delete1(3, [4, 3, 2, 3, 1], [4, 2, 3, 1]) ?
R = [4, 2, 3, 1] ; **% repetăm întrebare**
  Redo: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% nu se poate anula unificarea cu clauza 1 și atunci se urca în stiva, la apelul precedent, se anulează unificarea cu clauza 2 și se încearcă unificarea cu clauza 3**
  Fail: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% eșec**
false.

Astfel, această versiune a predicatului *delete/3* șterge doar prima apariție a elementului.

Urmăriți execuția apelului:
?- delete1(X, [3, 2, 4, 3, 1, 3], R).

2.2 Predicatul „length"

Predicatul *length(L,R)* calculează numărul de elemente din lista *L* și pune rezultatul în *R*. Recurența matematică poate fi formulată astfel:

$$lungime(L) = \begin{cases} 0, & L = [] \\ 1 + lungime\big(coada(L)\big), & altfel' \end{cases}$$

adică:

- Lungimea listei vide este 0
- Lungimea unei liste nevide [H|T] este lungimea lui T plus 1

În Prolog se va scrie:

```
% Varianta 1 (recursivitate înapoi)
length1([], 0).
length1([_|T], Len) :- length1(T, Lcoada), Len is 1+Lcoada.
```

Această versiune a predicatului *length* folosește o abordare de recursivitate înapoi. Astfel, rezultatul la nivelul *i*, are nevoie de rezultatul de la nivelul *i-1*. Lungimea (al doilea argument) este inițializată când apelul recursiv ajunge să fie unificat cu condiția de oprire (prima clauză) și este construit progresiv la fiecare întoarcere dintr-un apel recursiv prin incremente de unu.

```
% Varianta 2 (recursivitate înainte -> acumulator = al doilea argument)
length2([], Acc, Len) :- Len=Acc.
length2([_|T], Acc, Len) :- Acc1 is Acc + 1, length2(T, Acc1, Len).

length2(L, R) :- length2(L, 0, R).
% length2/2 = wrapper al predicatului length2/3 care folosește un acumulator
```

Cealaltă abordare este să numărăm elementele din listă în timp ce lista este descompusă (prin șablonul [H|T]) și să construim lungimea (în acumulator) în timp ce lista este parcursă prin recursivitate. Această abordare este cea de recursivitate înainte. Pentru a folosi această abordare, acumulatorul trebuie să fie inițializat cu 0 la fiecare întrebare. Cu fiecare element nou descoperit, lungimea crește cu 1. Pentru a face rezultatul disponibil, avem nevoie să unificăm acumulatorul cu o variabilă neinstanțiată (*al treilea argument*) care a fost pasat prin toate apelurile recursive fără a fi instanțiată.
Rețineți. În contrast cu alte limbaje de programare, în Prolog expresiile nu sunt evaluate implicit. Pentru a evalua o expresie trebuie să folosim operatorul **is.** Mod de folosire: <Variabilă> is <expresie>.

Urmăriți execuția apelurilor:
?- length1([a, b, c, d], Len).
?- length1([1, [2], [3|[4]]], Len).
?- length2([a, b, c, d], 0, Res).
?- length2([a, b, c, d], Len).

?- length2([1, [2], [3|[4]]], Len).
?- length2([a, b, c, d], 3, Len).

2.3 Predicatul „reverse”

Predicatul *reverse(L,R)* inversează ordinea elementelor din lista *L*. Recurența matematică poate fi formulată astfel:

$$invers(L) = \begin{cases} [], & L = [] \\ invers\big(coada(L)\big) \oplus \{primul(L)\}, & altfel' \end{cases}$$

adică:

- Inversul listei vide [] este [], și
- Inversul unei liste non-vide [H|T] poate fi obținut prin inversarea T-ului și prin adăugarea H-ului la finalul listei rezultate

În Prolog se va scrie:

```
% Varianta 1 (recursivitate înapoi)
reverse1([], []).
reverse1([H|T], R) :- reverse1(T, Rcoada), append1(Rcoada, [H], R).
```

În primul rând, obținem inversul cozii *T*, denumit *Rcoada* și construim inversul listei *L=[H|T]*, prin concatenarea primul element H la finalul lui *Rcoada*.

```
% Varianta 2 (recursivitate înainte -> acumulator = al doilea argument)
reverse2([], Acc, R) :- Acc=R.
reverse2([H|T], Acc, R) :- Acc1=[H|Acc], reverse2(T, Acc1, R).

reverse2(L, R) :- reverse2(L, [], R).
% reverse2/2 = wrapper a predicatului reverse2/3 care
% folosește un acumulator
% În contrast cu acumulatoarele de până acum, aici avem operații cu liste,
% astfel va fi instanțiată cu o listă vidă
```

În a doua clauză, elementele vor fi adăugate în fața acumulatorului când sunt descoperite. Întrucât acest comportament poate să pară contra-intuitiv, să analizăm urmatorul exemplu:

?- reverse2([1,2,3], [], R).

reverse2([1,2,3], [], R).
Acc1 = [1 | []] = [1]

```
reverse2([2,3], [1], R).
        Acc1 = [2 | [1]] = [2,1]
        reverse2([3], [2,1], R).
                ...
```

Astfel, când lista de intrare devine lista vidă (prima clauză), lista inversată este deja în acumulator. Ultimul lucru care trebuie făcut este unificarea acumulatorului (clauza 1) cu variabila (până atunci) neinstanțiată (al treilea argument).

Urmăriți execuția apelurilor:

```
?- reverse1([a, b, c, d], R).
?- reverse1([1, [2], [3|[4]]], R).
?- reverse2([a, b, c, d], R).
?- reverse2([1, [2], [3|[4]]], R).
?- reverse2([a, b, c, d], [1, 2], R).
```

2.4 Predicatul minim

Predicatul *min(L,M)* determină elementul minim din lista *L*. Dacă lista este vidă, va returna *false*. Recurența matematică poate fi formulată astfel:

$$min(L) = \begin{cases} primul(L), & daca\ primul(L) < min(coada(L)) \\ min(coada(L)), & altfel \end{cases}$$

În Prolog se va scrie:

```
% Varianta 1 (recursivitate înainte -> acumulator = al doilea argument)
min1([H|T], Mp, M) :- H<Mp, !, min1(T, H, M).
min1([_|T], Mp, M) :- min1(T, Mp, M).
min1([], Mp, M) :- M=Mp.

min1([H|T], M) :- min1(T, H, M).
% În contrast cu acumulatoarea folosite până acum
% pentru predicatul min1/3,
% acumulatorul (al doilea argument) va fi inițializat cu primul element
% Într-un mod similar, min1/2 este un wrapper.
```

O primă soluție pentru determinarea elementului minim într-o listă este să parcurgem elementele unul câte unul și să păstrăm, la fiecare pas, elementul minim curent. Când lista devine vidă, minimul parțial devine minim global. Această abordare corespunde unei strategii de recursivitate înainte. Primele 2 clauze ale predicatului parcurg lista: prima clauză acoperă cazul în care minimul parțial trebuie să fie actualizat (un nou minim parțial a fost găsit ca primul element curent), pe când în a doua clauză minimul este pasat mai departe fără

să fie modificat. Ultima clauză reprezintă condiția de oprire: lista devine vidă, astfel minimul parțial este unificat cu variabila (până atunci) neinstanțiată (al treilea argument).

```
% Varianta 2 (recursivitate înapoi)
min2([H], H).
min2([H|T], H):-min2(T, M), H<M.
min2([H|T], M):-min2(T, M), H>=M.
```

Această abordare utilizează recursivitatea înapoi: minimul este actualizat la revenirea din recursivitate (clauzele 2-3 actualizează minimul la revenire). Minimul este inițializat în condiția de oprire (clauza 1), cu ultimul element. *Rețineți*: nu este nevoie de un argument adițional (ca la recursivitatea înainte).

Studiați (folosind trace) execuția apelurilor:
?- min2 ([1, 2, 3, 4], M).
?- min2([4, 3, 2, 1], M).
?- min2 ([3, 2, 6, 1, 4, 1, 5], M).
?- min2([], M).

Repetați întrebările. Câte răspunsuri are fiecare întrebare? Care este ordinea răspunsurilor? (dacă se aplică). Ce complexitate au cele două soluții?

Predicatul *min2/2* poate fi îmbunătățit considerând următoarele observații:

- Cele două condiții din clauzele 2 și 3 ($H<M$ și $H>=M$) sunt complementare, si sunt utilizate pentru a lua decizia asupra cărui rezultat trebuie trimis mai sus, în funcție de rezultatul pe restul listei – minimul găsit pe T.
- In clauza a doua, minimul va fi reinițializat, pentru ca M nu poate fi minimul listei [H|T] – acesta este H; prin urmare, este necesara calcularea lui? Dacă inversam clauzele 2 cu 3, putem ajunge la următoarea formă:

```
min2([H], H).
min2([H|T], M):- min2(T, M), H>=M, !.
min2([H|T], H).
```

- Prin analiza clauzelor 1 și 3, putem observa că prima este un caz particular a celei de-a treia (De ce?). Prin urmare, putem renunța la ea. Versiunea finală a predicatului minim folosind recursivitate înapoi:

```
min2([H|T], M) :- min2(T, M), M<H, !.
min2([H|_], H).
```

Ce complexitate are aceasta ultimă varianta a predicatului *min2/2*?

Urmăriți execuția apelurilor:
?- min1([], M).
?- min1([3, 2, 6, 1, 4, 1, 5], M).
?- min2([], M).
?- min2([3, 2, 6, 1, 4, 1, 5], M).

Puteți identifica diferența dintre implementarea inițială si cea finală a predicatului?
Observație. Predicatul *min1/3* care folosește recursivitate înainte inițializează minimul ca primul element în wrapper, pe când în *min2/2* minimul este inițializat cu ultimul element din lista. Ajunge vreodată execuția pe acest predicat (*min2/2*) la un apel pe []? Ce se întâmplă in acea situație?

2.5 Operații pe mulțimi

Vom reprezenta o mulțime folosind o listă fără elemente duplicate. Reuniunea între 2 mulțimi poate fi realizată prin concatenarea listei a doua cu elementele din prima listă care nu apar în a doua listă. Recurența matematică poate fi formulată astfel:

$$L_1 \cup L_2 = \begin{cases} \{primul(L_1)\} \oplus (coada(L_1) \cup L_2), & primul(L_1) \notin L_2 \\ coada(L_1) \cup L_2, & altfel \end{cases}$$

În Prolog se va scrie:

```
union([], L, L).
union([H|T], L2, R) :- member(H, L2), !, union(T, L2, R).
% Prin folosirea predicatului member și recursivitate, putem verifica fiecare
% element (H) a listei L1 dacă este un element și a L2:
% dacă este -> nu va fi adăugat în rezultatul R
% Altfel, îl adăugăm in R.
union([H|T], L2, [H|R]) :- union(T, L2, R).
```

Urmăriți execuția apelurilor:
?-union([1,2,3], [4,5,6], R).
?-union([1,2,5], [2,3], R).
?-union(L1,[2,3,4],[1,2,3,4,5]).

3 Exerciții

1. Scrieți predicatul *intersect(L1,L2,R)* care realizează intersecția între două mulțimi.

Sugestie: Gândiți-vă la cum a fost implementat predicatul union. Rețineți – vom lua doar elementele care apar în ambele liste.

```
?- intersect([1,2,3], [1,3,4], R).
R = [1, 3] ;
false
```

2. Scrieți predicatul *diff(L1,L2,R)* care realizează diferența între două mulțimi (elementele care apar în prima mulțime și nu apar în a doua mulțime).
   Sugestie: Gândiți-vă la cum a fost implementat predicatul union și *intersect*. Rețineți – vom lua doar elementele care apar în prima listă și nu și in a doua.

```
?- diff([1,2,3], [1,3,4], R).
R = [2] ;
false
```

3. Scrieți predicatele *del_min(L,R)* și *del_max(L,R)* care șterg **toate** aparițiile minimului, respectiva ale maximului din lista *L*.

```
?- del_min([1,3,1,2,1], R).
R = [3, 2] ;
false

?- del_max([3,1,3,2,3], R).
R = [1, 2] ;
false
```

Exercițiu de dificultate ridicată: Încercați să implementați fiecare din aceste predicate folosind o singură traversare a listei, păstrând stabilitatea predicatelor *(del_min1/2* și *del_max1/2).*

4. Scrieți un predicat care inversează ordinea elementelor dintr-o listă începând cu al K-lea element.

```
?- reverse_k([1, 2, 3, 4, 5], 2, R).
R = [1, 2, 5, 4, 3] ;
false
```

5. Scrieți un predicat care codifică șirul de elemente folosind algoritmul RLE (Run-length encoding). Un șir de elemente consecutive și egale se vor înlocui cu perechi **[element, număr de apariții].**

```
?- rle_encode([1, 1, 1, 2, 3, 3, 1, 1], R).
R = [[1, 3], [2, 1], [3, 2], [1, 2]] ;
false
```

Opțional. Puteți modifica acest predicat astfel încât dacă un element este singular (nu are valori consecutive egale), să păstram acel element în loc să adăugam o pereche?

```
?- rle_encode2([1, 1, 1, 2, 3, 3, 1, 1], R).
R = [[1, 3], 2, [3, 2], [1, 2]] ;
false
```

6. Scrieți un predicat care rotește o listă K poziții la dreapta.

   Sugestie: încercați să implementați rotate_left mai întâi, este mai ușor.

```
?- rotate_right([1, 2, 3, 4, 5, 6], 2, R).
R = [5, 6, 1, 2, 3, 4] ;
false
```

7. Scrieți un predicat care extrage aleatoriu K element din lista L și le pune în lista rezultat R.

   Sugestie: folosiți predicatul predefinit *random_between/3(min_value, max_value, result)*.

```
?- rnd_select([a, b, c, d, e, f, g, h], 3, R).
R = [e, d, a] ;
false
```

8. Scrieți un predicat care decodifică șirul de elemente folosind algoritmul RLE (Run-length encoding). O pereche **[element, număr de apariții]** va fi înlocuită de un șir de elemente consecutive și egale.

```
?- rle_decode([[1, 3], [2, 1], [3, 2], [1, 2]], R).
R = [1, 1, 1, 2, 3, 3, 1, 1];
false
```

L5. Metode de sortare

1 Obiective

În această lucrare se prezintă multiple metode de sortare implementate în Prolog.

2 Considerații teoretice

2.1 Sortarea prin generarea permutărilor

Aceasta metodă generează toate permutările posibile ale listei de intrare până când se ajunge la o permutare care are toate elementele ordonate.

```
perm_sort(L,R):- perm(L, R),
                 is_ordered(R),
                 !.

perm(L, [H|R]):- append(A, [H|T], L),
                 append(A, T, L1),
                 perm(L1, R).
perm([], []).

is_ordered([H1, H2|T]):- H1 =< H2,
                         is_ordered([H2|T]).
is_ordered([_]). % dacă este doar un element, lista este deja ordonată
```

Predicatul generează o permutare a listei L prin apelul predicatului *perm/2*, după care verifică dacă este ordonată (*is_ordered/1*). Dacă R nu este ordonată, *is_ordered(R)* va eșua. Execuția va face backtracking la *perm(L,R)* și va rezulta într-o permutare nouă. Acest proces continuă până când permutarea generată este ordonată. În mod evident, această abordare este foarte ineficientă din punct de vedere algoritmic și a fost inclusă în acest laborator datorită logicii si a modalităților de generare a permutărilor.

Numărul de permutări ale unei liste cu *n* elemente este egal cu *n!*. Pentru a genera permutările, vom alege un element aleatoriu *H* din lista de intrare și îl vom pune pe prima poziție în lista rezultat. Alegerea lui *H* se va realiza folosind non-determinismul predicatului *append/3.* Lista de intrare poate fi scrisă ca o

concatenare de 2 sub-liste. Elementul *H* îl alegem să fie primul element din a doua sub-listă (De ce?).

Dacă am lua exemplul listei [1,2,3] și folosind predicatul *append* o separăm în două sub-liste:

```
?- append(L1, L2, [1,2,3]).
L1 = []
L2 = [1,2,3] ;
L1 = [1]
L2 = [2,3] ;
% ... și tot așa
```

Folosind șablonul *[H|T]* putem separa a doua listă:

```
?- append(L1, [H|T], [1,2,3]).
L1 = []
H = 1          T = [2,3] ;
L1 = [1]
H = 2          T = [3] ;
% ...
```

Permutările obținute prin concatenarea dintre L1 și T, și ulterior adăugarea primului element H la începutul listei rezultat:

De exemplu:

```
L1 = []        H = 1          T = [2,3]
```

Prin append-ul listei L1 și T se obține [2,3] iar prin concatenarea lui H la început, se obține [1,2,3], o primă permutare.

```
L1 = [1]       H = 2          T = [3] ;
```

Prin append-ul listei L1 și T se obține [1,3] iar prin concatenarea lui H la început, se obține [2,1,3].

Observație. Predicatul *perm/2* folosește backtracking pentru a genera toate permutările. Acest exemplu este folosit pentru a exemplifica logica de formare a unei permutări prin concatenări, nu arată logica a cum generează *perm/2* permutările într-o anumită ordine prin backtracking. Recomandăm urmărirea execuției acestui predicat.

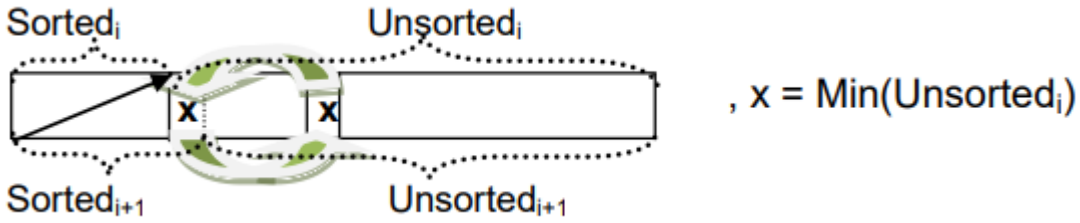
Mai avem de scris predicatul care verifică dacă lista rezultat este ordonată. Alegem ca lista rezultat să fie ordonată crescător.

Observație. Predicatul *is_ordered/1* are un singur argument, lista de intrare care vrem să o verificăm dacă este ordonată. Rezultatele posibile sunt *da* sau *nu*.

Urmăriți execuția apelurilor:
?- append(A, [H|T], [1, 2, 3]), append(A, T, R).
?- perm([1, 2, 3], L).
?- is_ordered([1, 2, 4, 4, 5]).
?- perm_sort([1, 4, 2, 3, 5], R).

2.2 Sortarea prin selecție

Această sortare alege la fiecare pas cel mai mic element din lista nesortată și îl mută la finalul liste deja sortate, după care continuă pe listă fără acel element. Cel mai mic element din lista de intrarea reprezintă primul element din lista rezultat.



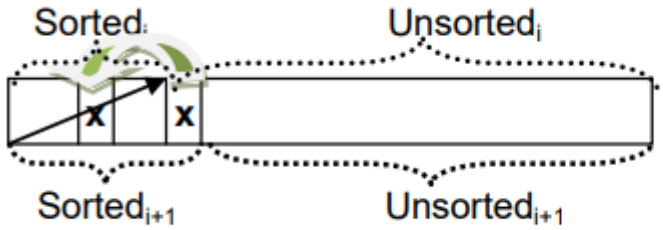

```
sel_sort(L, [M|R]):- min1(L, M),
                     delete1(M, L, L1),
                     sel_sort(L1, R).
sel_sort([], []).

delete1(X, [X|T], T) :- !.
delete1(X, [H|T], [H|R]) :- delete1(X, T, R).
delete1(_, [], []).

min1([H|T], M) :- min1(T, M), M<H, !.
min1([H|_], H).
```

Predicatele *min1/2* și *delete1/3* se pot găsi în lucrările precedente. Această versiunea a sortării prin selecție construiește soluția la întoarcerea din apelul recursiv. Prin urmare, variabila care conține rezultatul reține partea sortată a listei și lista de intrare conține partea nesortată, care se schimbă cu fiecare apel recursiv. Partea sortată este inițializată cu [] când recursivitatea se oprește (clauza 2), și crește la fiecare apel care se întoarce din recursivitate prin adăugarea minimului curent la începutul listei.

Urmăriți execuția apelurilor:
?- sel_sort([3, 2, 4, 1], R).
?- sel_sort([3, 1, 5, 2, 4, 3], R).

2.3 Sortarea prin inserare

Această metodă de sortare extrage la fiecare pas un element din lista nesortată (de obicei primul element) și îl inserează în poziția corectă în lista sortată (în Prolog, folosind căutare liniară).



, x = First(Unsorted$_i$)

```
ins_sort([H|T], R):- ins_sort(T, R1),
                     insert_ord(H, R1, R).
ins_sort([], []).

insert_ord(X, [H|T], [H|R]):- X>H,
                              !,
                              insert_ord(X, T, R).
insert_ord(X, T, [X|T]).
```

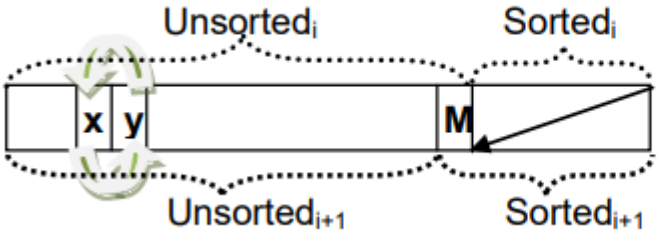
Această implementare utilizează o abordare bazată pe recursivitate înapoi: rezultatul apelurilor recursive (R1) este obținut primul, după care se obține rezultatul nivelului curent (*predicatul insert_ord/3*) prin inserarea elementului curent la poziția corectă în R1.

Urmăriți execuția apelurilor:

```
?- insert_ord(3, [], R).
?- insert_ord(3, [1, 2, 4, 5], R).
?- insert_ord(3, [1, 3, 3, 4], R).
?- ins_sort([3, 2, 4, 1], R).
```

2.4 Sortarea prin interschimbare

Sortarea prin interschimbare (metoda bulelor) este una dintre cele mai simple metode directe de sortare, dar și cea mai ineficientă (exceptând sortarea prin generarea permutărilor). Metoda realizează mai multe treceri prin lista de intrare. La fiecare trecere, verifică două câte două elemente adiacente și le interschimbă dacă nu respectă condiția de ordine. Astfel, fiecare trecere garantează că elementul maxim al părții nesortate ajunge la începutul părții anterior sortate (la pasul precedent). Astfel, la fiecare trecere, coada listei este sortată.



, M = Max(Unsorted$_i$)

```
bubble_sort(L,R):- one_pass(L,R1,F),
                   nonvar(F),
                   !,
                   bubble_sort(R1,R).
bubble_sort(L,L).

one_pass([H1,H2|T], [H2|R], F):- H1>H2,
                                 !,
                                 F=1,
                                 one_pass([H1|T],R,F).
one_pass([H1|T], [H1|R], F):- one_pass(T, R, F).
one_pass([], [] ,_).
```

Dacă la o trecere nu s-a făcut nici o interschimbare, atunci sortarea s-a terminat. O versiunea îmbunătățită a sortării bulelor se folosește de acest fapt și oprește algoritmul în acest caz. Pentru a verifica dacă s-a făcut o interschimbare ne vom folosi de un *flag (F).* Când se produce o interschimbare, *F* este inițializat cu o valoare constantă (1). La fiecare trecere, vom verifica acest flag:

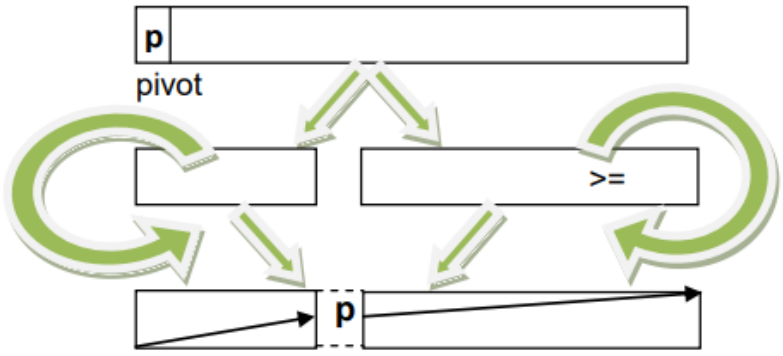
- Dacă *F* a fost inițializat (nu a rămas o variabilă liberă), atunci cel puțin o interschimbare a fost făcută, ceea ce înseamnă că lista s-ar putea să nu fie ordonată. Deci, un nou apel al *bubble_sort/2* este necesar.
- Dacă *F* a rămas o variabilă liberă după apelul *one_pass/3*, atunci apelul către *nonvar(F)* va da fail. Ceea ce înseamnă că lista L este sortată și va fi pasată către rezultat (clauza 2 a predicatului *bubble_sort/2*)

Urmăriți execuția apelurilor:

```
?- one_pass([1, 2, 3, 4], R, F).
?- one_pass([2, 3, 1, 4], R, F).
?- bubble_sort([1, 2, 3, 4], R).
?- bubble_sort([2, 3, 1, 4], R).
```

2.5 Quicksort

Sortarea rapidă (*Quick sort*) se folosește de un pivot care împarte lista de intrare în două sub-liste (tehnică *divide et impera*). O sub-listă cu elementele mai mici decât pivotul și o a doua sub-listă cu elementele mai mari decât pivotul. În cazul implementării din laborator folosim ca și pivot primul element. Repetăm acest proces până când sub-listele devin vide. Rezultatul este compus prin concatenarea *sub-listei cu elemente mai mici* cu *pivotul* și cu *sub-lista elementelor mai mari* (append-ul din prima clauză a predicatului *quick_sort*/2).




```
quick_sort([H|T], R):- % alegem pivot ca primul element
    partition(H, T, Sm, Lg),
    % sortăm sublista cu elemente mai mici decât pivotul
    quick_sort(Sm, SmS),
    % sortăm sublista cu elemente mai mari decât pivotul
    quick_sort(Lg, LgS),
    append(SmS, [H|LgS], R).
quick_sort([], []).

partition(P, [X|T], [X|Sm], Lg):-
     X<P,
     !,
     partition(P, T, Sm, Lg).
partition(P, [X|T], Sm, [X|Lg]):-
     partition(P, T, Sm, Lg).
partition(_, [], [], []).
```

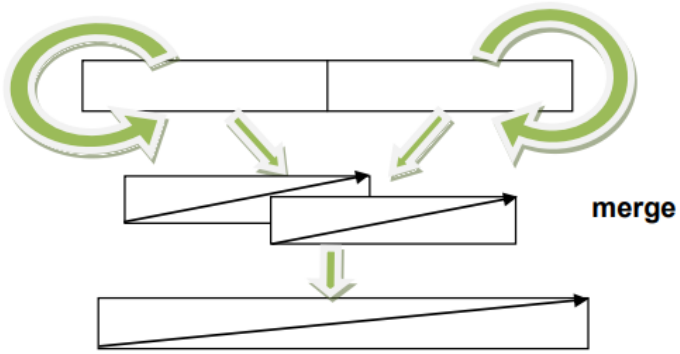
Predicatul *partition/4* are ca pivot primul argument și lista de intrare ca al doilea argument, când elementul curent (*X*) al listei este mai mic decât pivotul (*P*) este adăugat (clauza 1) la lista elementelor mai mici (al treilea argument), altfel este adăugat (clauza 2) la lista elementelor mai mari (al patrulea argument). Acest proces recursiv se oprește când lista de intrare (al doilea argument) devine vidă.

Urmăriți execuția apelurilor:

```
?- partition(3, [4, 2, 6, 1, 3], Sm, Lg).
?- quick_sort([3, 2, 5, 1, 4, 3], R).
?- quick_sort([1, 2, 3, 4], R).
```

2.6 Sortare prin interclasare

Sortarea prin interclasare împarte lista de intrare în doua sub-liste de lungimi egale (tehnica *divide et impera*). Apoi sortează cele două sub-liste și în final interclasează sub-listele deja sortate.




```
merge_sort(L, R):-
    split(L, L1, L2), % împarte L în doua subliste de lungimi egale
    merge_sort(L1, R1),
    merge_sort(L2, R2),
    merge(R1, R2, R). % interclasează sublistele ordonate
% split returnează fail dacă lista ii vidă sau are doar un singur element
merge_sort([H], [H]).
merge_sort([], []).

split(L, L1, L2):-
    length(L, Len),
    Len>1,
    K is Len/2,
    splitK(L, K, L1, L2).

splitK([H|T], K, [H|L1], L2):-
    K>0,
    !,
    K1 is K-1,
    splitK(T, K1, L1, L2).
splitK(T, _, [], T).
```

```
merge([H1|T1], [H2|T2], [H1|R]):-
    H1<H2,
    !,
    merge(T1, [H2|T2], R).
merge([H1|T1], [H2|T2], [H2|R]):-
    merge([H1|T1], T2, R).
merge([], L, L).
merge(L, [], L).
```

Predicatul *splitK/4* ia primele **K** elemente din lista de intrare **L** și le inserează în lista L1 (clauza 1), pe când restul elementelor sunt inserate în lista L2 (clauza 2). Predicatul *split* separă lista în două părți egale prin apelarea predicatului *splitK/4*, unde K este jumătate din lungimea listei (*K is Len/2*). Dacă lungimea listei de intrare este 0 sau 1, atunci apelul predicatului *split* va da fail, cauzând astfel rezoluția prin clauza 1 a *merge_sort/2* să dea fail – în acest punct, recursivitatea ar trebui să se oprească. Prin urmare, aceste apeluri vor fi unificate cu clauza 2 sau 3 a predicatului *merge_sort/2*. Predicatul *merge/3* execută unirea a două liste ordonate, prin adăugarea primului element din prima sau a doua listă la lista rezultată în funcție de care este mai mic.

Urmăriți execuția apelurilor:

```
?- split([2, 5, 1, 6, 8, 3], L1, L2).
?- split([2], L1, L2).
?- merge([1, 5, 7], [3, 6, 9], R).
?- merge([1, 1, 2], [1], R).
?- merge([], [3], R).
?- merge_sort([4, 2, 6, 1, 5], R).
```

3 Exerciții

1. Rescrieți predicatul *sel_sort/2* astfel încât să selecționeze cea mai mare valoare din partea nesortată, prin urmare să sorteze descrescător, folosind denumirea *sel_sort_max/2*. Predicatul *max1/2* poate fi creat prin modificarea predicatului *min1/2* din laboratorul anterior.

```
?- sel_sort_max([3,4,1,2,5], R).
R = [5, 4, 3, 2, 1];
false
```

2. Rescrieți predicatul *ins_sort* utilizând recursivitate *forward*, folosind denumirea *ins_sort_fwd/2*.

Recomandare: predicatul de insertion sort este format din două predicate, ambele folosind o abordare *backward*, încercați să le modificați pe ambele într-o abordare *forward*.

```
?- ins_sort_fwd([3,4,1,2,5], R).
R = [1, 2, 3, 4, 5];
false
```

3. Implementați *bubble_sort* cu un număr fix de treceri prin lista de intrare, folosind denumirea *bubble_sort_fixed/3*.

```
% bubble_sort_fixed(L, K, R). – K este numărul de treceri, egal cu n-1
?- bubble_sort_fixed([3,5,4,1,2], 4, R).
R = [1, 2, 3, 4, 5]
```

4. Scrieți un predicat care să sorteze o listă de caractere ASCII. (Puteți folosi o metodă de sortare la alegere).
   Sugestie: folosiți predicatul predefinit *char_code/2*

```
?- sort_chars([e, t, a, v, f], L).
L = [a, e, f, t, v] ;
false
```

5. Scrieți un predicat care să sorteze o lista de sub-liste în funcție de lungimea sub-listelor.

```
?- sort_lens([[a, b, c], [f], [2, 3, 1, 2], [], [4, 4]], R).
R = [[], [f], [4, 4], [a, b, c], [2, 3, 1, 2]] ;
false
```

Opțional. Complexitatea problemei poate crește dacă cerem sortarea după valori în cazul sub-listelor de lungimi egale, e.g. [1,1,1] și [1,1,2], predicatul ar trebui să analizeze suplimentar cele două liste în cazul de lungimi egale și să le compare element cu element.

```
?- sort_lens2([[], [1], [2, 3, 1, 2], [2, 3, 5, 2], [7,6,8], [4, 4]], R).
R = [[], [1], [4, 4], [7, 6, 8], [2, 3, 1, 2], [2, 3, 5, 2]];
false
```

6. Rescrieți predicatul *perm/2* fără a apela predicatul *append/3*, folosind denumirea *perm1/2*. Extragerea și ștergerea unui element trebuie realizate altfel.

?- perm1([1,2,3], R).
R = [1, 2, 3];
R = [1, 3, 2];
R = [2, 1, 3];
…

L6. Liste adânci

1 Obiective

În lucrarea de față vom prezenta o generalizare a tipului listă, într-o listă ale cărei elemente pot fi fie atomice, fie liste. Astfel, o listă adâncă poate să aibă elemente atomice la diverse nivele de imbricare.

Exemple:

```
L1 = [1,2,3,[4]]
L2 = [[1],[2],[3],[4,5]]
L3 = [[],2,3,4,[5,[6]],[7]]
L4 = [[[[1]]],1,[1]]
L5 = [1,[2],[[3]],[[[4]]],[5,[6,[7,[8,[9],10],11],12],13]]
L6 = [alpha,2,[beta],[gamma,[8]]]
```

O listă adâncă în Prolog este reprezentată de o structură recursivă, unde multiple liste de adâncimi diferite sunt imbricate una în cealaltă. În mod formal, o listă adâncă poate fi definită ca:

$$DL = [H|T] \; unde \; H \in \{atom, listă, listă \; adâncă\} \; și$$

$$T \in \{ \; listă, listă \; adâncă\}$$

Rețineți. O listă regulată este un caz special de listă adâncă.

Toate operațiile definite pe liste simple pot fi redefinite și pe liste adânci, inclusiv (dar nu numai) predicatele *member/2*, *append/3* și *delete/3*. Pentru a înțelege modul de funcționare a acestor operații pe liste adânci, trebuie să considerăm o listă adâncă echivalentă cu o listă simplă cu elemente de un tip diferit, dar doar cele de pe primul nivel.

Testați următoarele întrebări:

```
?- member(2,L5).
?- member([2], L5).
?- member(X, L5).
?- append(L1,R,L2).
?- append(L4,L5,R).
?- delete(1, L4,R).
?- delete(13,L5,R).
```

2 Considerații teoretice

2.1 Predicatul „atomic”

Pentru a executa diferite operații pe toate elementele listei adânci, trebuie să știm cum să lucrăm cu aceste elemente în funcție de tipul lor (dacă sunt atomi, le procesăm, dacă sunt liste, trebuie să continuăm cu descompunerea pentru a le procesa).

Vom folosi predicatul predefinit *atomic(X)* pentru a verifica dacă X este un element simplu (număr, simbol) sau este o structură complexă (ex: o altă listă).

Testați următoarele întrebări:

```
? – atomic(apple).
? – atomic(4).
? – atomic(X).
? – atomic(apple(2)).
? – atomic([1,2,3]).
? – atomic([]).
```

2.2 Predicatul „depth”

Acest predicat numără nivelul maxim de imbricare a unei liste adânci. Adâncimea unui atom este definită ca fiind 0, iar adâncimea unei liste simple (incluzând lista vidă) este 1.

Predicatul parcurge elementele listei și verifică natura lor (folosind *atomic/1*). Dacă elementul este atomic, apelează predicatul pe coada listei (clauza 2), iar dacă elementul nu este atomic se afla atât adâncimea lui, cât și a restului listei; rezultatul este maximul dintre adâncimea lui H crescută cu 1 (de ce?), și adâncimea lui T.

```
max(A, B, A):- A>B, !.
max(_, B, B).

depth([],1).
depth([H|T],R):- atomic(H),
                 !,
                 depth(T,R).
depth([H|T],R):- depth(H,R1),
                 depth(T,R2),
                 R3 is R1+1,
                 max(R3,R2,R).
```

Când calculăm adâncimea maximă vom trata trei cazuri:

1. Ajungem la lista vidă. Adâncimea este 1. **(clauza 1)**
2. Avem un prim element atomic, îl ignorăm deoarece nu influențează adâncimea. Adâncimea unei listei este egală cu adâncimea cozii. Prin urmare, trebuie să apelăm recursiv pe coadă pentru a continua parcurgerea listei. **(clauza 2)**
3. Avem o listă ca primul element al listei adânci. În acest caz, adâncimea listei va fi fie adâncimea cozii, fie adâncimea primei liste incrementată cu unu (*De ce?).* **(clauza 3)**

Testați predicatul pentru listele L1-L6 de mai sus, de exemplu:

```
?- depth(L1, R).
```

2.3 Aplatizarea unei liste adânci – predicatul „flatten”

Acest predicat aplatizează o listă adâncă. Lista rezultat va conține numai elemente atomice (lista simplă echivalentă listei adânci conținând aceleași elemente atomice, toate la nivel 1 de imbricare). Și în acest caz trebuie să se verifice natura fiecărui elementul. Dacă elementul este la rândul lui o listă, atunci va trebui aplatizată.

Această operație reprezintă obținerea unei liste simple. Pentru a face acest lucru, vom lua doar elementele atomice din lista de intrare și le vom adăuga la rezultat.

```
flatten([],[]).
flatten([H|T], [H|R]):- atomic(H), !, flatten(T,R).
flatten([H|T], R):- flatten(H,R1), flatten(T,R2), append(R1,R2,R).
```

Vom avea 3 cazuri principale:

- Aplatizarea unei liste vide rezultă într-o listă vidă. **(clauza 1)**
- Dacă primul element al listei este atomic, atunci îl adăugăm la rezultat și procesăm restul listei. **(clauza 2)**
- Dacă primul element nu este atomic, rezultat trebuie compus ca toate elementele atomice ale primului element și toate elementele atomice ale cozii. (*Cum colectăm cele două rezultatele într-o singura listă?)* **(clauza 3)**

Testați predicatul pentru listele L1-L6, de exemplu:

```
?- flatten(L1, R).
```

2.4 Elementele atomice care încep o listă – predicatul „heads”

Acest predicat extrage toate elementele atomice de la începutul fiecărei liste.

% Varianta 1

Ne vom folosi de un predicat auxiliar, care trece peste elementele atomice ale unei liste. Vom lua doar primul element din fiecare listă, apoi vom folosi predicatul auxiliar pentru a trece peste restul elementelor atomice.

```
skip([],[]).
skip([H|T],R):-
    atomic(H),
    !,
    skip(T,R).
skip([H|T],[H|R]):-
    skip(T,R).

% luăm primul element din fiecare listă,
% după trecem peste restul elementelor
% prin apelul predicatului skip, și apelăm recursiv pentru liste
heads1([],[]).
heads1([H|T],[H|R]):- atomic(H),
                      !,
                      skip(T,T1),
                      heads1(T1,R).
heads1([H|T],R):- heads1(H,R1),
                  heads1(T,R2),
                  append(R1,R2,R).
```

% Varianta 2

Ne vom folosi de un al treilea parametru (*flag*) pentru a ști dacă am extras capul listei curente sau nu.

```
heads2([],[],_).
% dacă flag=1 atunci suntem la început de lista și putem extrage capul
listei; în apelul recursiv setam flag=0
heads2([H|T],[H|R],1):-
    atomic(H),
    !,
    heads2(T,R,0).
```

```
% dacă flag=0 atunci nu suntem la primul element atomic și
% atunci continuam cu restul elementelor
heads2([H|T],R,0):-
    atomic(H),
    !,
    heads2(T,R,0).
% dacă am ajuns la aceasta clauza înseamnă că primul element nu este
atomic și atunci trebuie să apelam recursiv și pe acest element
heads2([H|T],R,_):-
    heads2(H,R1,1),
    heads2(T,R2,0),
    append(R1,R2,R).
% un wrapper pentru predicatul heads/3
heads2(L,R):- heads2(L, R, 1).
```

Testați predicatul pentru listele L1-L6, de exemplu:

?- heads1(L1, R).

?- heads2(L1, R).

2.5 Predicatul „member”

Putem folosi predicate pe care le-am învățat deja pentru o implementare simplă a predicatului *member/2* pentru liste adânci în felul următor:

```
% Varianta 1
member1(X, L):- flatten(L,L1), member(X,L1).
```

Observație: Această versiune poate găsi doar elemente atomice. Verificați a doua variantă a predicatului membru pentru o soluție mai generală.

Dacă dorim să extindem predicatul *member/2* implementat în sesiuni anterioare de laborator pentru a funcționa și pe liste adânci, trebuie să modificăm predicatul *member/2* astfel:

```
% Varianta 2
member2(H, [H|_]).
member2(X, [H|_]):- member1(X,H).
member2(X, [_|T]):- member1(X,T).
```

Predicatul *member2/2* funcționează similar cu predicatul *member/2* pentru liste simple, considerăm membru al listei toate elementele care apar în listă, atomice sau nu, la orice nivel.

Trasați următoarele întrebări:

```
?- member2(1,L1).
?- member2(4,L2).
?- member2([5,[6]], L3).
?- member2(X,L4).
?- member2(X,L6).
?- member2(14,L5).
```

3 Exerciții

1. Scrieți predicatul *count_atomic(L,R)* care calculează numărul de elemente atomice din lista *L* (*toate elementele atomice de la toate adâncimile*).

```
?- count_atomic([1,[2],[[3]],[[[4]]],[5,[6,[7,[8,[9],10],11],12],13]], R).
R = 13;
false.
```

2. Scrieți predicatul *sum_atomic(L,R)* care calculează suma elementelor atomice din lista *L* (*toate elementele atomice de la toate adâncimile*).

```
?- sum_atomic([1,[2],[[3]],[[[4]]]], R).
R = 10;
false.
```

3. Scrieți predicatul *replace(X,Y,L,R)* care înlocuiește pe *X* cu *Y* în lista adâncă *L* (la orice nivel de imbricare) și pune rezultatul în *R.*

```
?- replace(1, a, [[[[1,2], 3, 1], 4],1,2,[1,7,[[1]]]], R).
R = [[[[a, 2], 3, a], 4], a, 2, [a, 7, [[a]]]];
false.
```

4. Scrieți predicatul *lasts(L,R)* care extrage elementele atomice de pe ultima poziție (imediat anterior *']'*) din fiecare sublistă din *L*.

```
?- lasts([1,2,[3],[4,5],[6,[7,[9,10],8]]], R).
R = [3,5,10,8] ;
false.
```

5. Înlocuiți fiecare secvență constantă de o anumită adâncime cu lungimea ei (**fără** să utilizați predicatul *length/2*).

```
? -  len_con_depth([[1,2,3],[2],[2,[2,3,1],5],3,1],R).
```

R = [[3],[1],[1,[3],1],2].

6. Înlocuiți fiecare secvență constantă de o anumită adâncime cu adâncimea ei (**fără** să utilizați predicatul *depth/2*).

? - depth_con_depth([[1,2,3],[2],[2,[2,3,1],5],3,1],R).
R = [[1], [1], [1, [2], 1], 0].

7. Modificați predicatul *member2/2* pentru liste adânci astfel încât să fie determinist (în acest caz, deterministic se referă la faptul că predicatul va returna un singur răspuns – nu va exista opțiunea de *next* – este nevoie de multiple teste ale predicatului *member_deterministic/2* cu input-uri variate pentru a verifica).

?- member_deterministic(1, [1,2,1]).
true.

?- member_deterministic(1, [[1],2,1]).
true.

8. Scrieți un predicat care să sorteze o listă adâncă în ordine crescătoare a adâncimilor listelor. Dacă două elemente au aceeași adâncime, se vor compara în funcție de elementele atomice pe care le conțin.

   Sugestie:

   - L1 < L2, dacă L1 și L2 sunt liste, sau liste adânci, și dacă adâncimea listei L1 este mai mică decât adâncimea listei L2.
   - L1 < L2, dacă L1 și L2 sunt liste, sau liste adânci cu adâncime egală, toate elementele până la poziția *k* sunt egale, iar al *k+1-lea* element din lista L1 este mai mic decât al *k+1-lea* element din lista L (la adâncimi egale, lista cu indexul mai mic sublistei care va da ultima adâncime este considerat mai mare – ca în exemplu la comparația dintre 5 și [5])

?- sort_depth([[[[1]]], 2, [5,[4],7], [[5],4], [5,[0,9]]], R).
R = [2, [5,[0,9]], [[5],4], [5,[4],7], [[[1]]]] ;
false

L7. Arbori

1 Obiective

În lucrarea *L7. Arbori* vom introduce operațiile pe Arbori Binari de Căutare (ABC) și modalitatea de a reprezenta arbori ternari, și algoritmi bazați pe parcurgerea lor. În Prolog, arborii sunt reprezentați prin structuri recursive. Arborele gol îl reprezentăm prin simbolul *nil*.

2 Considerații teoretice

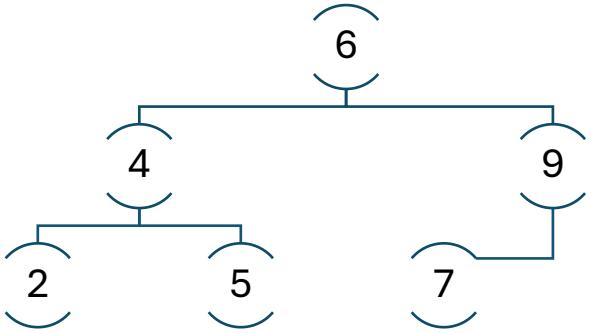
2.1 Reprezentare

Arborele Binar de Căutare (ABC) este reprezentat printr-o structură recurentă cu 3 argumente:

cheia nodului curent,
subarborele stâng (structură de același tip) și
subarborele drept (structură de același tip).

Exemplu:

t(6, t(4, t(2,nil,nil), t(5,nil,nil)), t(9, t(7,nil,nil), nil))



Pentru a nu scrie de fiecare dată arborele în interpretorul Prolog, puteți „salva” structura în fișier într-un fapt (axiomă).

Exemple:

```
tree1(t(6, t(4,t(2,nil,nil),t(5,nil,nil)), t(9,t(7,nil,nil),nil))).
tree2(t(8, t(5, nil, t(7, nil, nil)), t(9, nil, t(11, nil, nil)))).
...
% prin această întrebare, variabila T se unifică cu arborele din faptul tree1/1
?- tree1(T), operatie_pe_arbore(T, ...).
```

2.2 Parcurgerea arborelui

Există 3 posibilități de a parcurge arborele în funcție de ordinea în care sunt procesate nodurile: în-ordine, pre-ordine și post-ordine. De exemplu, traversarea în inordine colectează nodurile din subarborele stâng, apoi nodul curent și în final nodurile din subarborele drept.

```
% subarbore stâng, cheie și subarbore drept (ordinea din append)
inorder(t(K,L,R), List):-
    inorder(L,LL),
    inorder(R,LR),
    append(LL, [K|LR], List).
inorder(nil, []).

% cheie, subarbore stâng și subarbore drept (ordinea din append)
preorder(t(K,L,R), List):-
    preorder(L,LL),
    preorder(R, LR),
    append([K|LL], LR, List).
preorder(nil, []).

% subarbore stâng, subarbore drept și apoi cheia (ordinea din append-uri)
postorder(t(K,L,R), List):-
    postorder(L,LL),
    postorder(R, LR),
    append(LL, LR,R1),
    append(R1, [K], List).
postorder(nil, []).
```

Observație1. Cu toate că apelul recursiv al subarborelui drept este făcut înainte de procesarea nodului rădăcină, ordinea corectă este păstrată la construirea listei de ieșire în apelul către predicatul *append/3.* Astfel, în cazul predicatului *inorder/2*, nodurile din subarborele stâng apar primele, urmate de cheia rădăcinii, iar la final nodurile din subarborele drept.
Observație2. Observați că doar poziția nodului curent se schimbă, prin urmare singura diferență între aceste predicate este unde concatenăm nodul curent (K).

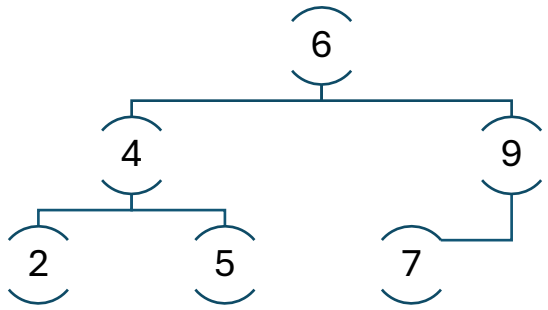
Urmărește execuția apelurilor:

```
?- tree1(T), inorder(T, L).
?- tree1(T), preorder(T, L).
?- tree1(T), postorder(T, L).
```

2.3 Afișare „pretty”

Ne vom folosi de afișarea „pretty” pentru a vizualiza corectitudinea operațiilor pe arbori. Cea mai ușoară variantă de a afișa un arbore este de a-l parcurge în inordine și de a afișa fiecare nod la un număr de spații egal cu adâncimea sa în arbore. Rădăcina se află la adâncime 0 și fiecare nod este afișat pe un alt rând.

Exemplu:




```
|       9
|               7
|6
|               5
|       4
|               2
```

Dacă studiem afișarea de mai sus putem observa că cheile din dreapta sunt afișate primele, urmate de rădăcină, iar apoi de cheile din stânga. Acest fapt ne sugerează că traversarea în inordine este potrivită pentru un astfel de afișaj. Prin urmare, predicatul care afișează este:

```
% wrapper
pretty_print(T):- pretty_print(T, 0).

% predicatul care afișează arborele
pretty_print(nil, _).
pretty_print(t(K,L,R), D):-
    D1 is D+1,
    pretty_print(R, D1),
    print_key(K, D),
    pretty_print(L, D1).

% predicat care afișează cheia K la D tab-uri față de marginea din stânga
% și inserează o linie nouă (prin nl)
```

```
print_key(K, D):-
    D>0,
    !,
    D1 is D-1,
    tab(8),
    print_key(K, D1).
print_key(K, _):-
    write(K),
    nl.
```

Urmărește execuția apelului:
?- tree2(T), pretty_print(T).

2.4 Căutare unei chei

Predicatul *search_key(Key, T)* verifică dacă există un nod cu cheia *Key* în arborele *T*. Recurența matematică poate fi formulată astfel:

$$search(key, T) = \begin{cases} false & T = null \\ true & key = T.root.key \\ search(key, T.left) & key < T.root.key \\ search(key, T.right) & key > T.root.key \end{cases}$$

Algoritmul poate fi descris în pseudocod ca:

```
if currentNode = null then return -1; //nu a fost găsit
if searchKey = currentNode.key then return 0; //găsit (clauza 1)
else  if searchKey < currentNode.key (clauza 2)
           //căutam in subarborele stâng
           then search(searchKey, currentNode.left);
      else (clauza 3)
           // căutam in subarborele drept
           search(searchKey, currentNode.right);
```

Acest pseudocod este ușor de transformat în specificații Prolog. Deoarece vrem ca predicatul să eșueze în cazul în care cheia nu este găsită, putem să specificăm acest fapt explicit printr-un *fail* când se ajunge la un *nil*, sau implicit, prin nerezolvarea cazului în care se ajunge la *nil*.

În Prolog se va scrie:

```
search_key(Key, t(Key, _, _)):- !.
search_key(Key, t(K, L, _)):- Key<K, !, search_key(Key, L).
search_key(Key, t(_, _, R)):- search_key(Key, R).
```

Urmărește execuția apelurilor:
?- tree1(T), search_key(5,T).
?- tree1(T), search_key(8,T).

2.5 Inserarea unei chei

Pentru inserarea unei chei într-un ABC este necesară căutarea poziției în arbore unde poate fi inserată cheia (astfel încât nodul adăugat sa fie frunză). Dacă se găsește un nod cu aceeași cheie, atunci nu se realizează inserarea. Când se ajunge la un *nil* în procesul de căutare, putem crea nodul nou.

```
insert_key(Key, nil, t(Key, nil, nil)). % inserează cheia în arbore
insert_key(Key, t(Key, L, R), t(Key, L, R)):- !. % cheia există deja
insert_key(Key, t(K,L,R), t(K,NL,R)):- Key<K,!,insert_key(Key,L,NL).
insert_key(Key, t(K,L,R), t(K,L,NR)):- insert_key(Key, R, NR).
```

Urmărește execuția apelurilor:

```
?- tree1(T),pretty_print(T),insert_key(8,T,T1),pretty_print(T1).
?- tree1(T),pretty_print(T),insert_key(5,T,T1),pretty_print(T1).
?- insert_key(7, nil, T1), insert_key(12, T1, T2), insert_key(6, T2, T3), insert_key(9, T3, T4), insert_key(3, T4, T5), insert_key(8, T5, T6), insert_key(3, T6, T7), pretty_print(T7).
```

2.6 Ștergerea unei chei

În cazul ștergerii unei chei trebuie să căutăm nodul care are acea cheie și apoi trebuie să ne asiguram că în urma ștergerii nodului respectiv, arborele va continua să fie arbore binar de căutare.

După găsirea cheii, ne vom afla in una din următoarele trei situații:

- Trebuie să ștergem un nod frunză (**clauza 1 și 2**, când L sau R = *nil*)
- Trebuie să ștergem un nod cu un copil **(clauza 1 și 2)**
- Trebuie să ștergem un nod cu doi copii **(clauza 3)**

Primele două cazuri sunt destul de simple. Pentru al treilea caz avem două alternative: sau înlocuim nodul care urmează să fie șters cu predecesorul (sau succesorul) – prin reconstruirea legăturilor, sau să mutăm subarborele stâng în partea stângă a subarborele drept (sau vice-versa).

Prima alternativă este implementată în predicatul *delete_key/3* de mai jos:

```
delete_key(Key, t(Key, L, nil), L):- !.
delete_key(Key, t(Key, nil, R), R):- !.
delete_key(Key, t(Key, L, R), t(Pred,NL,R)):- !, get_pred(L,Pred,NL).
delete_key(Key, t(K,L,R), t(K,NL,R)):- Key<K, !, delete_key(Key,L,NL).
delete_key(Key, t(K,L,R), t(K,L,NR)):- delete_key(Key,R,NR).
```

```
% caută nodul predecesor
get_pred(t(Pred, L, nil), Pred, L):- !.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):- get_pred(R, Pred, NR).
```

Urmărește execuția apelurilor:
?- tree1(T), pretty_print(T), delete_key(5, T, T1), pretty_print(T1).
?- tree1(T), pretty_print(T), delete_key(9, T, T1), pretty_print(T1).
?- tree1(T), pretty_print(T), delete_key(6, T, T1), pretty_print(T1).
?- tree1(T), pretty_print(T), insert_key(8, T, T1), pretty_print(T1), delete_key(6, T1, T2), pretty_print(T2), insert_key(6, T2, T3), pretty_print(T3).

2.7 Înălțimea unui arbore

Predicatul *height(T,R)* calculează distanța maximă între rădăcina arborelui și o frunză. Recurența matematică poate fi formulată astfel:

$$height(T) = \begin{cases} 0 & T = null \\ \max\big(height(T.left), height(T.right)\big) + 1 & altfel \end{cases},$$

adică:

- înălțimea unui nod *nil* este 0
- înălțimea unui nod diferit de *nil* este maximul dintre înălțimea subarborelui stâng și înălțimea subarborelui drept, plus 1 (max{T.Left, T.Right} + 1)

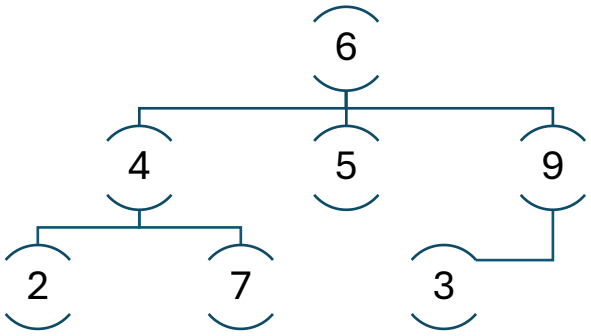
În Prolog se va scrie:

```
height(nil, 0).
height(t(_, L, R), H):-
    height(L, H1),
    height(R, H2),
    max(H1, H2, H3),
    H is H3+1.

max(A, B, A):-A>B, !.
max(_, B, B).
```

Urmărește execuția apelurilor:
?- tree1(T), pretty_print(T), height(T, H).
?- tree1(T), height(T, H), pretty_print(T), insert_key(8, T, T1), height(T1, H1), pretty_print(T1).

2.8 Arbori ternari

Într-un arbore ternar un nod poate să aibă maxim 3 copii. Cu toate că nu este la fel de ușor ca în cazul arborilor binari, relațiile de ordine pot fi stabilite și pentru arborii ternari. Arborele îl putem reprezenta folosind o structură de tipul *t(Key, Left, Middle, Right)*.




```
Example:
ternary_tree(t(6, t(4, t(2, nil, nil, nil), nil, t(7, nil, nil, nil)), t(5, nil, nil, nil), t(9, t(3, nil, nil, nil), nil, nil) ) ).
```

2.8.1 Pretty print pentru arbori ternari

Deoarece un nod într-un arbore ternar poate avea până la 3 copii, trebuie să folosim o strategie diferită pentru afișarea unei astfel de structuri decât cea folosită pentru arbori binari. O soluție ar fi să printăm fiecare nod la o adâncime de indentare de la marginea stângă (ca înainte); mai mult de atât, subarborele unui nod trebuie să apară sub nodul respectiv (ar trebui afișat după afișarea nodului), dar deasupra nodului următor la același nivel:

```
|6
|       4
|               2
|               7
|       5
|       9
|               3
```

Un asemenea tipar poate fi obținut prin parcurgerea în preordine a arborelui (Root, Left, Middle, Right), și afișarea fiecărei chei pe o linie, la diferite adâncimi de indentare de la marginea stângă.

3 Exerciții

Înainte de începe exercițiile, adăugați următoarele fapte care definesc arbori (binar și ternar) în fișierul sursă Prolog:

```
% Arbori:
tree1(t(6, t(4,t(2,nil,nil),t(5,nil,nil)), t(9,t(7,nil,nil),nil))).
ternary_tree(t(6, t(4, t(2, nil, nil, nil), nil, t(7, nil, nil, nil)), t(5, nil, nil, nil), t(9, t(3, nil, nil, nil), nil, nil))).
```

1. Scrieți predicatele care parcurge elementele unui arbore ternar în:
   1.1. *preorder*=Root->Left->Middle->Right
   1.2. *inorder*=Left->Root->Middle->Right
   1.3. *postorder*=Left->Middle->Right->Root

```
?- ternary_tree(T), ternary_preorder(T, L).
L = [6, 4, 2, 7, 5, 9, 3], T= ...

?- ternary_tree(T), ternary_inorder(T, L).
L = [2, 4, 7, 6, 5, 3, 9], T= ...

?- ternary_tree(T), ternary_postorder(T, L).
L = [2, 7, 4, 5, 3, 9, 6], T= ...
```

2. Scrieți un predicat *pretty_print_ternary/1* care face afișarea unui arbore ternar.

```
?- ternary_tree(T), pretty_print_ternary(T).
6
    4
        2
        7
    5
    9
        3
T= ... ;
```

3. Scrieți un predicat care calculează înălțimea unui arbore ternar.

```
?- ternary_tree(T), ternary_height(T, H).
```

H=3, T= … ;
false.

4. Scrieți un predicat care colectează într-o listă toate cheile din frunzele unui arbore binar.

?- tree1(T), leaf_list(T, R).
R=[2,5,7], T= … ;
false.

5. Scrieți un predicat care colectează într-o listă toate cheile din noduri interne (non-frunze) ale unui arbore binar de căutare.

?- tree1(T), internal_list(T, R).
R = [4, 6, 9], T = … ;
false.

6. Scrieți un predicat care colectează într-o listă toate nodurile de la aceeași adâncime dintr-un arbore binar.

?- tree1(T), same_depth(T, 2, R).
R = [4, 9], T = … ;
false.

7. Scrieți un predicat care calculează diametrul unui arbore binar.

$$diam(T) = \max\{diam(T.left), diam(T.right), height(T.left) + height(T.right) + 1\}$$

?- tree1(T), diam(T, D).
D = 5, T = … ;
false.

8. Scrieți un predicat care verifică dacă un arbore binar este simetric. Un arbore binar este simetric dacă subarborele stâng este imaginea în oglindă a subarborelui drept. Ne interesează structura arborelui nu și valorile (cheile) din noduri.

?- tree1(T), symmetric(T).
false.

```
?- tree1(T), delete_key(2, T, T1), symmetric(T1).
T = t(6,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,t(7,nil,nil),nil)),
T1 = t(6,t(4,nil,t(5,nil,nil)),t(9,t(7,nil,nil),nil));
false.
```

9. Rescrieți predicatul *delete_key* folosind nodul succesor (arbori binari de căutare).

```
?- tree1(T), delete_key(5, T, T1), delete_key_succ(5, T, T2).
T1 = T2, T2 = t(6,t(4,t(2,nil,nil),nil),t(9,t(7,nil,nil),nil)),
T = t(6,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,t(7,nil,nil),nil)).

?- tree1(T), delete_key(6, T, T1), delete_key_succ(6, T, T2).
T = t(6,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,t(7,nil,nil),nil)),
T1 = t(5,t(4,t(2,nil,nil),nil),t(9,t(7,nil,nil),nil)),
T2 = t(7,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,nil,nil))
```

L8. Structuri incomplete (liste și arbori)

1 Obiective

În lucrarea de față vom introduce structurile incomplete, care înlocuiesc structura vidă (ex: *[]* pentru liste sau *nil* pentru arbori) cu o variabilă liberă.

2 Considerații teoretice

2.1 Reprezentare

Structurile incomplete (structuri instanțiate parțial) oferă posibilitatea modificării variabilei de la final (instanțiere parțială), ceea ce permite adăugarea de elemente noi în aceeași structură (adăugarea la final nu necesită un argument de ieșire adițional). Pentru a ajunge la variabilă liberă de la final, listele incomplete sunt parcurse în aceeași manieră ca listele complete.

Clauzele care specifică comportamentul când se ajunge la finalul structurii trebuie să fie explicite (*chiar și în caz de eșec!*). Prin urmare, condiția de oprire – implicită la structuri complete – trebuie explicitată aici. Adițional, clauzele care testează atingerea variabilei de la finalul structurii trebuie să fie plasate înaintea celorlalte clauze ale predicatului – deoarece variabila liberă de la final se poate unifica cu orice (evitând astfel, unificări nedorite).

Exemple:

```
?- L = [a, b, c|_].
?- L = [1, 2, 3|T], T = [4, 5|U].
?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)).
?- T = t(7, t(5, t(3, A, B), C), t(11, D, E)), D = t(9, F, G).
```

2.2 Liste incomplete

Listele incomplete sunt un tip special de structură incompletă. În loc să se termine în [], o listă incompletă are o variabilă incompletă ca și coadă.

Listele incomplete sunt traversate în același fel ca listele complete, folosind șablonul [H|T] ; diferența apare la procesarea finalului de listă – unde nu întâlnim o listă vidă *[]* la final, ci o variabilă liberă. Acest fapt are următoarele implicații asupra predicatelor care folosesc liste incomplete:

1. testarea finalului de listă trebuie să fie făcută ***întotdeauna,*** chiar și în cazurile de *eșec* – iar eșecul trebuie să fie explicit
2. la testarea finalului de listă, trebuie să se verifice dacă s-a ajuns la o variabilă liberă:
    2.1. some_predicate(L, …):-var(L),!,…
3. clauza care verifică finalul listei trebuie să fie **întotdeauna** prima clauză a predicatului (De ce?)
4. orice listă poate fi adăugată la finalul unei liste incomplete, fără să fie nevoie de o structură separată de ieșire (concatenarea la finalul unei liste incomplete se poate face în aceeași structură de intrare):

    e.g. ?- L = [1, 2, 3|T], T = [4, 5|U], U=[6|_]

Considerând aceste observații, vom continua cu transformările unor predicate cunoscute de liste pentru a funcționa și pe liste incomplete.

2.2.1 Predicatul „member”

Înainte de a scrie noul predicat, urmăriți execuția apelurilor următoare pe predicatul *member1/2*:

```
?- L = [1, 2, 3|_], member1(3, L).
?- L = [1, 2, 3|_], member1(4, L).
?- L = [1, 2, 3|_], member1(X, L).
```

După cum se poate observa, când elementul apare în listă, predicatul *member1/2* funcționează corect; când elementul nu apare în listă, în loc să dea un răspuns negativ, predicatul adaugă elementul la finalul listei incomplete – funcționalitate incorectă. Pentru a corecta acest predicat, trebuie să adăugăm o clauză care specifică explicit eșecul atunci când se ajunge la finalul listei (la variabila liberă):

```
% trebuie testat explicit faptul ca am ajuns la sfârșitul listei
% ceea ce înseamnă că nu am găsit elementul căutat, așa că apelăm fail
member_il(_, L):-var(L), !, fail.
% celelalte 2 clauze sunt la fel ca în trecut
member_il(X, [X|_]):-!.
member_il(X, [_|T]):-member_il(X, T).
```

Urmăriți execuția noului predicat pe întrebările de mai sus.

2.2.2 Predicatul „insert”

După cum ați observat, adăugare unui element la sfârșitul unei liste se poate realiza pe lista de intrare și nu mai este nevoie de un parametru nou pentru lista rezultat – adăugarea se poate face în structura de intrare.

Pentru a face asta, trebuie să parcurgem lista de intrare element cu element și când ajungem la finalul listei, modificăm variabila liberă astfel încât să conțină elementul nou. Dacă elementul deja există, atunci nu îl mai adăugăm.

```
% am ajuns la finalul listei atunci putem adăuga elementul
insert_il1(X, L):-var(L), !, L=[X|_].
insert_il1(X, [X|_]):-!. % elementul există deja
insert_il1(X, [_|T]):- insert_il1(X, T). %traversăm lista să ajungem la final
```

Observație. Există similarități între *insert_il/2* și *member_il/2* – singura diferență este funcționalitatea când ajung la finalul listei.

De asemenea, predicatele *insert_il/2* și *member1/2* sunt foarte similare. De fapt, dacă ar fi să comparăm execuția de aproape, produc același comportament! Înseamnă că testarea apartenenței într-o listă completă este echivalentă cu inserarea într-o listă incompletă – putem să ștergem prima clauză a predicatului *insert_il/2* (De ce?).

```
insert_il2(X, [X|_]):-!.
insert_il2(X, [_|T]):- insert_il2(X, T).
```

Funcționalitatea primei clauze poate fi realizată și de a doua clauză. Dacă elementul căutat nu a fost găsit înseamnă că am ajuns la finalul listei, caz în care variabila de la final se va unifica cu [X|_], inserându-se astfel elementul dat la sfârșitul listei.

Urmărește execuția apelurilor:

```
?- L = [1, 2, 3|_], insert_il2(3, L).
?- L = [1, 2, 3|_], insert_il2(4, L).
?- L = [1, 2, 3|_], insert_il2(X, L).
```

2.2.3 Predicatul „delete”

În predicatul *delete/3* vom păstra aceeași variabilă neinițializată de la final și pentru lista de intrare și pentru lista rezultat. În rest, predicatul este identic cu varianta pentru liste complete.

```
delete_il(_, L, L):-var(L), !. % am ajuns la finalul listei
delete_il(X, [X|T], T):- !. % găsim, ștergem prima apariție și ne oprim
delete_il(X, [H|T], [H|R]):- delete_il(X, T, R). % traversăm, căutam elementul
```

Observație. Condiția de oprire, care corespunde ajungerii la finalul listei de intrare, este prima clauză și are forma știută. Clauzele 2 și 3 sunt aceleași care apar la predicatul *delete/3* pentru liste complete.

Urmărește execuția apelurilor:

```
?- L = [1, 2, 3|_], delete_il(2, L, R).
?- L = [1, 2, 3|_], delete_il(4, L, R).
?- L = [1, 2, 3|_], delete_il(X, L, R).
```

2.3 Arbori incompleți

Arborii incompleți sunt un caz special de structură incompletă – o ramură nu se mai termină cu *nil*, ci cu o variabilă liberă. Observațiile realizate pentru scrierea predicatelor pe liste incomplete se aplică și în cazul arborilor incompleți. Prin urmare, vom aplica aceste observații pentru a dezvolta predicatele discutate pentru liste în secțiunea anterioară: *search_it/2, insert_it/2, delete_it/3* – pentru arbori binari de căutare incompleți.

2.3.1 Predicatul „search”

La fel ca în cazul listelor, predicatul care face căutarea unei chei într-un arbore complet nu are un comportament corect pe arbore incomplet – trebuie să adăugăm explicit o clauză pentru situațiile de eșec (când se ajunge la variabila liberă):

```
search_it(_, T):- var(T), !, fail.
search_it(Key, t(Key, _, _)):- !.
search_it(Key, t(K, L, _)):- Key<K, !, search_it(Key, L).
search_it(Key, t(_, _, R)):- search_it(Key, R).
```

Urmărește execuția apelurilor:

```
?- T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), search_it(6, T).
?- T=t(7, t(5, t(3,_,_), _), t(11,_,_)), search_it(9, T).
```

2.3.2 Predicatul „insert”

Deoarece inserarea unui nod nou se realizează în frunze (la finalul structurii), putem returna rezultatul în structura de intrare. Dacă ne întoarcem la analogia cu listele incomplete, găsim că predicatul care face căutarea pe structura completă va realiza inserare pe arbori incompleți (De ce?):

```
% inserează sau verifică dacă elementul există deja în arbore
insert_it(Key, t(Key, _, _)):-!.
insert_it(Key, t(K, L, _)):-Key<K, !, insert_it(Key, L).
insert_it(Key, t(_, _, R)):-insert_it(Key, R).
```

Urmărește execuția apelurilor:

```
?- T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), insert_it(6, T).
?- T=t(7, t(5, t(3,_,_), _), t(11,_,_)), insert_it(9, T).
```

2.3.3 Predicatul „delete”

Ștergerea dintr-un arbore incomplet de căutare este similară cu ștergerea dintr-un arbore complet. La fel ca în cazul predicatului *delete/3* de la liste incomplete, vom păstra aceleași variabile neinițializate din arborele de intrare și pe arborele rezultat (ca terminator de ramură).

```
delete_it(_, T, T):- var(T), !.  % elementul nu este parte din arbore
delete_it(Key, t(Key, L, R), L):- var(R), !.
delete_it(Key, t(Key, L, R), R):- var(L), !.
delete_it(Key, t(Key, L, R), t(Pred,NL,R)):- !, get_pred(L,Pred,NL).
delete_it(Key, t(K,L,R), t(K,NL,R)):- Key<K, !, delete_it(Key,L,NL).
delete_it(Key, t(K,L,R), t(K,L,NR)):- delete_it(Key,R,NR).

% caută nodul predecesor
get_pred(t(Pred, L, R), Pred, L):- var(R), !.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):- get_pred(R, Pred, NR).
```

Urmărește execuția apelurilor:

```
?- T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), delete_it(6, T, R).
?- T=t(7, t(5, t(3,_,_), _), t(11,_,_)), delete_it(9, T, R).
```

3 Exerciții

Înainte de începe exercițiile, adăugați următoarele fapte care definesc arbori binari (unul complet și unul incomplet) în fișierul sursa Prolog:

```
% Arbori:
incomplete_tree(t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _))).
complete_tree(t(7, t(5, t(3, nil, nil), t(6, nil, nil)), t(11, nil, nil))).
```

Scrieți un predicat care:

1. Convertește o listă incompletă într-o listă completă și viceversa.

```
?- convertIL2CL([1,2,3|_], R).
R = [1, 2, 3].
```

```
?- convertCL2IL([1,2,3], R).
R = [1, 2, 3|_].
```

2. Concatenează 2 liste incomplete (rezultatul este tot o listă incompletă – *De câte argumente este nevoie, am putea renunța la unul?*).

```
?- append_il([1,2|_], [3,4|_], R).
R = [1, 2, 3, 4|_].
```

3. Inversează o listă incompletă (rezultatul este tot o listă incompletă). Implementați în ambele tipuri de recursivitate.

```
?- reverse_il_fwd([1,2,3|_], R).
R = [3, 2, 1|_].

?- reverse_il_bwd([1,2,3|_], R).
R = [3, 2, 1|_].
```

4. Aplatizează o listă adâncă incompletă (rezultatul este o listă simplă incompletă).

```
?- flat_il([[1|_], 2, [3, [4, 5|_]|_]|_], R).
R = [1, 2, 3, 4, 5|_] ;
false.
```

5. Convertește un arbore incomplet într-un arbore complet și viceversa.

```
?- incomplete_tree(T), convertIT2CT(T, R).
R = t(7,t(5,t(3,nil,nil),t(6,nil,nil)),t(11,nil,nil))

?- complete_tree(T), convertCT2IT(T, R).
R = t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _))
```

6. Parcurge un arbore incomplet în pre-ordine (cheile se adaugă într-o listă incompletă).

```
?- incomplete_tree(T), preorder_it(T, R).
R = [7, 5, 3, 6, 11|_]
```

7. Calculează înălțimea unui arbore binar incomplet.

?- incomplete_tree(T), height_it(T, R).
R=3

8. Calculează diametrul unui arbore binar incomplet.

$$diam(T) = \max\{diam(T.left), diam(T.right), height(T.left) + height(T.right) + 1\}$$

?- incomplete_tree(T), diam_it(T, R).
R=4

9. Determină dacă o listă incompletă este o sub-listă într-o altă listă incompletă.

?- subl_il([1, 1, 2|_], [1, 2, 3, 1, 1, 3, 1, ***1, 1, 2,*** 3, 4|_]).
true

?- subl_il([1, 1, 2|_], [1, 2, 3, 1, 1, 3, 1, 1, 1, 3, 2, 4|_]).
false

10. Scrieți predicatul *append_il/2* de concatenare a două liste incomplete folosind doar două argumente (fără argument pentru rezultat).

L9. Liste diferență. Efecte laterale

1 Obiective

În lucrarea aceasta introducem o modalitate nouă de reprezentare pentru liste. Listele incomplete au fost un pas intermediar spre liste diferență. Dacă în listele incomplete, variabila din coadă era anonimă, în cazul listelor diferență finalul listei este numit printr-o variabilă explicită.

Avantajul major pe care listele diferență îl introduc este posibilitatea de a accesa atât începutul, cât și sfârșitul structurii; acest comportament eficientizează adăugarea unui element/unei liste la finalul structurii: dacă în listele complete/incomplete, pentru a adăuga un element la final trebuie să parcurgem toată lista, în cazul listelor diferență se poate adăuga direct în variabila din coada listei (prin utilizarea acestui nou argument).

2 Considerații teoretice

2.1 Reprezentare

Listele diferență se reprezintă prin două părți: **începutul** listei și **sfârșitul** listei. De exemplu:

Lista L=[1,2,3] poate fi reprezentată în forma de listă diferență de variabilele:

S=[1,2,3|X] și E=X, *unde S reprezintă **începutul** (start), iar E **sfârșitul** (end)*

Denumirea de „diferență” vine de la faptul că lista L poate fi calculată prin diferența dintre S și E.

Lista vidă este reprezentată prin 2 variabile egale (S=E). Astfel condiția de oprire se schimbă:

- Pentru liste complete: `pred(L,...):- L=[].`
- Pentru liste incomplete: `pred(L,...):- var(L), !, ... .`
- Pentru liste diferență: `pred (S, E, ...):- S=E, !, ... .`

Exemplu:

S: [1,2,3,4]
E: [3,4]
S-E: [1,2]
S-E (lista diferență) reprezintă lista obținută prin scoaterea părții lui E din S

Nu există avantaje când folosim liste diferență ca în exemplul anterior, dar când combinăm conceptele de variabilă liberă și unificare, listele diferență devin unelte utile. De exemplu, lista [1,2] poate fi reprezentată ca lista diferență [1,2|X]-X, unde X este o variabilă liberă.

2.2 Adăugarea unui element la finalul unei liste diferență

O listă Prolog este accesată prin primul element și restul listei. Piedica acestui mod de a vedea lista este faptul că pentru a accesa al n-lea element, trebuie să accesăm toate elementele de dinaintea lui. Dacă, de exemplu, avem nevoie să adăugăm un element la finalul listei, atunci trebuie să trecem prin toate elementele din listă pentru a ajunge la final.

```
add_cl(X, [H|T], [H|R]):- add_cl(X, T, R).
add_cl(X, [], [X]).
```

Daca utilizăm liste diferență (reprezentate prin două părți, începutul liste S și finalul listei E), predicatul pentru adăugarea unui element la final devine:

```
add_dl(X, LS, LE, RS, RE):- RS = LS, LE = [X|RE].
% variabila LE va conține pe prima poziție elementul adăugat
```

Dacă îl testăm în interpretorul Prolog vom primi următorul răspuns:

```
?- LS=[1,2,3,4|LE], add_dl(5,LS,LE,RS,RE).
LE = [5|RE],
LS = [1,2,3,4,5|RE],
RS = [1,2,3,4,5|RE]
```

Putem vizualiza acest proces pas cu pas:

- Inițial avem → `LS = [1,2,3,4|LE]`
- Prima operație: `RS=LS` → `RS = [1,2,3,4|LE]`
- A doua operație: `LE=[X|RE]` → `RS = [1,2,3,4|[X|RE]]`
- Înlocuim X cu valoarea sa numerică →`RS = [1,2,3,4|[5|RE]]`
- Simplificăm lista `RS` → `RS = [1,2,3,4,5|RE]`

Pentru a înțelege mai bine modul de funcționare a predicatului, ne putem imagina lista ca fiind reprezentată prin doi “pointeri”, unul care arată începutul listei (LS) și al doilea finalul listei (LE), o variabilă fără o valoare atribuită.

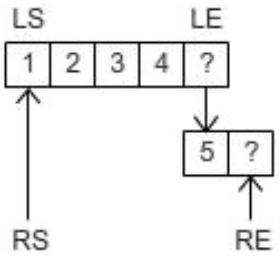


Figura 9.1. Adăugarea unui element la sfârșitul unei liste diferență

Rezultatul este, de asemenea, reprezentat prin două argumente care marchează începutul/sfârșitul structurii. Lista rezultat va fi lista de intrare cu un element inserat la final. Începutul listei de intrare și lista rezultată sunt aceeași, deci putem unifica variabila de început a listei de intrare cu variabila de început a listei rezultat (`RS=LS`).
Lista rezultat trebuie să aibă un final, ca si lista de intrare, aceasta va fi într-o variabilă (`RE`), dar trebuie cumva, să modificăm lista de intrare pentru a adăuga un nou element la final. Deoarece finalul listei de intrare este o variabilă liberă, putem să o unificăm cu lista care începe cu acest nou element și o variabilă nouă, finalul listei rezultat (`LE=[X|RE]`). După finalizarea execuției predicatului, putem observa ca lista de intrare LS și lista rezultat RS au aceeași valoare, iar finalul listei de intrare nu mai este o variabilă liberă (`LE=[5|RE]`).

2.3 Concatenare cu liste diferență

În cazul listelor diferență, predicatul *append_dl/6* se va scrie pe o singură linie:

```
append_dl(LS1,LE1, LS2,LE2, RS,RE):- RS=LS1, LE1=LS2, RE=LE2.
```

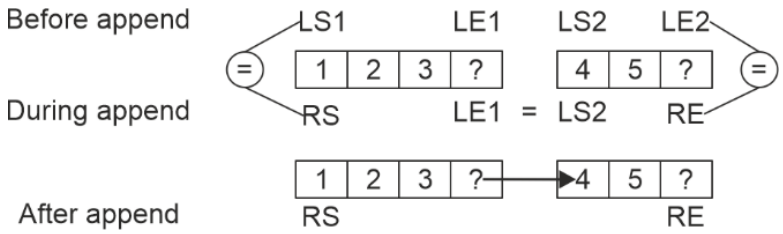


Figura 9.2. Concatenarea a două liste

Dacă îl apelăm în interpretorul Prolog, vom primi următorul răspuns:

```
?- LS1=[1,2,3|LE1], LS2=[4,5|LE2], append_dl(LS1, LE1, LS2, LE2, RS,
RE).
LE1 = LS2, LS2 = [4, 5|RE],
LE2 = RE,
LS1 = RS, RS = [1, 2, 3, 4, 5|RE].
```

Putem vizualiza acest proces pas cu pas:

- Inițial avem → `LS1=[1,2,3|LE1], LS2=[4,5|LE2]`

- Prima operație: `RS=LS1` → `RS=[1,2,3|LE1]`
- A doua operație: `LE1=LS2` → `LE1 = [4,5|LE2]`
  - Putem substitui `LE1` în `RS` → `RS=[1,2,3|[4,5|LE2]]`
- A treia operație: `RE=LE2`
  - Simplificăm lista `RS` → `RS=[1,2,3|[4,5|RE]]`
- Simplificam lista RS folosind șablonul: → `RS = [1,2,3,4,5|RE]`

2.3.1 Quicksort

Reamintim algoritmul quicksort (și predicatul): secvența de intrare este împărțită în două – secvența de elemente mai mici sau egale cu pivotul și secvența de elemente mai mari decât pivotul; această procedură este apelată recursiv pe fiecare partiție și secvențele sortate rezultate sunt concatenate cu pivotul pentru a genera secvența sortată:

```
quicksort([H|T], R):-
   partition(H, T, Sm, Lg),
   quicksort(Sm, SmS),
   quicksort(Lg, LgS),
   append(SmS, [H|LgS], R).
quicksort([], []).
```

Ca în cazul predicatului *inorder/2*, predicatul *quicksort/2* necesită timp suplimentar de execuție pentru apelul de *append/3* între rezultatele apelurilor recursive. Pentru a evita acest lucru, putem utiliza liste diferența:

```
quicksort_dl([H|T], S, E):- % s-a adăugat un parametru nou
    partition(H, T, Sm, Lg), % predicatul partition a rămas la fel
    quicksort_dl(Sm, S, [H|L]), %concatenare implicită
    quicksort_dl(Lg, L, E).
quicksort_dl([], L, L). % condiția de oprire s-a modificat

partition(P, [X|T], [X|Sm], Lg):-
     X<P,
     !,
     partition(P, T, Sm, Lg).
partition(P, [X|T], Sm, [X|Lg]):-
     partition(P, T, Sm, Lg).
partition(_, [], [], []).
```

Predicatul *partition/4* rămâne neschimbat, scopul acestuia este de a împărți lista în două prin compararea elementelor cu pivotul. Toate elementele din listă trebuie accesate pentru această operație, prin urmare nu putem îmbunătăți performanța acestui predicat.

Predicatul *quicksort_dl/3* funcționează în aceeași manieră ca înainte: împarte lista în elemente mai mari și mai mici decât pivotul și aplică *quicksort_dl* recursiv pe fiecare partiție. Diferența între versiunea originală și aceasta este la lista rezultat, reprezentată prin două elemente, începutul și finalul listei, și în consecință modul prin care cele două rezultate ale apelurilor recursive sunt legate cu pivotul (figura de mai jos).

Urmărește execuția apelurilor:

```
?- quicksort_dl([4,2,5,1,3], L, []).
?- quicksort_dl([4,2,5,1,3], L, _).
```

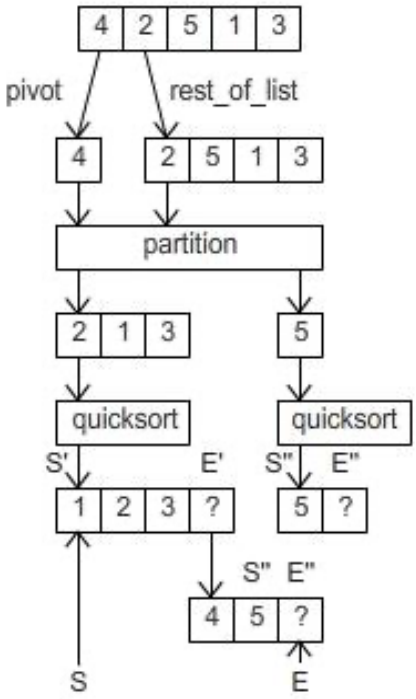


Figura 9.3. Quicksort folosind liste diferență

2.3.2 Parcurgerea arborilor binari

Predicatele pentru parcurgerea unui arbore pot fi utilizate pentru a extrage elementele dintr-un arbore într-o listă, într-o ordine specifică. In cadrul parcurgerii, predicatul *append/3* este apelat la nivelul fiecărui nod, pentru a construi lista rezultat din listele obținute pe apelurile pe subarbori și cheia curentă. Astfel, in execuție se va trece prin aceleași elementele ale listei de multiple ori pentru a forma lista rezultat:

```
inorder(t(K,L,R),List):-
   inorder(L,ListL),
   inorder(R,ListR),
   append1(ListL,[K|ListR],List).
inorder (nil,[]).
```

Prin urmărirea execuției predicatului *inorder/2,* putem observa efortul făcut de predicatul *append/3*. Este de asemenea vizibil că predicatul *append/3* accesează aceleași elemente din lista rezultat de mai multe ori pe măsură ce rezultatele intermediare sunt concatenate la cel final:

```
[. . .]
 8     3 Exit: inorder(t(5,nil,nil),[5]) ?
12     3 Call: append1([2],[4,5],_1594) ?
13     4 Call: append1([],[4,5],_10465) ?
13     4 Exit: append1([],[4,5],[4,5]) ?
12     3 Exit: append1([2],[4,5],[2,4,5]) ?
[. . .]
22     2 Call: append1([2,4,5],[6,7,9],_440) ?
23     3 Call: append1([4,5],[6,7,9],_20633) ?
24     4 Call: append1([5],[6,7,9],_21109) ?
25     5 Call: append1([],[6,7,9],_21585) ?
25     5 Exit: append1([],[6,7,9],[6,7,9]) ?
24     4 Exit: append1([5],[6,7,9],[5,6,7,9]) ?
23     3 Exit: append1([4,5],[6,7,9],[4,5,6,7,9]) ?
22     2 Exit: append1([2,4,5],[6,7,9],[2,4,5,6,7,9]) ?
[. . .]
```

Putem îmbunătăți eficiența predicatului *inorder/2* prin înlocuirea concatenării cu operația echivalentă pe liste diferență. Predicatul *inorder_dl/3* va avea 3 argumente: nodul curent, începutul listei rezultate și finalul listei rezultate:

```
% la finalul arborelui, unificăm începutul și finalul listei
% rezultat - lista vidă este reprezentată de 2 variabile egale
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
    % obtinem începutul și finalul listelor pentru subarborele stâng
și drept
    inorder_dl(L,LSL,LEL),
    inorder_dl(R,LSR,LER),
    % începutul listei rezultat este începutul listei subarborelui
stâng
    LS=LSL,
    % cheia K este adăugat între finalul din stânga și începutul din
dreapta
    LEL=[K|LSR],
    % finalul listei rezultat este finalul listei subarborelui drept
    LE=LER.
```

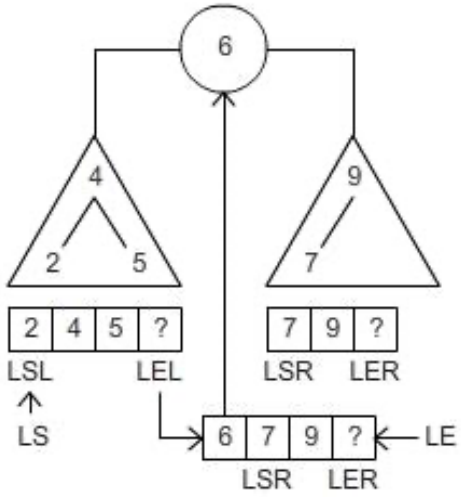


Figura 9.4. Concatenarea a două liste în parcurgerea în inordine a unui arbore

Urmărește execuția apelurilor:

```
?- tree1(T), inorder_dl(T,L,[]).
?- tree1(T), inorder_dl(T,L,_).
```

Predicatul poate fi simplificat prin înlocuirea unificărilor explicite cu unificări implicite:

```
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
    inorder_dl(L, LS, [K|LT]),
    inorder_dl(R, LT, LE).
```

2.4 Efecte laterale

Efectele laterale se referă la manipularea dinamică a definițiilor de predicate și acest lucru se poate realiza cu următoarele predicate predefinite:

- *assert/1 (= assertz/1)* → adaugă la sfârșitul bazei de cunoștințe clauza dată în argument;
- *asserta/1* → adaugă la începutul bazei de cunoștințe clauza dată în argument;
- *retract/1* → șterge prima clauză care se unifică cu argumentul;
- *retractall/1* → șterge toate clauzele care se unifică cu argumentul (în SWI-Prolog răspunsul va fi *true* chiar dacă nu șterge nimic).

Predicatele care sunt definite și/sau apelate în fișierul „.pl” se numesc predicate statice. Predicatele care sunt manipulate cu *assert/retract* se numesc predicate dinamice. În contrast cu predicatele statice pe care le-am văzut până acum, predicatele dinamice trebuie să fie declarate ca fiind dinamice.

În Prolog, un predicat este static sau dinamic. Un predicat static are faptele/regulile predefinite la începutul execuției și acestea nu se schimbă pe parcursul execuției. În mod normal, faptele/regulile sunt scrise într-un fișier de cod Prolog care este încărcat în timpul sesiunii Prolog. Dacă vrem să adăugam faptele adiționale (sau chiar reguli) unui predicat în timpul execuției, vom folosi *assert/asserta/assertz*; în cazul în care vrem să ștergem fapte vom folosi *retract/retractall*. Pentru a se putea executa aceste operații, predicatul trebuie să fie dinamic.

Un predicat adăugat pentru prima dată cu *assert/1* este un predicat dinamic. Dacă vrem să manipulăm un predicat care apare în fișierul „.pl" atunci trebuie să-l declarăm explicit ca predicat dinamic, adăugând următoare linie la începutul fișierului:

```
:-dynamic <nume_predicat>/<aritate>.
```

Când folosim aceste predicate trebuie să ținem cont de următoarele aspecte:

Backtracking-ul nu invalidează efectul unui *assert*, ex: dacă un predicat a fost adăugat cu *assert*, rămâne in baza de predicate până când este șters explicit cu retract, chiar dacă nodul corespondent acelui apel de *assert* este șters din arborele de execuție (ex: din cauza backtracking-ului).

Predicatul *assert* **întotdeauna rezultă în succes; nu face backtracking**.

- Predicatul retract **poate să eșueze, caz în care se lansează mecanismul de backtracking**. În cazul predicatului *retract*, backtracking-ul invalidează temporar ștergerea pentru predicatele din același corp al clauzei cu apelul predicatului *retract* și care au fost apelate înaintea predicatului *retract*; astfel se păstrează *logical update view*.

Pentru a înțelege mai bine cum funcționează *retract/1* urmăriți execuția apelului:

```
?-    assert(insect(ant)),
      assert(insect(bee)),
      retract(insect(A)),
      writeln(A),
      retract(insect(B)),
      fail.
```

Ați observat probabil că această întrebare ne dă rezultatul:

```
ant
bee
false
```

Deși al doilea retract șterge faptul *insect(bee)*, când se face backtracking și se ajunge la primul apel de *retract*, clauza este încă prezentă în logical view- practic, nu vede faptul că clauza a fost ștearsă de al doilea apel de retract. Astfel *insect(A)* încă se poate unifica cu „bee”.

Prolog are și predicatul *retractall/1,* cu următorul comportament: va șterge toate clauzele predicatelor care se potrivesc cu argumentul. În unele versiuni de Prolog, *retractall* poate da fail dacă nu există nimic de extras. Pentru a evita acest lucru, ați putea să alegeți să faceți un assert a unei clauze dummy de tipul potrivit. Însă, în SWI Prolog, *retractall* reușeste chiar și pentru un apel cu niciun fapt/regulă care să se potrivească.

Manipularea dinamică a bazei de cunoștințe prin *assert/retract* este utilizată în meta-programare și pentru salvarea rezultatelor de la calcule (memoisation/ caching), astfel încât să nu fie pierdute prin backtracking. Astfel, dacă aceeași întrebare este pusă în viitor, răspunsul poate fi obținut fără a fi nevoie de a recalcula. Această tehnică se numește memoisation, sau caching, în unele aplicații poate să crească eficiența. De asemenea, aceste operații pot fi folosite pentru a schimba dinamic comportamentul unor predicate la rulare (meta-programare). Acest proces duce la un cod greu de urmărit. În cazurile cu mult backtracking, devine și mai greu. Prin urmare, această caracteristică non-declarativă a Prolog-ului ar trebui folosită cu atenție.

Un exemplu de predicat care memorează rezultatele parțiale, folosind efecte laterale, pentru a calcula al n-lea număr din *secvența Fibonacci* (ați întâlnit varianta mai puțin eficientă în laboratorul al doilea):

```
:-dynamic memo_fib/2.

fib(N,F):- memo_fib(N,F), !.
fib(N,F):-
    N>1,
    N1 is N-1,
    N2 is N-2,
    fib(N1,F1),
    fib(N2,F2),
    F is F1+F2,
    assertz(memo_fib(N,F)).
fib(0,1).
fib(1,1).
```

Urmărește execuția apelurilor:

```
?- listing(memo_fib/2). % afișează toate definițiile predicatului memo_fib cu 2 parametrii
```

```
?- fib(4,F).
?- listing(memo_fib/2).
?- fib(10,F).
?- listing(memo_fib/2).
?- fib(10,F).
```

2.4.1 Afișarea rezultatelor memorate

Oricând avem nevoie să colectăm toate răspunsurile stocate în baza de predicate prin assert-uri, putem folosi **failure driven loops** care fortează Prolog-ul să facă backtracking până când rămâne fără posibilități. Tiparul pentru un failure driven loop care citește toate clauzele stocate – de exemplu pentru predicatul *memo_fib/2* de mai sus – și afișează toate numerele *fibonacci* calculate:

```
print_all:-
    memo_fib(N,F),
    write(N),
    write(' - '),
    write(F),
    nl,
    fail.
print_all.
```

Ne vom folosi de tehnica backtracking (forțând *eșecul*) pentru a parcurge toate clauzele predicatului *memo_fib/2* adăugate în baza de cunoștințe cu *assert*.

Urmărește execuția apelurilor:

```
?-print_all.
?-retractall(memo_fib(_,_)).
?-print_all.
```

2.4.2 Colectarea rezultatelor memorate

Pentru colectarea rezultatelor într-o listă ne putem folosi de predicatul predefinit: *findall/3.*

Urmărește execuția apelurilor:

```
?- findall(X, append(X,_,[1,2,3,4]), List).
?- findall(lists(X,Y), append(X,Y,[1,2,3,4]), List).
?- findall(X, member(X,[1,2,3]), List).
```

Vom vedea un exemplu de folosire a efectelor laterale pentru a găsi toate răspunsurile unei întrebări: vom scrie predicatul care generează toate permutările unei liste și le returnează într-o listă. Vrem ca predicatul să funcționeze în felul următor:

```
?- all_perm([1,2,3],L).
L=[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]];
```

no.

Știind că deja puteți implementa predicatul *perm/2* – predicatul care generează o permutare a listei de intrare (vezi lucrarea *L5. Metode de sortare*) – specificația predicatului *all_perm/2* este:

```
all_perm(L,_):-
    perm(L,L1),
    assertz(p(L1)),
    fail.
all_perm(_,R):- collect_perms(R).

collect_perms([L1|R]):- retract(p(L1)), !, collect_perms(R).
collect_perms([]).

perm(L, [H|R]):-append(A, [H|T], L), append(A, T, L1), perm(L1, R).
perm([], []).
```

Urmărește execuția apelurilor:

```
?- retractall(p(_)), all_perm([1,2],R).
?- listing(p/1).
?- retractall(p(_)), all_perm([1,2,3],R).
```

Întrebări:

1. De ce am nevoie de *retractall* înainte de apelul la *all_perm/2*?
2. De ce este nevoie de ! după **retract** în predicatul *collect_perms/1*?
3. Ce tip de recursivitate este folosit în predicatul *collect_perms/1*? Se poate face colectarea în celălalt tip de recursivitate? Care este ordinea permutărilor în acest caz?
4. Predicatul *collect_perms/1* distruge rezultatele salvate?

3 Exerciții

Înainte de a începe exercițiile, adăugați următoarele fapte care definesc arbori (complet și incomplet binar) în fișierul sursă Prolog:

```
% Arbori:
complete_tree(t(6, t(4,t(2,nil,nil),t(5,nil,nil)), t(9,t(7,nil,nil),nil))).
incomplete_tree(t(6, t(4,t(2,_,_),t(5,_,_)), t(9,t(7,_,_),_))).
```

Scrieți un predicat care:

1. Convertește o listă completă într-o listă diferență și viceversa.

```
?- convertCL2DL([1,2,3,4], LS, LE).
LS = [1, 2, 3, 4|LE]

?- LS=[1,2,3,4|LE], convertDL2CL(LS,LE,R).
R = [1, 2, 3, 4]
```

2. Convertește o listă incompletă într-o listă diferență și viceversa.

```
?- convertIL2DL([1,2,3,4|_], LS, LE).
LS = [1, 2, 3, 4|LE]

?- LS=[1,2,3,4|LE], convertDL2IL(LS,LE,R).
R = [1, 2, 3, 4|_]
```

3. Aplatizează o listă adâncă folosind liste diferență în loc de append.

```
?- flat_dl([[1], 2, [3, [4, 5]]], RS, RE).
RS = [1, 2, 3, 4, 5|RE] ;
false
```

4. Generează toate descompunerile posibile a unei liste în doua sub-liste fără a folosi predicatul predefinit findall.

```
?- all_decompositions([1,2,3], List).
List=[ [[], [1,2,3]], [[1], [2,3]], [[1,2], [3]], [[1,2,3], []] ] ;
false
```

5. Traversează un arbore în *pre-ordine* și încă unul pentru *post-ordine* folosind liste diferență în manieră implicită

```
?- complete_tree(T), preorder_dl(T, S, E).
S = [6, 4, 2, 5, 9, 7|E]

?- complete_tree(T), postorder_dl(T, S, E).
S = [2, 5, 4, 7, 9, 6|E]
```

6. Colectează toate nodurile care au chei pare, dintr-un arbore binar **complet** folosind liste diferență.

```
?- complete_tree(T), even_dl(T, S, E).
S = [2, 4, 6|E]
```

7. Colectează toate nodurile care au chei între K1 și K2, dintr-un arbore binar de căutare **incomplet** folosind liste diferență.

```
?- incomplete_tree(T), between_dl(T, S, E, 3, 7).
S = [4, 5, 6|E]
```

8. Colectează toate cheile de la o adâncime dată K, dintr-un arbore binar de căutare **incomplet** folosind liste diferență.

```
? – incomplete_tree(T), collect_depth_k(T, 2, S, E).
S = [4, 9|E].
```

L10. Grafuri

1 Obiective

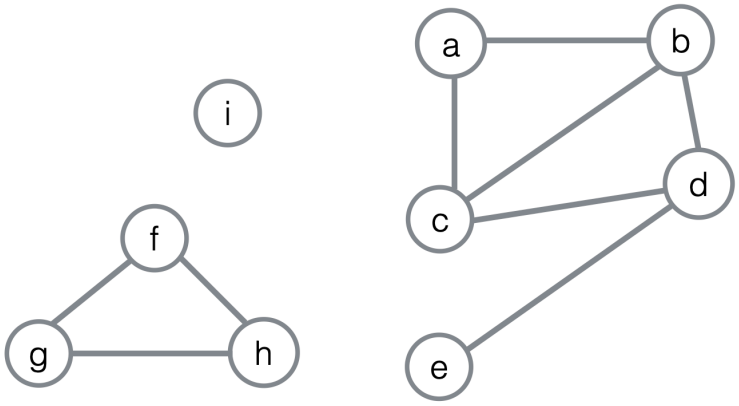
În lucrarea de față vom prezenta cum se poate reprezenta un graf în Prolog și cum putem căuta drumul între două noduri (drum simplu, ciclu, drum optim, drum restricționat, ciclu Hamiltonian).

2 Considerații teoretice

2.1 Reprezentare

Un graf este o structura compusa din două mulțimi: mulțimea nodurilor și cea a muchiilor. Dacă graful este orientat, vorbim despre vârfuri și arce.

Exemplu de graf neorientat:



Există mai multe alternative de a reprezenta un graf în Prolog și le putem clasifica după următoarele criterii:

A. Tipul de reprezentare:

1. O colecție de noduri și o colecție de muchii
2. O colecție de noduri și lista asociată de vecini

B. Locul unde sunt salvate:

1. In baza de cunoștințe, într-o colecție de predicate de tip axiomă/fapt (*knowledge base*)
2. In memoria principală, ca *data object* (de ex. într-o structură, lista, etc.)

În consecință, sunt utilizate frecvent patru reprezentări principale:

A1B1: Un predicat *edge,* stocat ca si colecție de fapte (forma *edge-clause*):

```
edge(a,b).
edge(b,a).
edge(b,c).
edge(c,b). %etc

edge(i, nil).
```

Nodurile izolate se reprezintă ca având o muchie cu *nil*: edge(f, nil). Dacă graful este unul neorientat, putem scrie predicatul în modul următor (pentru a evita nevoia de a scrie muchiile în ambele direcții):

```
is_edge(X,Y):- edge(X,Y); edge(Y,X).
```

A2B1: O colecție de noduri și listele asociate de vecini, stocate în faptele predicatului *neighbor/2* (forma *neighbor list-clause*):

```
neighbor(a, [b, d]).
neighbor(b, [a, c, d]).
neighbor(c, [b, d]).
neighbor(h, []). %etc
```

A1B2: O listă de muchii, in memorie (forma *edge-list*):

?- Graph = [e(a,b), e(b,a), …].

A2B2: O listă de liste de vecini (forma *neighbor list-list*):

?- Graph = [n(a, [b,d]), n(b, [a,c,d]), n(c, [b,d]), n(d, [a,b,c]), n(e, [f,g]), n(f, [e]), n(g, [e]), n(h, [])].

Cea mai potrivită reprezentare depinde de problemă. Prin urmare, este convenient să știm modurile prin care putem să transformăm între diferite reprezentări de grafuri. Aici, furnizăm un exemplu de conversie din forma de *neighbor list-clause (A2B2)* în forma de *edge-clause (A1B2)*:

```
% declarăm predicatul dinamic pentru a putea folosi retract
:-dynamic neighbor/2.
% predicatul neighbor este considerat a fiind static deoarece este introdus
% în fișier, doar prin adăugarea declarației explicite ne este permis
% să folosim operația de retract asupra lui
```

```
% un exemplu de graf – prima componentă conexă a grafului
neighbor(a, [b, d]).
neighbor(b, [a, c, d]).
neighbor(c, [b, d]).
%etc.

neighb_to_edge:-
   %extrage un predicat neighbor
   retract(neighbor(Node,List)),!,
   % și apoi îl procesează
   process(Node,List),
   neighb_to_edge.
neighb_to_edge. % dacă nu mai sunt predicate neighbor/2, ne oprim

% procesarea presupune adăugarea de predicate edge și node
% pentru un predicat neighbor, prima dată adăugăm muchiile
% până când lista devine vidă iar abia apoi predicatele de tip node
process(Node, [H|T]):- assertz(gen_edge(Node, H)), process(Node, T).
process(Node, []):- assertz(gen_node(Node)).
```

Inițial graful este salvat în baza de predicate. Predicatul *neighb_to_edge* citește o clauză a predicatului *neighbor* la un moment dat și proceseaza informația din fiecare clauză separat.

Predicatul *process/2* traversează lista de vecini (al doilea argument) a nodului curent (primul argument) și adaugă (prin *assert*) un nou fapt la predicatul *gen_edge* pentru fiecare vecin nou al nodului curent. La condiția de oprire(a doua clauză), ne oprim când lista de vecini devine vidă și facem *assert* la un nou fapt de tip *gen_node*.

Predicatul *neighb_to_edge/0* se folosește de recursivitate și *retract* pentru a parcurge secvențial (si șterge) clauzele predicatului *neighbor/2,* și se oprește când nu mai exista nicio clauză a acestuia. Fără folosirea operației de *retract*, această implementare ar putea rezulta într-o buclă infinită (de ce?).

Pentru a pune întrebări acestui predicat, avem sintaxa următoare:

```
?- neighb_to_edge.
true;
false.
```

Pentru a verifica rezultatul, putem inspecta baza de predicate folosind *listing/1* (verificați specificația predicatului). Totodată, se poate utiliza *retractall/1*, pentru a avea garanția că predicatele stocate provin din execuția curentă (ștergem, anterior execuției, toate – eventualele – clauze ale predicatului). Întrebarea devine:

```
?- retractall(gen_edge(_,_)), neighb_to_edge.
true;
false.

?- listing(gen_edge/2).
```

```
:- dynamic gen_edge/2.

gen_edge(a, b).
gen_edge(a, d).
gen_edge(b, a).
gen_edge(b, c).
gen_edge(b, d).
gen_edge(c, b).
gen_edge(c, d).
```

Această abordare prezintă un dezavantaj. Datorită utilizării *retract*-ului, predicatul *neighbor/2* va fi eliminat în întregime din baza de predicate și nu va mai putea fi utilizat în aceeași instanță de execuție. Verificați următoarea întrebare:

```
?- retractall(gen_edge(_,_)), neighb_to_edge, listing(gen_edge/2), listing(neighbor/2).
```

2.1.1 Implementări alternative

Predicatul anterior, *neighb_to_edge*, poate fi implementat într-un mod echivalent prin utilizarea unui *failure-driven loop* (laboratorul anterior, efecte laterale), fără distrugerea predicatului *neighbor/2*. Predicatul *process/2* va rămâne același:

```
neighb_to_edge_v2:-
   neighbor(Node,List), % accesăm faptul curent
   process(Node,List),
   fail. % lansam backtracking, pt. a forta neighbor/2 sa treaca la urm. fapt
neighb_to_edge_v2.
```

În clauza 1 se instanțiază și se procesează, pe rând (prin backtracking), argumentele fiecărei clauze a predicatului *neighbor/2*. După ce toate s-au

„parcurs”, se ajunge la a doua clauză a *neighb_to_edge/0*, unde execuția se oprește cu succes.

Prima implementare (folosind recursivitatea) are aceeași eficiență în timp ca a doua versiune, dar distruge predicatul *neighbor/2* (din cauza utilizării *retract*-ului). Se poate implementa aceasta transformare folosind recursivitatea fără a distruge graful original, utilizând un predicat *seen/1* pentru a marca ce fapte ale predicatului *neighbor/2* au fost deja parcurse – evitându-se, astfel, intrarea într-o buclă infinită. Care ar fi complexitatea acelei soluții? Este eficientă strategia?

2.2 Drumuri în graf

Aceasta secțiune prezinta soluții pentru: drum simplu între două noduri, drum restricționat, drum optim și ciclu Hamiltonian al unui graf.

2.2.1 Drum Simplu

Presupunem că graful este reprezentat sub forma *edge-clause*. Următorul predicat caută un drum între două noduri și returnează lista de noduri parcurse:

```
% path(Source, Target, Path)
% drumul parțial începe cu nodul sursă – acesta este un wrapper
path(X, Y, Path):-path(X, Y, [X], Path).

% când sursa (primul argument) este egal cu destinația (al doilea argument),
% atunci știm că drumul a ajuns la final
% și putem unifica drumul parțial cu cel final
path(Y, Y, PPath, PPath).
path(X, Y, PPath, FPath):-
   edge(X, Z),                    % căutăm o muchie
   not(member(Z, PPath)),         % care nu a mai fost parcursă
   path(Z, Y, [Z|PPath], FPath).  % și o adăugăm în rezultatul parțial
```

Urmărește execuția apelului:
?- path(a,c,R).

Puteți folosi exemplul de graf de la începutul lucrării, sau chiar să adăugați niște muchii la acesta. Ce se întâmplă când repetăm întrebarea?

2.2.2 Drum Restricționat

Drumul restricționat presupune trecerea printr-un anumit set de noduri în ordinea dată. Deoarece *path/3* returnează drumul în ordine inversă atunci vom aplica reverse.

```
% restricted_path(Source, Target, RestrictionsList, Path)
restricted_path(X,Y,LR,P):-
   path(X,Y,P),
   reverse(P,PR),
   check_restrictions(LR, PR).

% verificăm dacă se respectă restricția
check_restrictions([],_):- !.
check_restrictions([H|TR], [H|TP]):- !, check_restrictions(TR,TP).
check_restrictions(LR, [_|TP]):-check_restrictions(LR,TP).
```

Predicatul *restricted_path/4* caută un drum între nodul sursă și destinație și verifică dacă acest drum satisface restricțiile specificate în LR (i.e. dacă trece printr-o secvență de noduri, specificată în LR), folosind predicatul *check_restrictions/2*, care face de fapt verificarea.

Predicatul *check_restrictions/2* traversează lista de restricții (primul argument) și lista care reprezintă drumul (al doilea argument) simultan, cât timp primele elemente din cele două liste coincid (clauza 2). La momentul în care sunt diferite, vom continua doar cu a doua listă (clauza 3). Predicatul reușește când prima listă devine vidă (clauza 1).

Întrebare: Ce se întâmplă dacă mutăm condiția de oprire ca ultima clauză? Avem nevoie de tăierea de backtracking în condiția de oprire?

Urmărește execuția apelurilor:
```
?- check_restrictions([2,3], [1,2,3,4]).
?- check_restrictions([1,3], [1,2,3,4]).
?- check_restrictions([1,3], [1,2]).
?- restricted_path(a, c, [a,c,d], R).
```

2.2.3 Drum Optim

Considerăm un drum optim între două noduri din graf ca acel drum care are lungime minimă (număr minim de noduri in cale). O soluție posibilă generează toate drumurile prin backtracking și selectează drumul de lungime minimă. Evident, această abordare este ineficientă. O îmbunătățire ar fi sa reținem soluția optimă parțială folosind efecte laterale (în baza de predicate) și sa o actualizăm pe măsură ce avansăm:

```
:- dynamic sol_part/2.

% optimal_path(Source, Target, Path)
optimal_path(X,Y,Path):-
   asserta(sol_part([], 100)),      % 100 = distanța maximă inițială
   path(X, Y, [X], Path, 1).
optimal_path(_,_,Path):-
   retract(sol_part(Path,_)).

% path(Source, Target, PartialPath, FinalPath, PathLength)
% când ținta este egală cu sursa, salvăm soluția curentă
path(Y,Y,Path,Path,LPath):-
   % scoatem ultima soluție
   retract(sol_part(_,_)),!,
   % salvăm soluția curentă
   asserta(sol_part(Path,LPath)),
   % căutăm o altă soluție
   fail.
path(X,Y,PPath,FPath,LPath):-
   edge(X,Z),
   not(member(Z,PPath)),
   % calculăm distanța parțială
   LPath1 is LPath+1,
   % extragem distanța de la soluția precedentă
   sol_part(_,Lopt),
   % dacă distanța curentă nu depășește distanța precedentă
   LPath1<Lopt,
   % mergem mai departe
   path(Z,Y,[Z|PPath],FPath,LPath1).
```

Predicatul *path/4* generează, prin backtracking, toate drumurile care sunt mai bune decât soluția curentă parțială și actualizează soluția curentă parțială când un drum mai scurt este găsit. Când o soluție mai bună decât cea curentă este găsită, predicatul înlocuiește soluția optimă veche în baza de predicate (clauza 1) și după continuă căutarea, prin lansarea mecanismului de backtracking (folosind *fail*).

Urmărește execuția apelului:
?- optimal_path(a,c,R).

Rețineți. Când folosim *assert/retract*, trebuie să "curățăm" rezultatele rulărilor anterioare de pe baza de predicate, pentru a nu compromite rulările ulterioare.

2.2.4 Ciclu Hamiltonian

Un ciclu Hamiltonian este un ciclu care trece prin toate nodurile o singură dată (cu excepția nodului de start care este începutul și sfârșitul acestui ciclu). Evident, nu toate grafurile conțin un astfel de ciclu. Predicatul *hamilton/3* este prezentat mai jos:

```
% hamilton(NumOfNodes, Source, Path)
hamilton(NN, X, Path):- NN1 is NN-1, hamilton_path(NN1, X, X, [X],Path).
```

Predicatul *hamilton_path/5* vă rămâne de implementat. Predicatul acesta ar trebui să caute un drum închis începând din X, de lungime NN1 (numărul de noduri din graf, minus 1).
Urmăriți execuția predicatului *hamilton/3* folosind graful dat ca exemplu.

3 Exerciții

1. Continuati implementarea predicatului pentru construirea unui ciclu Hamiltonian.

```
edge_ex1(a,b).
edge_ex1(b,c).
edge_ex1(a,c).
edge_ex1(c,d).
edge_ex1(b,d).
edge_ex1(d,e).
edge_ex1(e,a).
```

```
?- hamilton(5, a, P).
P = [a, e, d, c, b, a]
```

2. Scrieți predicatul *euler/3* care poate să găsească drumuri Euleriene într-un graf dat, pornind de la un nod dat.

```
% euler(NE, S, R). – unde S este nodul sursă
% și NE este numărul de muchii din graf
edge_ex2(a,b).
edge_ex2(b,e).
edge_ex2(c,a).
edge_ex2(d,c).
edge_ex2(e,d).
```

```
?- euler(5, a, R).
```

R = [[b, a], [e, b], [d, e], [c, d], [a, c]];
R = [[c, a], [d, c], [e, d], [b, e], [a, b]]

3. Scrie predicatul *cycle(X,R)* care găsește un ciclu ce pornește din nodul *X* dintr-un graf G (folosind reprezentarea *edge/2*) și pune rezultatul în *R*. Predicatul trebuie să returneze toate ciclurile prin backtracking.

```
edge_ex3(a,b).
edge_ex3(a,c).
edge_ex3(c,e).
edge_ex3(e,a).
edge_ex3(b,d).
edge_ex3(d,a).
```

?- cycle(a, R).
R = [a,d,b,a] ;
R = [a,e,c,a] ;
false

4. Scrieți predicatul *cycle(X,R)* din exercițiul anterior folosind reprezentarea *neighbour/2*.

```
neighb_ex4(a, [b,c]).
neighb_ex4(b, [d]).
neighb_ex4(c, [e]).
neighb_ex4(d, [a]).
neighb_ex4(e, [a]).
```

?- cycle_neighb(a, R).
R = [a,d,b,a] ;
R = [a,e,c,a] ;
false

5. Scrieți un predicat care convertește din A1B2 (edge-clause) în A2B2 (neighbor list-clause).

```
edge_ex5(a,b).
edge_ex5(a,c).
edge_ex5(b,d).
```

?- retractall(gen_neighb(_,_)), edge_to_neighb, listing(gen_neighb/2).

:- dynamic gen_neighb_list/2.

```
gen_neighb(c, []).
gen_neighb(d, []).
gen_neighb(a, [c, b]).
gen_neighb(b, [d]).
true
```

6. Predicatul *restricted_path/4* găsește un drum între nodul sursă și cel destinație și verifică dacă drumul găsit conține nodurile din lista de restricții. Acest predicat folosește recursivitate înainte, ordinea nodurilor trebuie inversată în ambele liste – lista de drum și de restricții. Motivați de ce această strategie nu este eficientă (urmăriți ce se întâmplă). Scrieți un predicat mai eficient care caută un drum restricționat între nodul sursă și cel destinație.

```
edge_ex6(a,b).
edge_ex6(b,c).
edge_ex6(a,c).
edge_ex6(c,d).
edge_ex6(b,d).
edge_ex6(d,e).
edge_ex6(e,a).
```

```
? - restricted_path_efficient(a, e, [c,d], P2).
P = [a, b, c, d, e];
P = [a, c, d, e];
false
```

7. Rescrieti *optimal_path/3* astfel încât să funcționeze pe grafuri ponderate: atașați o pondere pentru fiecare muchie din graf și calculați drumul de cost minim dintr-un nod sursă la un nod destinație.

   Predicatul *edge* va avea 3 parametri.

```
edge_ex7(a,c,7).
edge_ex7(a,b,10).
edge_ex7(c,d,3).
edge_ex7(b,e,1).
edge_ex7(d,e,2).
```

```
?- optimal_weighted_path(a, e, P).
P = [e, b, a]
```

8. Scrie o serie de predicate care să rezolve problema Lupul-Capra-Varza: “un fermier trebuie să mute în siguranță, de pe malul nordic pe malul sudic, un lup, o capră și o varză. În barcă încap maxim doi și unul dintre ei trebuie să fie fermierul. Dacă fermierul nu este pe același mal cu lupul și capra, lupul va mânca capra. Același lucru se întâmplă și cu varza și capra, capra va mânca varza.”

   Sugestii:

   - Puteți alege să codificați spațiul de stări ca instanțe a unei configurări ale celor 4 obiecte (Fermier, Lup, Capră, Varză): reprezentate ca o listă cu poziția celor 4 obiecte [Fermier, Lup, Capră, Varză] sau ca o structură complexă (ex. F-W-G-C, sau state(F,W,G,C)).
   - starea inițială este [n,n,n,n] (toți sunt în nord) și starea finală este [s,s,s,s] (toți au ajuns în sud); pentru reprezentarea folosind liste a stărilor (ex. dacă Fermierul trece lupul -> [s,s,n,n], această stare nu ar trebui să fie validă deoarece capra mănâncă varza).
   - la fiecare trecere Fermierul își schimba poziția (din „n” în „s” sau invers) și **cel mult** încă un participant (Lupul, Capra, Varza).
   - problema poate fi văzută ca o problema de căutare a drumului într-un graf.

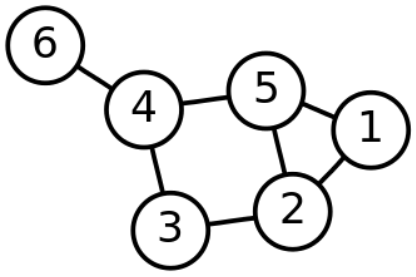
L11. Algoritmi de parcurgere a grafurilor (DFS și BFS)

1 Obiective

În lucrarea curentă vom prezenta cum se pot parcurge grafurile în Prolog, atât în strategia de parcurgere în lățime, cat și în adâncime; vom presupune reprezentarea *edge-clause*, prin predicatul *edge*/2.

2 Considerații teoretice

Avem mai jos un exemplu de graf neorientat, și o posibilă reprezentare a lui în Prolog:




```
edge(1,2).
edge(1,5).
edge(2,3).
edge(2,5).
edge(3,4).
edge(4,5).
edge(4,6).

is_edge(X,Y):- edge(X,Y);edge(Y,X).
```

2.1 Parcurgere în adâncime (DFS)

Ca în lucrarea anterioară, vom folosi forma *edge-clause* de prezentare a grafurilor. Deoarece parcurgerea în adâncime este un mecanism al Prolog-ului, toate predicatele drumurilor din lucrarea anterioară deja folosesc o strategie DFS de căutare. Mai jos aveți predicatul care implementează căutarea DFS dintr-un nod sursă (prin explorarea componentelor conexe a nodului sursă). Ne vom folosi de un predicat auxiliar pentru a stoca nodurile deja vizitate.

```
:- dynamic nod_vizitat/1.

% dfs(Source, Path)
dfs(X,_) :- df_search(X). % parcurgerea nodurilor
% când parcurgerea se termină, începe colectarea
dfs(_,L) :- !, collect_reverse([], L). % colectarea rezultatelor
```

```
% predicatul de parcurgere
df_search(X):-
    % salvăm X ca nod vizitat
    asserta(nod_vizitat(X)),
    % luăm un prim edge de la X la Y, restul le vom găsi prin backtracking
    is_edge(X,Y),
    % verificăm daca acest Y a fost deja vizitat
    not(nod_vizitat(Y)),
    % dacă nu a fost -  de aceea avem nevoie de negare –
    % atunci vom continua parcurgerea prin mutarea nodului curent la Y
    df_search(Y).

% predicatul de colectare - colectarea se face în ordine inversă
collect_reverse(L, P):-
    % scoatem fiecare nod vizitat
    retract(nod_vizitat(X)), !,
    % îl adăugăm la lista ca primul element (ordine inversă)
    collect_reverse([X|L], P).
    % unificăm primul si al doilea argument,
    % rezultatul va fi în al doilea argument
collect_reverse(L,L).
```

Urmărește execuția apelului:
?- dfs(1,R).
R = [1, 2, 3, 4, 5, 6].

2.2 Parcurgerea în lățime (BFS)

Strategia BFS se folosește de o coadă pentru a reține ordinea de expandare a nodurilor. La fiecare pas, un nou nod este citit din coadă și expandat – toți vecinii nevizitați vor fi adăugați în coadă. În soluția prezentată mai jos, coada este implementată folosind efecte laterale, prin combinația potrivită de *assert/retract.*

```
:- dynamic nod_vizitat/1.
:- dynamic coada/1.          % coada reține nodurile care trebuie expandate

% bfs(Source, Path)
bfs(X, _):- % parcurgerea nodurilor
    assertz(nod_vizitat(X)), % adăugăm sursa ca nod vizitat
    assertz(coada(X)), % adăugăm sursa în coadă
    bf_search.
```

```
bfs(_,R):-
    !,
    collect_reverse([],R). % colectarea rezultatelor

bf_search:-
   retract(coada(X)), % scoatem nodul care trebuie expandat
   !,
   expand(X), % apelăm predicatul de expansiune
   bf_search. % recursivitate

expand(X):-
   is_edge(X,Y), % găsim un nod Y cu o muchie la X-ul dat
   not(nod_vizitat(Y)), % verificăm daca Y a fost vizitat
   asserta(nod_vizitat(Y)), % adăugăm Y la nodurile vizitate
   assertz(coada(Y)), % adăugam Y în coadă pentru a fi expandat
   % la un moment dat
   fail. % fail-ul este necesar pentru a găsi un alt Y
expand(_).
```

Urmărește execuția apelului:
?- bfs(1,R).
R = [1, 2, 5, 3, 4, 6].

2.3 Best-First Search

Algoritmul *Best-First Search* utilizează o strategie de căutare greedy, care se folosește de o euristică pentru a estima costul unui drum de la nodul curent la nodul destinație. În fiecare pas, algoritmul selecționează nodul cu distanța estimată cea mai mică până la nodul destinație (folosind acea euristică).

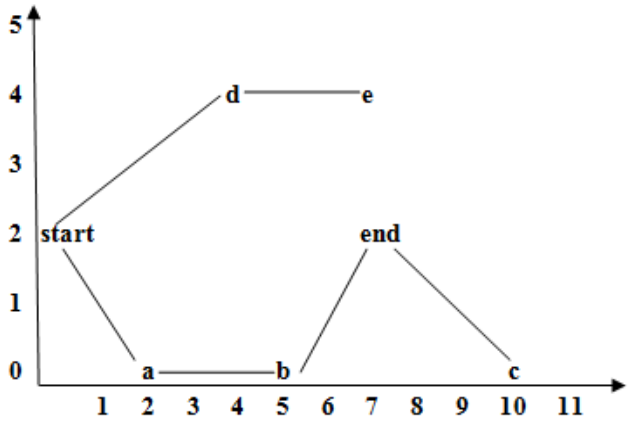


Figura 11.1: Un exemplu de graf pentru algoritmul Best-First Search

Graful de mai sus poate fi reprezentat folosind o variație a formei neighbor list-clause:

```
pos_vec(start,0,2,[a,d]).
pos_vec(a,2,0,[start,b]).
pos_vec(b,5,0,[a,c, end]).
pos_vec(c,10,0,[b, end]).
pos_vec(d,3,4,[start,e]).
pos_vec(e,7,4,[d]).
pos_vec(end,7,2,[b,c]).

is_target(end).
```

Nodul de final este specificat ca nodul target, folosind o clauză de predicat. Specificațiile predicatului sunt prezentate mai jos:

```
best([], []):-!.
best([[Target|Rest]|_], [Target|Rest]):- is_target(Target),!.
best([[H|T]|Rest], Best):-
        pos_vec(H,_,_, Neighb),
        expand(Neighb, [H|T], Rest, Exp),
        quick_sort(Exp, SortExp, []),
        best(SortExp, Best).

% Bazat pe calea curentă (al doilea argument), predicatul expand/4
% caută prin vecinii ultimului nod expandat (primul argument)
expand([],_,Exp,Exp):- !.
expand([H|T],Path,Rest,Exp):-
        \+(member(H,Path)), !, expand(T,Path,[[H|Path]|Rest],Exp).
expand([_|T],Path,Rest,Exp):- expand(T,Path,Rest,Exp).

% Predicatul quick_sort/3 utilizează liste diferență
quick_sort([H|T],S,E):-
        partition(H,T,A,B),
        quick_sort(A,S,[H|Y]),
        quick_sort(B,Y,E).
quick_sort([],S,S).

% În acest caz, predicatul partition/4 folosește un predicat auxiliar
% order/2 care definește modul de a partiționa ca fiind
% bazat pe distanțte
```

```
partition(H,[A|X],[A|Y],Z):- order(A,H), !, partition(H,X,Y,Z).
partition(H,[A|X],Y,[A|Z]):- partition(H,X,Y,Z).
partition(_,[],[],[]).

% predicat care calculează distanța între două noduri
dist(Node1,Node2,Dist):-
        pos_vec(Node1, X1, Y1, _),
        pos_vec(Node2, X2, Y2, _),
        Dist is (X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2).

% predicatul order/2 bazat pe distanțe folosit in partition/4
order([Node1|_],[Node2|_]):-
        is_target(Target),
        dist(Node1,Target,Dist1),
        dist(Node2,Target,Dist2),
        Dist1<Dist2.
```

Urmărește execuția apelului:
?- best([[start]], Best).

3 Exerciții

1. Modificați predicatul DFS astfel încât să caute noduri numai până la o anumită adâncime (DLS – Depth-Limited Search). Setați limita de adâncime printr-un predicat, depth_max(2). de exemplu.

```
edge_ex1(a,b).
edge_ex1(a,c).
edge_ex1(b,d).
edge_ex1(d,e).
edge_ex1(c,f).
edge_ex1(e,g).
edge_ex1(f,h).
```

?- dfs(a,DFS), dls(a, DLS).
DFS = [a, b, d, e, g, c, f, h],
DLS = [a, b, d, c, f].

2. Având următoarea secvență de cod în Prolog care implementează algoritmul BFS fără efecte laterale:

a. Modificați astfel încât să nu necesite utilizarea predicatului *reverse* (spre exemplu, prin folosirea listelor diferență).
b. Modificați astfel încât să funcționeze pe reprezentarea de *edge* în loc de reprezentarea de *neighbor*.

```
?- bfs(a, R).
R = [a, b, c, d, e].
```

```
neighbor(a, [b,c]).
neighbor(b, [a,d]).
neighbor(c, [a,e]).
neighbor(d, [b]).
neighbor(e, [c]).

bfs1(X, R) :-
   bfs1([X], [], P), reverse(P, R).

bfs1([], V, V).
bfs1([X|Q], V, R):-
   \+member(X, V),
   neighbor(X, Ns),
   remove_visited(Ns, V, RemNs),
   append(Q, RemNs, NewQ),
   bfs1(NewQ, [X|V], R).

remove_visited([], _, []).
remove_visited([H|T], V, [H|R]):- \+member(H, V), !, remove_visited(T, V, R).
remove_visited([_|T], V, R):- remove_visited(T, V, R).
```

L12. Probleme recapitulative

În această lucrare vom recapitula, prin o serie de probleme propuse spre rezolvare, concepte legate de:

- Operații aritmetice
- Operații pe liste (complete, diferență, incomplete); recursivitate înainte si înapoi
- Taiere de backtracking
- Operații pe liste adânci
- Operații pe arbori (compleți, incompleți)
- Grafuri și efecte laterale (assert, retract)

1 Operații aritmetice

1. Calculați cel mai mare divizor comun a două numere.
   ?- cmmdc(15,25,R).
   R = 5.

2. Calculați cel mai mic multiplu comun a două numere.
   ?- cmmmc(15,25,R).
   R = 75.

3. Calculați divizorii unui număr natural.
   ?- divisor(15,R1), divisor(2,R2), divisor(1,R3), divisor(0,R4),divisor(6,R5).
   R1 = [1,3,5,15], R2 = [1,2], R3 = [1], R4 = alot, R5 = [1,2,3,6].

4. Convertiți un număr în binar (puterile lui 2 cresc de la dreapta la stânga)
   ?- to_binary(5,R1),to_binary(8,R2),to_binary(11,R3).
   R1 = [1,0,1], R2 = [1,0,0,0], R3 = [1,0,1,1].

5. Inversați un număr natural.
   ?- reverse(15,R1), reverse(121235124,R2).
   R1 = 51, R2 = 421542121.

2 Operații pe liste

6. Calculați suma elementelor unei liste.
   ?- sum([1,2,3,4,5], R).

R = 15.

7. Dublați elementele impare și ridicați la pătrat cele pare.
   ?- numbers([2,5,3,1,1,5,4,2,6],R).
   R = [4,10,6,2,2,10,16,4,36].

8. Extrageți, dintr-o lista de întregi, numerele pare în E și numerele impare în O.
   ?- separate_parity([1,2,3,4,5,6], E, O).
   E = [2,4,6], O=[1,3,5].

9. Înlocuiți toate aparițiile lui X cu Y dintr-o lista completa.
   ?- replace_all(1, a, [1,2,3,1,2], R).
   R = [a,2,3,a,2].

10. Înlocuiți toate aparițiile lui of X într-o listă diferență (al doilea si al treilea argument) cu secvența [Y,X,Y].
    % replace_all(X, S, E, Y, R), where the difference list is S-E = [1,2,3,4,2,1,2]
    ?- replace_all(2,[1,**2**,3,4,**2**,1,**2**,2,3],[2,3],8,R).
    R = [1,**8,2,8**,3,4,**8,2,8**,1,**8,2,8**].

11. Ștergeți aparițiile lui X pe poziții pare (numerotatea poziției începe de la 1).
    ?- delete_pos_even([1,2,3,4,2,3,3,2,5],2,R).
    R = [1,3,4,2,3,3,5].

12. Ștergeți elementele de pe poziții divizibile cu K.
    ?- delete_kth([6,5,4,3,2,1], 3, R).
    R = [6,5,3,2].

13. Ștergeți elementele de pe poziții divizibile cu K de la finalul listei.
    ?- delete_kth_end([1,2,3,4,5,6,7,8,9,10],3,R)
    R = [1,3,4,6,7,9,10].

14. Ștergeți toate aparițiile elementului minim/maxim dintr-o listă.
    ?- delete_min([4,5,1,2], R).

R = [4,5,2].

15. Ștergeți elementele duplicate dintr-o listă (păstrează prima sau ultima apariție).
    ?- delete_duplicate([3,4,5,3,2,4], R).
    R = [3,4,5,2]. sau R = [5,3,2,4].

16. Inversează o listă incompletă.
    ?- reverse([1, 2, 3, 4, 5|_], R).
    R = [5, 4, 3, 2, 1|_].

17. Inversați elementele dintr-o lista după poziția K.
    ?- reverse_k([1,2,3,4,5,6], 2, R).
    R = [1,2,6,5,4,3].

18. Codificați o listă cu RLE (Run-length encoding). Elemente consecutive se înlocuiesc cu *(element, nr_apariții)*.
    ?- rle_encode([a,a,a,a,b,c,c,a,a,d,e,e,e,e], R).
    R = [[a,4], [b,1] ,[c,2], [a,2], [d,1] , [e,4]].

19. Codificați o listă cu RLE (Run-length encoding). Două sau mai multe elemente consecutive se înlocuiesc cu *(element, nr_apariții)*. Dar dacă numărul de apariții este egal cu 1 atunci se scrie doar elementul.
    ?- rle_encode1([1,1,1,2,3,3,4,4], R).
    R = [(1,3), 2, (3,2), (4,2)].

20. Decodificați o listă cu RLE (Run-length encoding).
    ?- rle_decode([[a,4], [b,1] ,[c,2], [a,2], [d,1] , [e,4]],R).
    R = [a,a,a,a,b,c,c,a,a,d,e,e,e,e].

21. Rotiți lista K poziții în dreapta.
    ?- rotate_k([1,2,3,4,5,6|_], 2, R).
    R = [5,6,1,2,3,4|_].

22. Sortați o listă de caractere în funcție de codul ASCII.
    ?- sort_chars([e, t, a, v, f], R).
    R = [a, e, f, t, v].

23. Sortați o listă de liste în funcție de lungimea listelor de nivel 2.
    ?- sort_len([[a, b, c], [f], [2, 3, 1, 2], [], [4, 4]], R).
    R = [[], [f], [4, 4], [a, b, c], [2, 3, 1, 2]].

24. Stergeți elementele duplicate de pe poziții impare dintr-o listă (indecșii încep de la 1).
    ?- remove_dup_on_odd_pos([1,2,3,1,3,3,3,9,10,6,10,8,7,3],R).
    R = [2,1,3,9,6,8,7,3].

3 Operații pe liste adânci

25. Calculați adâncimea maximă a unei liste imbricate.
    ?- depth_list([1, [2, [3]], [4]], R1), depth_list([], R2).
    R1 = 3, R2 = 1.

26. Aplatizați o listă imbricată cu liste complete/incomplete.
    ?- flatten([[1|_], 2, [3, [4, 5|_]|_]|_], R).
    R = [1,2,3,4,5|_].

27. Aplatizați doar elementele de la o adâncime dată într-o listă imbricată.
    ?- flatten_only_depth([[1,5,2,4],[1,[4,2],[5,[6,7,8]]],[4,[7]],8,[11]],3,R).
    R = [4,2,5,7].

28. Calculați suma elementelor de la nivelul K intr-o lista imbricată.
    ?- sum_k([1, [2, [3|_]|_], [4|_]|_], 2, R).
    R = 6.

29. Calculați numărul de liste într-o listă imbricată.
    ?- count_lists([[1,5,2,4],[1,[4,2],[5]],[4,[7]],8,[11]],R).
    R = 8.

30. Înlocuiți toate aparițiile lui X cu Y în lista imbricată.
    ?- replace_all_deep(2, 5, [[1, [2, [3, 2]], [4]], R).
    R = [1, [5, [3, 5]], [4]].

31. Înlocuiți fiecare secvență cu o adâncime constantă cu lungimea într-o listă adâncă.

?- len_con_depth([[1,2,3],[2],[2,[2,3,1],5],3,1],R).
R = [[3],[1],[1,[3],1],2].

4 Operații pe arbori

32. Calculați adâncimea unui arbore binar complet/incomplet.

tree(t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil))).
?- tree(T), depth_tree(T, R).
R = 3.

33. Colectați toate nodurile unui arbore binar complet/incomplet în inordine folosind liste complete.

tree(t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil))).
?- tree(T), inorder(T, R).
R = [2,4,5,6,7,9].

34. Colectați toate frunzele dintr-un arbore binar.

tree(t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil))).
?- tree(T), collect_k(T, R).
R = [2,5,7].

35. Scrieți un predicat care verifică dacă un arbore este arbore binar de căutare.

tree(t(3, t(2, t(1, nil, nil), t(4, nil, nil)), t(5, nil, nil))).
?- tree(T), is_bst(T).
false.

36. Arbore binar imcomplet. Colectați nodurile impare cu un singur copil într-o listă incompletă.

tree(t(26,t(14,t(2,_,_),t(15,_,_)),t(50,t(35,t(29,_,_),_),t(51,_,t(58,_,_))))).
?- tree(X), collect_odd_from_1child(X,R).
R = [35, 51|_].

37. Arbore ternar incomplet. Colectați cheile între X și Y (interval închis) într-o listă diferență.

tree(t(2,t(8,_,_,_),t(3,_,_,t(4,_,_,_)),t(5,t(7,_,_,_),t(6,_,_,_),t(1,_,_,t(9,_,_,_))))).
?- tree(T), collect_between(T,2,7,R,[1,18]).
R = [2,3,4,5,6,7,1,18].

38. Arbore binar. Colectați cheile pare ale frunzelor într-o listă diferență.

tree(t(5,t(10,t(7,nil,nil),t(10,t(4,nil,nil),t(3,nil,t(2,nil,nil)))),t(16,nil,nil))).
?- tree(T), collect_even_from_leaf(T,R,[1]).
R = [4,2,16,1].

39. Înlocuiți elementul minim dintr-un arbore ternar incomplet cu rădăcina.

tree(t(2,t(8,_,_,_),t(3,_,_,t(1,_,_,_)),t(5,t(7,_,_,_),t(6,_,_,_),t(1,_,_,t(9,_,_,_))))).
?- tree(T), replace_min(T,R).
R =
t(2,t(8,_,_,_),t(3,_,_,t(2,_,_,_)),t(5,t(7,_,_,_),t(6,_,_,_),t(2,_,_,t(9,_,_,_)))).

40. Colectați toate nodurile de la adâncimea K dintr-un arbore binar.

tree(t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil))).
?- tree(T), collect_k(T, 2, R).
R = [4, 9].

41. Colectați toate nodurile de la adâncimi impare dintr-un arbore binar incomplet (rădăcina are adâncime 0).

tree(t(26,t(14,t(2,_,_),t(15,_,_)),t(50,t(35,t(29,_,_),_),t(51,_,t(58,_,_))))).
?- tree(X), collect_all_odd_depth(X,R).
R = [14,50,29,58].

42. Colectați subarborii cu rădăcini conținând valoarea mediană dintr-un arbore ternar incomplet.

Observație. Mediana este "mijlocul" listei sortate de chei.

```
tree(t(2,t(8,_,_,_),t(3,_,_,t(1,_,_,_)),t(5,t(7,_,_,_),t(5,_,_,_),t(1,_,_,t(9,_,_,
_))))).
?- tree(T), median(T,R).
R = [
        t(5,t(7,_,_,_),t(5,_,_,_),t(1,_,_,t(9,_,_,_)))),
        t(5,_,_,_)
].
```

43. Înlocuiți fiecare nod cu înalțimea într-un arbore binar incomplet (frunzele au înalțimea 0).

```
tree(t(2,t(4,t(5,_,_),t(7,_,_)),t(3,t(0,t(4,_,_),_),t(8,_,t(5,_,_))))).
?- tree(T), height_each(T,R).
R = tree(t(3,t(1,t(0,_,_),t(0,_,_)),t(2,t(1,t(0,_,_),_),t(1,_,t(0,_,_))))).
```

44. Scrieți un predicat care înlocuiește întregul subarbore al unui nod (cu o cheie dată ca argument) cu un singur nod care are cheia suma cheilor subarborelui acelui nod (dacă nu există un nod cu aceea cheie, rămâne neschimbat).

```
tree(t(14,t(6,t(4,nil,nil),t(12,t(10,nil,nil),nil)),t(17,t(16,nil,nil),t(20,nil,nil)))
).
?- tree(T), sum_subtree(T,6,R).
R = t(14,t(32,nil,nil),t(17,t(16,nil,nil),t(20,nil,nil)))).
```

5 Grafuri

45. Colectați toate nodurile unui graf reprezentat sub forma *edge-clause*.

```
edge(1,2).
edge(3,1).
edge(2,3).

?- collect(R).
R = [1,2,3].
```

46. Calculați gradul interior/exterior al fiecărui nod dintr-un graf folosind predicatul dinamic *info(Node, OutDegree, InDegree).*

edge(1,2). edge(2,1). edge(1,4). edge(1,3). edge(3,2).
=> info(1,3,1). info(2,1,2). info(3,1,1). info(4,0,1).

Referințe bibliografice

[1] Rodica Potolea, Programare Logică, Vol. 1, UTPress, 2007

[2] Camelia Lemnaru, Rodica Potolea, Logic Programming. A Hands-on Approach, UTPress, 2018, ISBN 978-606-737-292-2