UNIVERSITATEA TEHNICĂ CLUJ-NAPOCA

PROGRAMARE LOGICĂ

ÎNDRUMĂTOR DE LABORATOR

Autori:

Tudor Mureşan

Rodica Potolea

Eneia Todoran

Alin Dumitru Suciu

Cluj-Napoca, 1998

PREFAȚĂ

Prezentul Îndrumător prezintă un număr de 12 lucrări de laborator utilizate în cadrul disciplinei de Programare Logică. Scopul acestor lucrări este de a ghida formarea deprinderilor practice în specificarea predicativă a problemelor. Se urmăreşte însuşirea temeinică a semanticii declarative şi a semanticii procedurale a programelor logice, precum şi a tehnicilor de programare Prolog.

Pentru însuşirea semanticii procedurale şi a rezoluției SLD, se utilizează un model propriu de implementare aflat la baza unui interpretor Prolog didactic, cu facilități de trasare a structurilor şi a arborilor de deducție. Pentru restul lucrărilor se utilizează componentele Turbo Prolog şi SICStus Prolog. Este acoperită problematica algoritmilor determinişti şi nedeterminişti de prelucrare a listelor, arborilor şi grafurilor, prin utilizarea structurilor complete şi incomplete (terminate în variabilă logică), precum şi a listelor şi structurilor diferență. De asemenea se prezintă exemple de metaprogramare (interpretare şi generare de programe).

Îndrumătorul de laborator de Programare Logică se adresează studenților secției de calculatoare, pregătindu-i pentru aplicațiile ulterioare bazate pe utilizarea limbajului Prolog şi a formalismelor derivate din el.

Autorii

CUPRINS

Lucrarea 1: Elemente de bază în programarea logică............................ 7

Lucrarea 2 : Elemente de bază în programarea logică (2)..................... 11

Lucrarea 3 : Operații pe liste. Tăierea backtrackingului........................ 15

Lucrarea 4 : Operații pe liste (2).. 19

Lucrarea 5 : Sortare (metode directe).. 23

Lucrarea 6 : Metode avansate de sortare..… 27

Lucrarea 7 : Obiecte recursive. Obiecte incomplete............................. 30

Lucrarea 8 : Arbori AVL... 34

Lucrarea 9 : Grafuri. Drumuri în graf.. 40

Lucrarea 10 : Strategii de căutare în graf.. 43

Lucrarea 11 : Determinarea componentelor biconexe ale unui graf....... 47

Lucrarea 12 : Metaprogramare.. 50

Bibliografie.. 54

LUCRAREA DE LABORATOR NR. 1

ELEMENTE DE BAZĂ ÎN PROGRAMAREA LOGICĂ

1. Scopul lucrării

Lucrarea de față introduce sintaxa, structura şi semantica operațională a limbajului Prolog. În acest cadru sunt introduse noțiunile de: predicat, clauză, regulă, fapt, unificare. Sunt introduse tipurile de date elementare.

2. Considerații teoretice

Ideea programării logice este de a utiliza formalisme cum sunt calculul propozițiilor şi calculul predicatelor direct drept limbaje de programe. Avantajul unui asemenea sistem de calcul constă în naturalețea exprimării problemelor din lumea reală precum şi în posibilitatea interpretării logice a programelor.

Prolog este poate cel mai răspândit limbaj de programare logică. El are la bază calculul predicatelor de ordinul întâi (numele de predicate sunt întotdeauna constante). În acest sens Prolog este un **limbaj declarativ** (limbaj de specificare predicativă) spre deosebire de limbaje ca Pascal şi C (care se zic imperative).

Un predicat este în esență o relație n-ară între argumentele sale. Prolog utilizează **forma clauzală Horn** pentru definirea predicatelor (o clauză este reprezentată ca o implicație a cărei consecință (cap de regulă) constă dintr-un singur predicat). Un predicat este definit printr-o colecție de clauze Horn în formatul general:

p(...) if p_{11}(...) and p_{12} and ... and p_{1n}(...).
...
p(...) if p_{m1}(...) and p_{m2} and ... and p_{mk}(...).

Sintactic există o singură categorie: **termenul** (care în esență este un arbore). Un termen poate fi:

- un **număr** (ex.: 12; 14.171 etc.),
- un **simbol** (ex.: ion),
- un nume de **variabilă** (ex.: X; un identificator care începe cu literă mare denotă o variabilă în timp ce un identificator care începe cu literă mică denotă un simbol constant)
- o **structură** (ex.: persoana(ion,21,adr(baritiu,X))).

Trebuie remarcat aici că Prologul standard este un limbaj **slab tipizat**. Nu există declarații de tip pentru variabile iar verificările semantice sunt amânate până la execuție. Există o orientare spre tipizare în sistemele moderne de programare logică. Pentru a se prelua o parte din verificările semantice în faza de compilare sistemul Turbo Prolog al firmei Borland a introdus declarațiile de domenii şi declarațiile de antet pentru predicate.

Predicatele pot fi legate prin conectivele logice:

- **if** (implicație),
- **and** (conjuncție)
- **or** (disjuncție).

Acestea admit o notație echivalentă care este de obicei preferată: "**:-**" (pentru if), "**,**" (pentru and) şi "**;**" (pentru or). Antecedentul unei implicații (corpul clauzei) este opțional. O clauză fără corp se numeşte **fapt**, în timp ce o clauză cu corp se numeşte **regulă**.

Următorul set de clauze reprezintă un program Prolog corect:

```
barbat(ion).
barbat(radu).
femeie(maria).
femeie(ana).
parinte(ion,radu).
parinte(ion,maria).
tata(X,Y) if barbat(X) and parinte(X,Y).
```

S-au specificat patru predicate (barbat, femeie, parinte şi tata) prin 7 clauze (şase fapte şi o regulă). Se remarcă faptul că clauzele ce definesc un predicat apar grupate.

Prolog este un limbaj interogativ. O întrebare/scop (engl. "goal") este de fapt o clauză fără cap (o întrebare poate fi şi o conjuncţie dar nu poate avea cap). În prezenta lucrare se va considera drept referinţă sistemul Prolog Edinburg în care (după introducerea programului de mai sus) poate avea loc dialogul:

```
?-tata(ion,Z).
DA
Z=radu
:
DA
Z=maria
:
NU
?-tata(ion,maria).
DA
```

Sistemul dă implicit primul răspuns la întrebare. Celelalte soluţii pot fi obţinute prin "repetarea întrebării" cu comanda "**:**". De remarcat faptul că ordinea în care sunt regăsite soluţiile la probleme este dată de ordinea în care apar clauzele în baza de date. Prolog foloseşte un mecanism rezolutiv (procesul de calcul este un arbore de deducţie) pentru a afla răspunsul la întrebare. Sistemul caută întotdeauna să determine valoarea de adevăr a întrebării puse. Mai sus, predicatul tata(ion,Z) este adevărat pentru Z=radu sau Z=maria şi este fals pentru orice altă valoare (interpretare) a lui Z. Răspunsurile la întrebare sunt găsite prin mecanismul de **backtracking** care este implicit în Prolog.

Transmiterea parametrilor se face prin **unificare** (prin potrivire; engl. "matching"; vezi curs). Acest mecanism implică două entităţi şi poate fi folosit pentru legarea variabilelor libere şi pentru teste logice. Eşuarea matchingului corespunde unui eşec de predicat. Eşecul determină revenirea (backtracking) la cel mai apropiat predicat în care au rămas alternative (clauze) neinspectate. Revenirea poate fi forţată şi prin apelul explicit al predicatului **fail** care corespunde constantei logice "fals" (negaţia este implementată ca eşec în Prolog).

Pentru scrierea unui program Prolog se pot avea în vedere analogiile (↔):

- cap regulă ↔ declarare (antet) procedură;
- corp regulă ↔ corp procedură;
- variabile ce apar în clauză ↔ date locale procedură;
- variabile ce apar în cap regulă ↔ parametri procedură;

Trebuie însă subliniat faptul că în Prolog nu există atribuire. Pentru fiecare valoare nouă trebuie creat un nume (variabilă) nou (în Prolog fiecare stare are un nume).

Stiva de execuție din limbajele procedurale devine arbore de execuție (care conține istoria procesării) în Prolog. Când se repetă întrebarea (sau când revenirea este forțată prin program) este necesar să se cunoască întreaga istorie a procesării pentru a se putea furniza următoarea soluție, deci arborele de execuție trebuie păstrat. La fiecare eşec se şterg nodurile (din arbore) în care nu mai există alternative şi se revine înspre rădăcină. Când avem un eşec total arborele de execuție este complet şters din memorie.

Conceptual, arborele de deducție este un arbore ŞI-SAU (vezi curs). Într-un asemenea arbore nivelurile ŞI alternează cu niveluri SAU. Un nod de pe un nivel SAU poate fi rezolvat dacă reprezintă un fapt sau (dacă reprezintă un cap de regulă) dacă toți descendenții săi (apeluri de predicate din corpul regulei) aflați pe un nivel ŞI pot fi rezolvați. Nivelurile SAU corespund colecțiilor de clauze care definesc predicatele. Un asemenea raționament trebuie să se "bizuie" pe fapte (periferia arborelui de deducție va consta fie din fapte, fie din eşecuri).

La nivel operațional nu se construieşte un întreg arbore ŞI-SAU ci, corespunzător fiecărei soluții, se vor construi în memorie numai nivelurile ŞI. Reconsiderarea unei alternative revine la ştergerea arborelui înspre rădăcină până la un punct de reluare.

Pentru programul Prolog discutat mai sus construim arborele de execuție corespunzător întrebării "?- tata(ion, Z).":

```
?-tata(ion,Z).  { X=ion; Y=Z }
  tata(X,Y)
      |
?-barbat(ion) ——— ?-parinte(ion,Z)  { Z=radu }
  barbat(ion)        parinte(ion,radu)
```

Între acolade "{...}" sunt prezentate legările (instanțierile) efectuate în urma unui matching reuşit (succes) între un scop sau un sub-scop (întrebare) şi un cap de regulă sau un fapt. În urma matchingului între întrebarea "?-tata(ion,Z)" şi capul de regulă "tata(X,Y)" s-au efectuat instanțierile "{ X=ion; Y=Z }". Rezolvarea apelului "tata(ion,Z)" revine la rezolvarea sub-scopurilor "?-barbat(ion)" şi "?-parinte(ion,Z)". Acestea succed direct prin suprapunerea peste faptele "barbat(ion)" şi "parinte(ion,radu)" cu legarea lui "Z" la "radu". Se remarcă faptul că după încheierea procesului rezolutiv trebuie executat un proces de "culegere" a soluției. Instanțierile de variabile efectuate în procesul rezolutiv sunt preluate pentru a se efectua legarea variabilelor din rădăcină (scop). O asemenea "legare" este realizată numai în urma unui succes. În urma unui eşec (local sau global) toate legările de variabile efectuate se pierd odată cu ştergerea arborelui sau sub-arborelui de execuție corespunzător.

Prezentăm mai jos o trasare echivalentă generată cu ajutorul şistemului Prolog Edinburg didactic aşa cum urmează a fi executată în cadrul şedintelor de laborator. Sistemul permite activarea sau inhibarea trasării cu ajutorul predicatului "trace(on/off)". Deasemenea, se poate utiliza predicatul "printstr(on/off)" pentru vizualizarea structurii interne a termenilor Prolog (arbori binari).

```
?-trace(on).
?-printstr(on).
?-tata(ion,Z).
.
     tata
     .
           Z
     .
           ion
?-
DA
?tata(ion,Z)#tata(B_1,C_1)
```

```
Z=C_1
B_1=ion C_1=radu
     ?barbat(B_1)#barbat(ion)
     B_1=ion C_1=radu
     ?parinte(B_1,C_1)#parinte(ion,radu)
     B_1=ion C_1=radu
Z=radu
```

Mai sus, matchingurile corespunzătoare nodurilor din arborele de execuție aflate pe acelaşi nivel (de exemplu în aceeaşi conjuncție) sunt egal indentate. Elementele (nume de variabile) din contextul interogării, respectiv din contextul sub-scopului sunt automat generate de sistem, iar indicii lor (i,i+1,..., pentru variabilele B_i, C_i, ...) cresc odată cu avansul în recursivitate (vezi şi exemplul de trasare pentru predicatul "append" de mai jos).

3. Desfăşurarea lucrării

Studenții vor testa predicatele prezentate pe diverse date de intrare. Se va utiliza sistemul Prolog Edinburg didactic în care se vor efectua de fiecare dată trasare şi afişarea arborelui de structură.

4. Întrebări şi probleme

4.1. Completați baza de date din primul exemplu discutat în lucrare cu faptele:

```
barbat(dan).
parinte(dan,ion).
```

şi scrieți un predicat care să stabilească relația de "bunic" între persoane. Construiți arborele de derivare pentru o întrebare care utilizează noul predicat.

LUCRAREA DE LABORATOR NR. 2

ELEMENTE DE BAZĂ ÎN PROGRAMAREA LOGICĂ (2)

1. Scopul lucrării

Lucrarea de față introduce noțiunile de backtracking şi recursivitate precum şi structura de listă. Este deasemenea prezentat procesul rezolutiv (privit ca instrument universal de calcul) care stă la baza semanticii operaționale Prolog.

2. Considerații teoretice

Exemplul din lucrarea anterioară reprezintă un **raționament deductiv**. Prolog însă permite şi efectuarea de **raționamente inductive**, exprimate prin definiții recursive de predicate. În fapt, **recursivitatea** este modul natural de exprimare a algoritmilor repetitivi în limbajele declarative. Putem de exemplu testa apartenența unui element la o listă cu ajutorul următorului predicat:

```
member(X,[X|_]).
member(X,[_|Y]):-member(X,Y).
```

În exemplul de mai sus este introdusă notația Prolog pentru listă. Notația pentru listă vidă (fără elemente) este "[]". O listă de numere este "[1,2,7,5]" iar o listă de simboli "[a,b,c]". Există o notație frecvent utilizată care subliniază caracterul recursiv al obiectelor de tip listă:

Lista=[PrimulElement|RestLista]

De exemplu:

```
?-L=[a,b,c],L=[E|R].
DA
L=[a,b,c]
E=a
R=[b,c]
```

Se remarcă faptul că R este o listă şi E este un element. Este admisă şi notația mai generală în care sunt evidențiate un număr arbitrar dar fix de elemente din capul unei liste. De exemplu:

```
?-L=[1,2,3,4],L=[E1,E2,E3|R].
DA
L=[1,2,3,4]
E1=1
E2=2
E3=3
R=[4]
```

Exemplele de mai sus au prezentat numai liste omogene (cu elemente de acelaşi tip). Prolog Edinburg este un limbaj slab tipizat şi admite şi liste eterogene cum ar fi "[a,2,[3,c],d]". Definiția predicatului "member" utilizează şi notația "_" care referă o entitate (variabilă) **indiferentă**, a carei valoare nu este necesară în calculele specificate de clauza în care apare. Ca

regulă generală, orice variabilă care apare o singură dată într-o clauză oarecare poate fi înlocuită cu notația "_".

Un avantaj al slab-tipizării constă în aceea că definiția de mai sus a predicatului member poate fi utilizată indiferent de tipul elementelor din listă.

```
?-member(1,[2,1,3]).
DA
?-member(a,[z,c,d,a,u]).
DA
```

Deoarece un predicat stabileşte o relație între argumentele sale el poate fi utilizat în diverse interpretări (şabloane de intrare/ieşire). De exemplu:

```
?-member(X,[a,b]).
DA
X=a
:
DA
X=b
:
NU
```

Comportamentul prezentat mai sus este nedeterminist. Interpretarea este următoarea: "X" este membru în lista "[a,b]" dacă "X=a" sau "X=b". Nedeterminismul este modelat în Prolog prin mecanismul implicit de backtracking. Studenții sunt invitați să construiască cei doi arbori de execuție corespunzători celor două valori posibile pentru X.

Atunci când este necesar Prolog "creează şablon" pentru datele structurate, utilizând dacă este cazul notația pentru element indiferent "_".

```
?-member(1,L).
DA
L=[1|_]
:
DA
L=[_,1|_]
...
```

Apelat ca mai sus, predicatul member returnează în principiu o infinitate de soluții creând şablon pentru variabila de ieşire care aici este chiar lista la care se testează apartenența elementului "1".

Pentru concatenarea a două liste se poate utiliza predicatul:

```
append([],A,A).
append([A|B],C,[A|D]):-append(B,C,D).
```

Arborele de execuție pentru un predicat definit recursiv reflectă raționamentul inductiv corespunzător, iar adâncimea lui depinde de datele de intrare. De exemplu:

```
?-append([a,b],[c,d],L).           { A1=a; B1=[b]; }
 append([A1|B1],C1,[A1|D1])  { C1=[c,d]; L=[a|D1]; }
        |
```

```
?-append([b],[c,d],D1)           { A2=b; B2=[]; }
  append([A2|B2],C2,[A2|D2]){ C2=[c,d]; D1=[b|D2]; }
        |
?-append([],[c,d],D2)            { A3=D2=[c,d]; }
  append([],A3,A3)
```

Pentru construirea unui asemenea arbore de execuție regula este ca într-un context dat (toate predicatele din partea stângă şi dreaptă a unei implicații care defineşte o clauză) să se folosească în mod consecvent aceleaşi variabile (şi deci aceiaşi indici). La schimbarea contextului se vor genera indici noi.

Din nou, la sfârşit este necesar un proces de "culegere a soluției" în urma căruia L=[a|D1]=[a,b|D2]=[a,b,c,d]. Deoarece afişarea arborelui se face după culegerea rezultatului în sistemul Prolog Edinburg didactic utilizat de studenți, variabilele de ieşire (L şi D1 în exemplul de mai sus) apar gata instanțiate. Astfel:

```
?-append([a,b],[c,d],L).
.
        append
        .
        L
        .
        .
        .
                d
        .
                c
        .
        .
        .
                b
        .
                a
?-
DA

?append([a,b],[c,d],L)#append([B_1|C_1],D_1,[B_1|E_1])
L=[B_1|E_1]
B_1=a C_1=[b] D_1=[c,d] E_1=[B_2|E_2]
        ?append(C_1,D_1,E_1)#append([B_2|C_2],D_2,[B_2|E_2])
        B_1=a C_1=[b] D_1=[c,d] E_1=[B_2|E_2]
        B_2=b C_2=[] D_2=[c,d] E_2=[c,d]
        ?append(C_2,D_2,E_2)#append([],B_3,B_3)
        B_2=b C_2=[] D_2=[c,d] E_2=[c,d]
                B_3=[c,d]

L=[a,b,c,d]
```

Pentru un predicat există în general atâtea interpretări câte şabloane de intrare/ieşire sunt posibile. Desigur, nu toate intepretările unui predicat sunt interesante pentru o anume aplicație, după cum unele pot fi de interes pur teoretic. Pentru predicatul "append" reținem însă utilizarea sa pentru concatenarea a două liste (prezentată mai sus). Acesta este un comportament

"funcţional", în sensul că pentru date de intrare stabilite (două liste arbitrare) valoarea datelor de ieşire este unic definită (concatenarea celor două liste în ordinea în care au fost prezentate la apel). Un predicat stabileste însă în general o relaţie n-ară între argumente (o funcţie fiind un caz particular de relaţie). O asemenea relaţie este construită în Prolog cu ajutorul mecanismului de backtracking care permite ca o anumită întrebare să fie interpretată în general conform mai multor clauze din definiţia unui predicat. O altă utilizare frecventă a predicatului "append" este pentru generarea "descompunerilor" unei liste.

```
?-append(X,Y,[a,b,c]).
DA
X=[]
Y=[a,b,c]
:
DA
X=[a]
Y=[b,c]
:
DA
X=[a,b]
Y=[c]
:
DA
X=[a,b,c]
Y=[]
:
NU
```

Dialogul de mai sus sugerează utilizarea predicatului "append" pentru alegerea nedeterministă a unui element dintr-o listă astfel:

```
?-append(_,[E|_],[a,b]).

DA
E=a
:
DA
E=b
:
NU
```

3. Desfăşurarea lucrării

Studenţii vor testa predicatele prezentate pe diverse date de intrare. Se va utiliza sistemul Prolog Edinburg didactic în care se vor efectua de fiecare dată trasare şi afişarea arborelui de structură.

4. Întrebări şi probleme

4.1. Să se construiască arborii de execuţie pentru diverse întrebări, utilizând predicatele "member" şi "append".

LUCRAREA DE LABORATOR NR. 3

OPERAȚII PE LISTE. TĂIEREA BACKTRACKINGULUI

1. Scopul lucrării

Lucrarea de față se adresează celor familiarizați deja cu reprezentarea internă. Se presupune de asemenea că este cunoscută sintaxa Prolog. Scopul lucrării este de a crea deprinderea scrierii de predicate care operează cu liste. Al doilea obiectiv este înțelegerea mecanismului tăierii backtrackingului.

2. Considerații teoretice

În prima parte dorim să implementăm câteva operații cu mulțimi. În continuare vom considera că mulțimile sunt reprezentate sub formă de liste (impunem deci restricția ca într-o listă toate elementele să fie distincte).

Pentru reuniunea a două mulțimi (reprezentate ca liste), în mulțimea rezultat se vor afla elementele comune şi necomune, o singură dată. Pentru aceasta, parcurgem prima listă (reprezentând mulțimea); dacă primul element al său se află şi în cea de-a doua listă (caz în care testul member(H,L) din corpul primei clauze se va termina cu succes), el nu se adaugă în rezultat, adăugarea făcându-se în bloc, pentru toate elementele celei de-a doua liste (clauza 3 a predicatului). Dacă elementul din capul primei liste nu se află în cea de-a doua listă (testul member(H.L) din corpul primei clauze va eşua, iar mecanismul de backtracking va reevalua acelaşi scop prin unificarea cu a doua clauză a predicatului. Este evident că testul not(member(H,L)) se va evalua în acest caz la adevărat (vezi întrebarea 1 de la punctul 4 al lucrării)), acesta se adaugă, ca fiind primul element al rezultatului. Evident, clauzele sunt recursive; deci după analiza unui element şi luarea deciziei corespunzătoare, analiza se continuă pe coada primei liste, şi aceeaşi a doua listă, în scopul obținerii restului rezultatului (R din al treilea argument). Continuarea analizei se face printr-un subscop în corpul clauzei corespunzătoare (reun(T,L,R)). Deoarece ambele clauze sunt recursive trebuie să existe cel puțin una nerecursivă (condiția de terminare a recursivității). Aceasta corespunde totodată şi cazului particular în care prima listă este vidă, caz în care al doilea argument se adaugă în întregime rezultatului. Este evidentă utilizarea argumentelor în acest predicat, reun(mulțime_1, mulțime_2,rezultat).

```
reun([H|T],L,R):-member(H,L), reun(T,L,R).
reun([H|T],L,[H|R]):-not(member(H,L)), reun(T,L,R).
reun([ ],L,L).
```

O interpretare asemănătoare poate fi făcută pentru alte două operații pe mulțimi: intersecția şi diferența.

Vor aparține mulțimii intersecție doar acele elemente care aparțin amânduror mulțimi argument. Analiza se face la fel, pentru primul element al primului argument, adăugându-l, respectiv neadăugându-l la rezultat şi apoi recursiv pe restul listei, printr-un subscop-apel recursiv în corpul clauzei. Clauza a treia, din nou, reprezintă terminarea recursivitații şi respectiv cazul particular în care primul argument este lista vidă, caz în care rezultatul nu conține nici un element. Argumentele sunt int(mulțime_1,mulțime_2,rezultatat).

```
int([H|T],L,[H|R]):-member(H,L), int(T,L,R).
int([H|T],L,R):-not(member(H,L)), int(T,L,R).
int([ ],L,[ ]).
```

În mulțimea diferență vor fi regăsite acele elemente care aparțin primei mulțimi fără să aparțină celei de-a doua. Argumentele au aceeaşi semnificație ca la precedentele predicate.

```
dif([H|T],L,R):-member(H,L), dif(T,L,R).
dif([H|T],L,[H|R]):-not(member(H,L)), dif(T,L,R).
dif([ ],L,[ ]).
```

Dorim să realizăm acum un predicat care şterge un element dint-o listă (revenim la accepțiunea obişnuită a listelor; ele nu mai reprezintă mulțimi). În cazul în care elementul apare de mai multe ori în listă, dorim să se şteargă o singură apariție a sa. Pentru aceasta, va trebui să comparăm elementul de eliminat cu elementul curent al listei: dacă acestea coincid, rezultatul va fi format din restul listei de intrare (clauza 1). Dacă nu, elementul este lăsat în rezultat, coada listei rezultat generându-se la apelul recursiv din corpul clauzei (2). Condiția de terminare corespunde listei de intrare vidă.

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

La rularea predicatului, la întrebarea:

```
?-del(1,[1,2,1,,3,1],R),
```

obținem rezultatul:

```
DA
R=[2,1,3,1]
```

iar prin repetarea întrebarii obținem respectiv următoarele răspunsuri:

```
DA
R=[1,2,3,1]

DA
R=[1,2,1,3,]
DA
R=[1,2,1,3,1]
NU
```

Pentru a obține un singur răspuns, predicatul trebuie să fie determinist. În cazul nostru, nedeterminismul rezultă din faptul că în a doua clauză, deşi elementul de şters (X) şi capul listei (H) sunt reprezentate prin variabile distincte, ele se pot unifica (unificare explicită şi obligatorie în prima clauză). Comportamentul celor două clauze este însă diferit, de unde şi nedeterminismul predicatului: în primul caz, cel al unificării explicite, elementul nu se adaugă în rezultat, terminându-se prelucrarea (fără apel recursiv), în cel de-al doilea, elementul se adaugă rezultatului, continuându-se prelucrarea (apelul recursiv). Avem deci de a face cu două comportamente distincte pentru o aceeaşi situație.

O modalitate de a transforma un predicat nedeterminist într-unul determinist este tăierea backtrackingului. Să ne reamintim că atunci când un scop q se unifică cu o clauza de forma:

$q(\ ):-b_1\ (\ ),b_2(\ ),...,b_{n-1}(\ ),!,b_n(\ ),...$

operatorul de tăierea backtrackingului (cut, !) este efectiv doar dacă unificarea scopului cu capul regulii s-a făcut cu succes şi de asemenea toate subscopurile din corpul regulii aflate la stânga operatorului s-au executat cu succes (adică b_1 ( ),b_2( ),...,b_{n-1}( )). În acest caz, instanţierile făcute nu mai pot fi anulate prin backtracking decât ca un tot unitar. Subscopurile aflate la dreapta operatorului nu sunt afectate: ele fac backtracking în mod obişnuit. Când nici unul din aceste subscopuri nu mai furnizează soluţii, se revine (în lanţul de scopuri) la subscopul anterior lui q.

Deci tăierea (!) afectează:

- subscopurile din corpul clauzei aflate la stânga operatorului (o conjuncţie de scopuri urmată de tăiere va produce cel mult o soluţie)
- toate clauzele aflate după clauza curentă (un scop ce se unifică cu o clauză conţinând !, nu va mai putea furniza soluţii utilizând clauzele următoare)

şi nu afectează:

- subscopurile din corpul clauzei aflate la dreapta operatorului
- clauzele aflate înaintea celei curente.

În arborele de execuţie, odată ce nodul corespunzător tăierii de backtracking a fost generat, nici unul din nodurile aflate pe acelaşi nivel la stânga (împreună cu întregi subarborii lor), şi nici nodul părinte nu mai fac backtracking. Primul nod care mai poate face backtracking este nodul aflat imediat la stânga nodului părinte şi pe acelaşi nivel cu el.

În cazul specificat, tăierea de backtracking afectează scopul q şi fiecare din scopurile b_1 ( ),b_2( ),...,b_{n-1}( ), şi nu afectează nimic altceva.

Dacă unificarea lui q cu clauza curentă eşuează din cauza unui matching nereuşit cu capul clauzei sau din cauza unui eşec la unul din subscopurile din corp aflate la stânga operatorului ! , acesta devine inefectiv.

Utilizând operatorul de tăierea backtrackingului pentru modificarea predicatului del obţinem:

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

Semantica lui s-a modificat. Devenind un predicat determinist, la întrebarea anterioară obţinem o singură soluţie, după care, la repetarea ei, răspunsul este NU.

Explicaţia este următoarea: atâta timp cât elementul de şters nu se află pe prima poziţie în listă, prima clauză nu poate fi utilizată pentru matching. Matchingul cu a doua reuşeşte însă, şi se avansează recursiv în coada listei. Când elementul de şters coincide cu capul listei, unificarea cu prima clauză devine posibilă. Dar în corpul acestei clauze, operatorul de tăiere a backtrackingului ne asigură ca ultimul scop (din lanţul executat până în acest punct) nu se va mai unifica cu nici o altă clauză. Cum toate scopurile anterioare au fost deterministe, înseamnă că la întrebarea iniţială vom obţine un singur răspuns şi o singură soluţie. Deci tăierea backtrackingului din corpul primei clauze "impune" o condiţie implicită clauza/clauzele următoare (în general negatul condiţiei ce apare în clauza ce conţine tăierea de backtracking). În cazul nostru condiţia ce se impune implicit este X<>H.

În general pentru un predicat ce conţine clauzele:

```
p:-c,!,b1.
p:-b2.
```

interpretarea este:

```
p:-c,b1.
p:-not(c),b2.
```

Folosind cele mai sus menționate, predicatul de substituție a unui element dintr-o listă cu un alt element specificat este:

```
subst(el_nou,el_de-subst,lista_veche,lista_noua)

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

iar dacă se doreşte o singură soluție,

```
subst(X,H,[H|T],[X|T]):-!.
subst(X,Y,[H|T],[H|R]):-subst(X,Y,T,R).
subst(X,Y,[ ],[ ]).
```

3. Desfăşurarea lucrării

Se vor testa toate predicatele menționate, analizând caracterul lor determinist/nedeterminist. Se vor construi arborii de execuție corespunzători diferitelor întrebări.

4. Intrebări şi probleme

4.1. Ce se întâmplă dacă în predicatele de reuniune, intersecție şi diferență testul not(member(H,L)) din corpul clauzei a doua se elimină? Nu este el redundant (din moment ce în prima clauză testul member(H,L) a eşuat, deoarece nici o altă condiție de test nu mai diferențiază cele două clauze, nu este implicit adevărat not(member(H,L)))?

4.2. Modificați predicatele de reuniune, intersecție şi diferență utilizând tăierea de backtracking. Comparați eficiența celor două soluții.

4.3. Modificați predicatul member astfel încât să devină determinist.

4.4. Comparați rezultatele obținute la întrebarea:

```
?-member(X,[1,2,3]).
```

rulând cu predicatul member modificat, respectiv nemodificat.

LUCRAREA DE LABORATOR NR. 4

OPERAȚII PE LISTE (2)

1.Scopul lucrării

Această lucrare continuă prezentarea de predicate de lucru pe liste. Se are în vedere şi eficiența specificărilor predicative prezentate.

2.Considerații teoretice

2.1. Inversarea unei liste

Uneori algoritmul prin natura lui furnizează soluția sub forma unei liste (vezi de exemplu algoritmii de traversare graf din lucrările 8 şi 9). Deoarece accesul la elementele unei liste în Prolog se face numai printr-un capăt conform definiției `Lista=[Element|RestLista]` accesarea în ordine inversă presupune inversarea listei.

O soluție imediată pentru inversarea unei liste se poate obține prin utilizarea predicatului 'append', prezentat în lucrarea 2.

```
inv([],[]).
inv([A|B],R):-inv(B,C),append(C,[A],R).
```

Prima clauză exprimă faptul că: 'inversa listei vide este lista vidă'. Cea de a doua clauză spune că 'inversa unei liste nevide se obține prin adăugarea primului său element în coada restului inversat'. Soluția este naturală însă pentru sporirea eficienței vom dori să evităm utilizarea predicatului 'append'.

Inversarea unei liste se poate face prin utilizarea unui parametru de acumulare. Ideea este sugerată prin secvența de rescriere de mai jos:

```
{[1,2,3],[],?} => {[2,3],[1],?} => {[3],[2,1],?} => {[],[3,2,1],[3,2,1]}
```

Exprimarea predicativă a acestui proces de calcul este:

```
/* inv1( Lista {i}, ParamAcumulare {i}, ListaInv {o} ) */
inv1([],R,R).
inv1([A|B],T,R):-inv1(B,[A|T],R).
```

Prima clauză reprezintă condiția de terminare a recursivității şi permite raportarea soluției prin cel de al treilea parametru. Inițial, parametrul de acumulare trebuie să fie lista vidă. Un exemplu de apel ar fi:

```
?-inv1([1,2,3],[],L).
DA
L=[3,2,1]
```

Dacă parametrul de acumulare este o listă oarecare, aceasta se va regăsi nemodificată în rezultat:

```
?-inv1([1,2,3],[7,8],L).
DA
L=[3,2,1,7,8]
```

Predicatul 'append' permite alegerea unui element 'arbitrar' dintr-o listă astfel:

```
append(_,[Element|_],Lista)
```

Deoarece această alegere se face prin mecanismul de backtracking al Prolog, în care clauzele sunt inspectate în ordinea în care apar în program, elementele sunt raportate în ordinea în care apar în listă. Prin inversarea clauzelor predicatului 'append':

```
append([],L,L).
append([A|B],C,[A|D]):-append(B,C,D).
```

elementele pot fi raportate în ordine inversă astfel:

```
?-append(_,[E|_],[1,2,3]).
DA
E=3
:
DA
E=2
:
DA
E=1
:
NU
```

Această soluție permite regăsirea elementelor din listă prin forțarea eşecului predicatului 'append'. Predicatele 'inv' şi 'inv1' furnizează lista inversată. Aceasta permite apoi regăsirea recursivă a elementelor unei liste.

2.2. Determinarea elementului minim (maxim) al unei liste

Această problemă presupune existența unei relații de ordine între elementele listei. În cele ce urmează vom considera liste de întregi cu relația de ordine cunoscută. Determinarea elementului minim implică parcurgerea întregii liste. Deasemenea este necesară memorarea unui minim parțial, şi actualizarea sa când este cazul. Deoarece repetiția se face în Prolog prin recursivitate minimul parțial va fi transmis ca şi parametru. Minimul parțial se poate inițializa cu valoarea primului element al listei. O astfel de soluție este prezentata mai jos:

```
minl([X|Y],Min):-min(Y,X,Min).

min([],M,M).
min([X|Y],M,Min):-X<M,min(Y,X,Min).
min([X|Y],M,Min):-X>=M,min(Y,M,Min).
```

Predicatul 'minl' inițializează minimul parțial la primul element al listei. Prima clauză a predicatului 'min' reprezintă condiția de terminare a recursivității şi permite raportarea soluției. Celelalte două clauze permit tratarea celor două tipuri de monotonii (crescătoare/descrescătoare) din cadrul listei. O primă îmbunătățire a acestei soluții se poate face pe baza unei observații generale. Se poate utiliza tăierea de backtracking pentru a evita inspectarea tuturor gărzilor unui predicat. În exemplul nostru ultimele două clauze ale predicatului 'min' se pot scrie astfel:

```
min([X|Y],M,Min):-X<M,!,min(Y,X,Min).
min([X|Y],M,Min):-min(Y,M,Min).
```

Se sporeşte atât concizia în exprimare cât şi viteza, deoarece dacă prima gardă 'X<M' succede predicatul '!' efectuează tăierea de backtracking şi nu se mai inspectează următoarea clauză. Determinismul poate fi forțat prin gărzi disjuncte sau prin utilizarea tăierii de backtracking.

Raţionamentul prezentat mai sus se bazează pe utilizarea unui parametru suplimentar cu semantică de minim parţial. Vom prezenta mai jos un raţionament superior atât prin expresivitate cât şi prin eficienţă conform semanticii operaţionale Prolog.

```
/* min1( Lista {i}, Minim {o} ) */

min1([H|T],M):-min1(T,MT),MT<H,M=MT.
min1([H|T],M):-min1(T,MT),MT>=H,M=H.
min1([H],H).
```

La nivelul interpretării predicative soluţia de mai sus exprimă faptul că minimul unei liste este minimul restului listei dacă acesta este 'mai mic' decât elementul din capul listei, altfel coincide cu elementul din capul listei.

Noutatea acestui tip de raţionament (faţă de exemplele prezentate) constă în utilizarea unui apel recursiv a unui predicat în gărzile din propria definiţie. Acum putem, pe baza mecanismului cunoscut să eliminăm exprimarea explicită a celei de a doua gărzi prin utilizarea tăierii de backtracking. Deasemenea, odată eliminată cea de a doua gardă se poate elimina şi ultima clauză deoarece cazul listei de lungime unu este acum acoperit de cea de a doua clauză.

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

Acest raţionament tratează minimul parţial prin utilizarea mecanismului de negaţie ca eşec din Prolog. Eşuarea unui predicat conduce la anularea legărilor de variabile realizate de acel predicat. Se leagă minimul parţial mai întâi la ultimul element din listă (atins prin parcurgerea recursivă a listei). Pe revenirea din recursivitate această legare se pierde (în caz de eşec <=> violare a condiţiei de minim) actualizându-se cu elementul curent.

O generalizare imediată a acestui predicat este un predicat de eliminare a elementului minim dintr-o listă. Acest predicat va fi utilizat în algoritmul de sortare prin selecţie (vezi lucrarea 5).

```
/* elimmin( Lista {i}, Minim {o}, RestLista {o} ) */
elimmin([H|T],H1,[H|R]):-elimmin(T,H1,R),H1<H,!.
elimmin([H|R],H,R).
```

De remarcat că o singură apariţie a elementului minim va fi eliminată din listă. Lăsăm cititorul să precizeze şi să justifice care anume.

2.3. Generarea permutărilor unei mulţimi

Vom reprezenta mulţimea ca listă. Elementele listei vor fi considerate distincte câte 2 (chiar dacă au atribute comune sau au aceeeaşi valoare). Se ştie că numărul de permutări ale unei mulţimi de N elemente este N!. Algoritmul evidenţiază puterea nedeterminismului ca şi tehnica de concepere a algoritmilor. De fapt, puterea de expresie a limbajului Prolog rezidă din două elemente: unificare şi nedeterminism. Algoritmul de generare a permutărilor constă în selecţia unui element arbitrar din mulţime şi depunerea sa în faţa unei permutări a restului mulţimii. Aceasta se exprimă cu uşurinţă cu ajutorul predicatului 'append'.

```
/* permutari( Multime {i}, Permutare {o} ) */
permutari(L,[H|T]):-
      append(V,[H|U],L),
      append(V,U,W),permutari(W,T).
permutari([],[]).
```

În primul apel predicatul 'append' este folosit pentru alegerea unui element arbitrar al listei L, iar în cel de-al doilea apel pentru obținerea listei rest.

Această soluție poate fi îmbunătățită. Cele două apeluri ale predicatului 'append' pot fi înlocuite cu apelul unui predicat care să combine cele două acțiuni: selecție şi eliminare element arbitrar dintr-o listă.

```
sterge_elem([X|Z],X,Z).
sterge_elem([Y|Z],X,[Y|U]):-sterge_elem(Z,X,U).
```

Predicatul de generare a permutărilor se va scrie astfel:

```
permutari1(L,[H|T]):-sterge_elem(L,H,U), permutari1(U,T).
permutari1([],[]).
```

2.4. O generalizare a predicatului "append"

Se poate construi un predicat asemănător cu append care însă să efectueze concatenarea a trei liste. O soluție imediată ar fi:

```
append1(X,Y,Z,L):-append(Y,Z,U),append(X,U,L).
```

Daca X, Y şi Z sunt variabile legate la apel atunci comportamentul este cel dorit. Această soluție are însă dezavantajul că, atunci când se doreşte să se obțină toate descompunerile în trei subliste a unei liste date toți trei parametri din primul apel al lui 'append' sunt variabile nelegate.

Se poate renunța la apelul predicatului 'append' şi se poate generaliza definiția acestuia pentru concatenarea a trei liste. În acest fel se elimină anomalia amintită.

```
append2([],[],L,L).
append2([],[A|B],C,[A|D]):-append2([],B,C,D).
append2([H|T],U,V,[H|W]):-append2(T,U,V,W).
```

3. Desfăşurarea lucrării

Studenții vor executa trasarea predicatelor prezentate în sistemul Prolog standard. Se vor efectua trasări comparative pentru soluțiile alternative prezentate urmărindu-se eficiența algoritmilor.

4. Întrebări şi probleme

4.1. Construiți arborele de deducție ŞI-SAU pentru cele două variante de predicat pentru determinarea minimului unei liste.

4.2. Scrieți un predicat de eliminare a elementelor duplicate dintr-o listă, respectiv de determinare a numărului de elemente distincte dintr-o listă.

4.3. Explicați ce se întâmplă la apelul predicatului 'append1' (prima soluție pentru concatenarea a trei liste) în forma : ?-append1(X,Y,Z,[1,2,3]).

4.4. Se ştie că predicatul 'append' poate fi utilizat pentru selectarea unui element arbitrar dintr-o listă. Să se scrie un predicat cu numai două argumente (lista, element) care să efectueze această sarcină. Se cer două variante:

- elementele să fie regăsite în ordinea din listă;
- elementele să fie fi regăsite în ordinea inversă;

4.5. Să se scrie un program care determină lista "sumă" a două liste de numere.

4.6. Să se determine sublistele unei liste care au aceeaşi monotonie.

4.7. Să se determine monotonia (sublista) cea mai lungă a unei liste date.

LUCRAREA DE LABORATOR NR. 5

SORTARE (METODE DIRECTE)

1. Scopul lucrării

Sortarea este în general înțeleasă drept procesul de rearanjare a unui set dat de obiecte într-o ordine specificată. Scopul sortării este de a facilita ulterior căutarea membrilor setului sortat. Sortarea este o activitate des întrebuințată în procesarea datelor .

Date fiind articolele :

$S=a_1,...,a_n$

sortarea constă într-o permutare

$S'=a_{k1},...,a_{kn}$

astfel încât ,dată fiind o funcție de ordonare f, să avem :

$f(a_{k1}) <= f(a_{k2}) <= ... <= f(a_{kn})$

De obicei funcția f nu este folosită ca o procedură de calcul ci este memorată ca un câmp explicit al fiecărui articol. În rezumat post condiția sortării (vezi curs) este :

post_sort(S,S')=e_permutare(S,S'),e_ordonata(S')

Observație: Aceste cerințe sunt general valabile pentru toate programele din această lucrare.

2. Considerații teoretice

Utilizarea Prologului ca mecanism de specificare prezintă avantajul apropierii de descrierea naturală a problemelor .Deoarece algoritmii clasici de sortare au fost dezvoltați pentru structuri de gen tablou și secvență am ales folosirea listelor (din Prolog) pentru reprezentarea datelor. Limitarea introdusă de accesul articolelor prin capul listei nu afectează algoritmii descriși în acest mod.

2.1. Sortare prin generarea permutărilor

O soluție rezultată imediat din precizarea cerințelor constă în generarea permutărilor, urmată de verificarea ordonării fiecărei permutări generate în parte (până la obținerea permutării care satisface relația de ordine). Soluția este imediată dar în același timp este ineficientă. Pe lângă determinarea unui număr mare de permutări se verifică starea de ordonare a fiecăreia.

```
domains
      list=integer*
predicates
      sort(list,list)
      permutation(list,list)
      is_ordered(list)
      sorted(integer,list)
      append(list,list,list)
      order(integer,integer)
clauses
      append([],L,L).
      append([H|T],L,[H|R]):-
            append(T,L,R).

      sort(L1,L2):-
            permutation(L1,L2),
            is_sorted(L2),!.

      permutation(L,[H|T]):-
            append(V,[H|U],L),append(V,U,W),
```

```
        permutation(W,T).
    permutation([],[]).

    is_ordered([H|L]):-
        sorted(H,L).

    sorted(_,[]).
    sorted(N,[H|T]):-order(N,H),sorted(H,T).

    order(N,H):-N<H.
```

Predicatul sort (având ca parametri lista de sortat şi lista sortată) are o singură clauză cu semnificaţia: L2 este rezultatul sortării lui L1 dacă L2 este o permutare a listei L1 şi dacă L2 este ordonată. Predicatul permutation: prima clauză foloseşte descompunerea cu ajutorul append-ului a listei L în toate cele n moduri posibile date de şablonul [H|U], (n fiind cardinalul lui L) urmată de recompunerea constituenţilor (generând lista W) cu excepţia elementului selectat H, pe care îl adaugă în capul listei obţinute prin permutarea lui W (permutare ce se obţine prin apelul recursiv al aceluiaşi predicat). Clauza a doua specifică faptul că lista vidă este o permutare a listei vide.

Predicatul is_ordered specifică prin clauza lui că o listă este ordonată dacă în raport cu capul listei restul listei este sortată. Predicatul sorted specifică în clauză că o listă este sortată în raport cu un element dacă între element şi capul listei este definită relaţia de ordine cerută şi restul listei este sortat în raport cu capul listei. Faptul aceluiaşi predicat are semnificaţia că lista vidă este sortată în raport cu orice element.

Principiul urmărit în următorii doi algoritmi are la bază împărţirea unui set T în două zone logice: sursă (S) şi destinaţie (D). În permanentă S+D = T ocupă zona de memorie iniţial alocată setului. D este zona deja sortată (iniţial vidă, în final T) iar S este zona încă nesortată (iniţial T, în final vidă). Cele două metode se deosebesc prin modul de alegere al unui element din S şi de adăugare a lui la D. La fiecare pas S scade cu un element care se adaugă la D.

2.2. Sortare prin inserţie

Sortarea prin inserţie se caracterizează printr-o alegere aleatoare a elementului din sursă (deci cel mai simplu este să preluăm capul listei) şi inserarea lui pe poziţia stabilită de relaţia de ordine în D .

```
predicates
    sortins(list,list)
    ins(integer,list,list)
clauses
    sortins([],[]).
    sortins([X|L],M):-
        sortins(L,N),ins(X,N,M).

    ins(X,[A|L],[A|M]):-
        order(A,X),!,ins(X,L,M).
    ins(X,L,[X|L]).
```

Clauza predicatului sortins precizează că M este rezultatul sortării prin inserare a listei [X|L] dacă X se inserează pe poziţia dată de relaţia de ordine în lista obţinută prin sortarea prin inserţie a lui L (sortare obţinută prin apelul recursiv al aceluiaşi predicat). Faptul predicatului precizează că lista vidă este sortată.

O altă variantă a sortării prin inserţie (mai aproape de specificaţia sursa+destinatie=tablou) este:

```
    sortins2(A,B):-insort(A,[],B).
```

(semnificația parametrilor fiind: lista de sortat, parametru de acumulare reprezentând rezultatul parțial deja sortat şi respectiv lista sortată)

```
insort[H|T],P,R):-ins(H,P,Q),insort(T,Q,R).
insort([],P,R):-R=P.
```

Observație: Clauza doi s-ar mai fi putut scrie insort([],R,R)., dar s-a preferat varianta anterioară pentru o mai uşoară interpretare a clauzei.

2.3. Sortarea prin selecție

Sortarea prin selecție presupune selecția elementului minim din S şi adăugarea sa la capătul lui D .

```
predicates
      min(list,integer,integer)
      minl(list,integer)
      sortsel(list,list)
      err_el(integer,list,list)
clauses
      /* sterge prima aparitie a unui */
      /* element dintr-o lista        */
      err_el(_,[],[]):-!.
      err_el(A,[A|B],B):-!.
      err_el(A,[B|C],[B|D]):-
            err_el(A,C,D).

      min([],M,M).
      min([X|Y],M,Min):-
            X<M,min(Y,X,Min).
      min([X|Y],M,Min):-
            X>=M,min(Y,M,Min).

      minl([X|Y],Z):-
            min(Y,X,Z).

      sortsel([],[]).
      sortsel(L,[X|Y]):-
            minl(L,X),err_el(X,L,L2),
            sortsel(L2,Y).
```

Pentru sortarea prin selecție o altă variantă porneşte de la rescrierea minimului (noul predicat generând totodată şi lista din care minimul lipseşte).

```
minerr([H|T],X,[H|R]):-minerr(T,X,R),X<H,!.
minerr([H|T],H,T).

selsort(A,[H|R]):-minerr(A,H,T),selsort(T,R).
```

Sortarea prin selecție a unei liste A este o listă având ca prim element minimul listei A (determinat cu predicatul minerr ca fiind H) şi ca rest al listei rezultatul sortării prin selecție (deci apel recursiv al aceluiaşi predicat) a listei inițiale din care s-a eliminat minimul (T = A - [H]).

2.4. Sortarea prin interschimbare

Sortarea prin interschimbare presupune test de vecinătate. În exemplul următor predicatul append permite selectarea tuturor perechilor de elemente alăturate. Elementele sunt

interschimbate dacă este cazul, altfel append furnizează următoarea pereche. Când întreaga listă este parcursă ea este gata sortată şi cea de a doua clauză busort furnizează rezultatul.

```
predicates
	busort(list,list)
clauses
	busort(L,S):-
		append(X,[A,B|Y],L),order(B,A),
		append(X,[B,A|Y],M),!,busort(M,S).
	busort(L,L).
```

Pentru sortarea prin interschimbare o altă variantă ar fi:

```
	bubble2(X,Y,F):-sort(X,Z,F),bound(F),!,bubble2(Z,Y,NewF).
	bubble2(X,X,_).

	sort([H,I|T],[H|R],F):-order(H,I),!,sort([I|T],R,F).
	sort([H,I|T],[I|R],F):-F=0,sort([H|T],R,F).
```

Predicatul sort realizează o singură parcurgere a listei interschimbând elementele adiacente a căror relaţie de ordine nu este respectată. Dacă cel puţin o interschimbare a fost necesară predicatul setează şi variabila liberă F folosită pe post de semafor (clauza a doua). Predicatul bubble2 face o parcurgere a listei după care testează semaforul şi în caz că acesta a fost setat se apelează recursiv.

3. Desfăşurarea lucrării

Se va executa trasarea predicatelor de mai sus cu urmărirea instanţierii variabilelor. Se va compara eficienţa predicatelor anterioare cu alte alternative.

4. Întrebări şi probleme

4.1. Să se studieze comparativ predicatele sortins cu insort, sortsel cu selsort şi busort cu bubble (identificând deosebirile de nuanţă ale aceloraşi algoritmi). Să se traseze urmărindu-se comparativ rezultatele intermediare.

4.2. Să se aprecieze calitativ predicatele analizate. Să se aprecieze oportunitatea alegerii predicatului potrivit unei situaţii particulare.

4.3. Să se scrie alte predicate care să funcţioneze pe baza aceloraşi algoritmi.

LUCRAREA DE LABORATOR NR. 6

METODE AVANSATE DE SORTARE

1. Scopul lucrării

Prezentarea sortării prin partiționare (quicksort) în două variante Prolog. Este deasemenea discutată utilizarea listelor diferență în conceperea algoritmilor.

2. Considerații teoretice

Specificarea Prolog sporeşte lizibilitatea algoritmilor prin absența din limbaj a unei instrucțiuni de atribuire. Practic în Prolog fiecare stare are un nume. Dacă dorim să utilizăm valori noi trebuie să creem nume noi. Obținerea soluțiilor cât şi testarea condițiilor logice în Prolog se face prin mecanismul de unificare. Algoritmii sunt exprimați sub formă de raționamente. În spatele specificărilor predicative se regăsesc tehnici matematice fundamentale în teoria calculabilității cum ar fi inducția matematică, rescrierea, etc.

În cele ce urmează vom aborda problema sortării prin partiționare. Metodele directe de sortare prezentate în lucrarea nr. 5 au în cel mai bun caz o eficiență de ordinul N^2, unde N reprezintă numărul de elemente din secvență asupra căreia se aplică sortarea. Datorită eficienței sale medii de ordinul $N*\log_2 N$, eficiența teoretică limită pentru problema sortării, algoritmul de sortare prin partiționare este cunoscut şi sub numele de 'Quicksort'.

2.1. Specificarea Prolog a sortării prin partiționare

Sortarea prin partiționare este o metodă recursivă care constă în împărțirea unei secvențe în două subsecvențe asupra cărora se aplică aceeaşi metodă. Această strategie urmează bine-cunoscutul dicton "Divide et impera" (dezbină şi condu). O problemă mare este rezolvată prin descompunerea sa şi rezolvarea unor probleme mai mici.

Se efectuează partiționarea unei secvențe L (reprezentată ca listă în Prolog) după un element arbitrar H din L în două subsecvențe (de elemente din L) L_{st} şi L_{dr} astfel încât: $\forall X \in L_{st}, \forall Y \in L_{dr}: X \leq H, H \leq Y$. Prin aplicarea recursivă a aceleiaşi metode pe subsecvențele L_{st} şi L_{dr} se obține în final secvența sortată.

Alegerea elementului H se poate face arbitrar. O soluție naturală în Prolog constă în alegerea primului element din lista L de sortat. Fie deci L=[H|T]. Programul Prolog care implementează ideea prezentată este:

```
quicksort([H|T],S):-
            partitionare(H,T,LS,LD),
            quicksort(LS,L1S),
            quicksort(LD,L1D),
            append(L1S,[H|L1D],S).
quicksort([],[]).

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

2.2. Îmbunătățirea specificării prin utilizarea listelor diferență

Dezavantajul specificării predicative de mai sus este că după obținerea partițiilor ordonate mai este necesară încă o parcugere a uneia dintre ele de către predicatul 'append' (vezi lucrarea 2). Acest dezavantaj poate fi eliminat prin utilizarea 'listelor diferență'. Ideea de listă diferență poate fi înțeleasă din desenul de mai jos în care R este sublistă a lui L.

```
 L         R
 ↓         ↓
[E1,...,En|R]
```

Lista diferență D, notată L\R, este:

$D=L\backslash R=[E_1,...,E_n]$

Dacă se dă o listă L ('pointer' la începutul listei L) şi o sublistă R ('pointer' la un element din lista L), atunci lista diferență D=L\R, constă din elementele din L care nu sunt în R (elementele dintre cei doi 'pointeri' L şi R).

Putem folosi această tehnică pentru a elimina operația de concatenare (cu predicatul 'append') din specificarea predicativă din secțiunea 2.1. Dealtfel, în algoritmul 'quicksort', aşa cum a apărut el în literatură (într-o prezentare procedurală) nu este necesară o asemenea operație.

Dorim să construim un predicat 'q' cu următoarea semnificație:

```
q(L,S,R)
```

unde S\R este permutarea sortată a lui L. Atunci definiția 'quicksort' va fi:

```
quicksort(L,S):- q(L,S,[]).
```

Definitia lui 'q' va fi:

```
q([H|T],S,R):-
     partitionare(H,T,A,B),
     q(A,S,[H|Y]),
     q(B,Y,R).
q([],S,S).
```

Demonstrăm corectitudinea acestei specificări printr-un raționament inductiv după lungimea k a listei de sortat.

1. Pentru k=0 (listă vidă), avem într-adevăr q([],S,S), adică S\S este chiar lista vidă ([]) sortată.
2. Ipoteza inductivă. Pentru lung(L)≤k are loc q(L,S,R), unde S\R este permutarea sortată a lui L.
3. Fie lung(L)=k+1, şi L=[H|T] (folosim notațiile din definiția lui q prezentată mai sus). Deoarece atât A cât şi B au cel mult k elemente urmează că Y\R este permutarea sortată a lui B şi S\[H|Y] este permutarea sortată a lui A. Urmează că structura lui S este:

S=(permutarea sortată a lui A) urmată de
(elementul H) urmat de
(permutarea sortată a lui B) urmată de
(lista R)

Deoarece T este alcătuit din toate elementele $A_i \in A, B_i \in B$ şi din elementul H cu $A_i \leq H \leq B_i$ (conform definiției predicatului 'partitionare'), urmează că într-adevăr S\R este permutarea sortată a lui [H|T].

3. Desfăşurarea lucrării

Studenții vor executa trasarea predicatelor de sortare rapidă prezentate. Se va analiza comparativ eficiența celor doi algoritmi.

4. Întrebări şi probleme

4.1. Să se efectueze demonstrarea matematică printr-un raționament inductiv a corectitudinii primei versiuni prezentate pentru predicatul 'quicksort'.

4.2. Să se construiască arborii de execuție (comparativ - eventual să se efectueze trasarea în sistemul Prolog standard) pentru cele două versiuni de 'quicksort' prezentate.

LUCRAREA DE LABORATOR NR. 7

OBIECTE RECURSIVE. OBIECTE INCOMPLETE

1. Scopul lucrării

Lucrarea îşi propune familiarizarea studenţilor cu noţiunile de obiect recursiv şi obiect incomplet.

2. Consideraţii teoretice

Există cazuri practice care impun existenţa unor tipuri de date compuse. Aceste tipuri de date se introduc prin intermediul unui functor:

```
tip=t(tip1, tip2, ..,tipn);
```

unde tipi, i=1..n, reprezintă tipurile de date componente tipului compus.

2.1. Obiecte recursive

Recursivitatea poate fi folosită pentru descrierea obiectelor cărora nu li se cunoaşte în avans numărul de elemente.

Ex: Studenţii unui an:
an=vid(fără studenti)
an=a(nume,an).

Astfel a(albu,X) semnifică faptul că "albu" e primul student din an iar X reprezintă restul anului.
Un an avand trei studenţi se reprezintă:
a(albu, a(pop, a(rus,vid))).

Declaraţia tipului de dată constă în declaraţia celor două alternative, care mai poate fi făcută
an=a(nume, an); vid

cu semnificaţia că a(nume, an) e un obiect compus având functorul a, nume reprezintă tipul obiectelor aparţinând tipului compus recursiv şi an reprezintă restul obiectului.

2.2. Arbori

Pe baza consideraţiilor anterioare, putem defini arborii ca obiecte compuse şi recursive.

```
tree=t(id, tree, tree); nil
```

unde id reprezintă obiectul conţinut în nodul rădăcină, şi cei doi arbori recursivi reprezintă subarborii drept respectiv stâng.

2.2.1. Inserarea

Putem "creşte" un arbore, pornind de la arborele vid(nil), inserând noduri.

Generarea unui arbore de căutare (arbore cu proprietatea că pentru orice nod cheile din subarborele stâng sunt mai mici decât cheia nodului, iar cheile din subarborele drept sunt mai mari decât cheia nodului) se face cu ajutorul predicatului

```
ins(cheie_de_inserat, arbore_sursa, arbore_rezultat).

ins(I,nil,t(I,nil,nil)):-!.
ins(I,t(I,L,R),t(I,L,R)):-!,write("exista in arbore").
ins(I,t(I1,L,R),t(I1,NL,R)):-I<I1,!,ins(I,L,NL).
ins(I,t(I1,L,R),t(I1,L,NR)):-ins(I,R,NR).
```

Semnificaţia clauzelor fiind:

1. Inserarea unui nod într-un arbore vid reprezintă frunza având cheia dată.
2. Inserarea unui nod într-un arbore ce conține în nodul rădăcină cheia de inserat este arborele inițial (cu transmiterea mesajului că nu s-a făcut inserare propriu-zisă).
3. Inserarea unui nod într-un arbore ce are în rădăcină o cheie mai mare decât cheia de inserat este un arbore rezultat în urma inserării nodului în subarborele stâng al arborelui sursă.
4. Dacă nici una din alternativele anterioare nu este îndeplinită (existența tăierii în corpul clauzelor 1, 2, 3 ne scutește de a mai verifica I>I1) arborele rezultat se obține prin inserarea în subarborele drept și copierea rădăcinii și a subarborelui stâng.

Observație: Inserarea se face **totdeauna** ca frunză.

2.2.2. Ștergerea

Dacă inserarea s-a făcut ca frunză, în cazul ștergerii se disting mai multe variante:

1. ștergerea unei frunze - presupune eliminarea frunzei din arbore.
2. ștergerea unui nod cu un singur succesor - nodul eliminat și legătura de la tată redirectată către fiul nodului ce se elimina.
3. ștergerea unui nod având ambii succesori - se va face pe baza următoarei strategii: considerând acest nod ca rădăcină a arborelui ce începe în acest nod, se va elimina prin înlocuirea lui cu un alt nod al arborelui (iar acesta din urmă va fi eliminat efectiv) cu precizarea că arborele rezultat în urma înlocuirii să rămână în continuare arbore de căutare. Deci rădăcina va trebui înlocuită cu cea mai mare cheie mai mică decât rădăcina sau cea mai mică cheie mai mare decât rădăcina. Aceste noduri corespund nodului precedent și următor rădăcinii la parcurgerea în inordine. Deci eliminarea efectivă va fi făcută pentru nodul cel mai din dreapta al subarborelui stâng sau pentru nodul cel mai din stânga al subarborelui drept (unul din aceste noduri luând locul rădăcinii). Eliminarea acestui nod intră totdeauna în varianta 1 sau 2. În cazul nostru am ales înlocuirea rădăcinii cu cheia cea mai mare, mai mică decât rădăcina (micşorând subarborele stâng, dreptul nemodificat).

```
stergere(cheie,sursa,rezultat)

stergere(I,nil,nil):-!,write("nu exista in arbore").
stergere(I,t(I,nil,R),R):-!.
stergere(I,t(I,L,nil),L):-!.
stergere(I,t(I,L,R),t(NI,NL,R)):-!, stergnod(L,NI,NL).
stergere(I,t(I1,L,R),t(I1,NL,R)):-I<I1,!,stergere(I,L,NL).
stergere(I,t(I1,L,R),t(I1,L,NR)):-stergere(I,R,NR).
stergnod(t(S,LS,nil),S,LS):-!.
stergnod(t(IS,LS,RS),NI,t(IS,LS,R1)):-stergnod(RS,NI,R1).
```

Predicatul stergnod realizează înlocuirea efectivă prevăzută la cazul trei (parametri fiind: arborele în care se caută cheia cea mai din dreapta, cheia nodului cel mai din dreapta și arborele rezultat în urma transformării).

Demonstrăm corectitudinea predicatului stergnod printr-un raționament inductiv după lungimea k a ramurii dreapta a subarborelui în care se caută cheia maximă și se șterge nodul corespunzător.

1. Pentru k=0 (subarbore dreapta vid), cheia căutată este cea din nodul rădăcină și subarborele nou generat va fi subarborele stâng al arborelui dat (condiție satisfacută prin clauza 1 a predicatului).

2. Ipoteza inductivă: Presupunem că pentru arbori având lungimea ramurii dreapta <=k predicatul funcționează corect.

Fie acum un arbore având lungimea ramurii dreapta = k + 1. În acest caz cheia care se va şterge efectiv (NI) se caută în subarborele drept (corectitudinea căutării fiind asigurată de funcţionarea corectă a predicatului stergnod pe subarborele RS, a cărui ramură dreapta este <= k), iar subarborele (LS) stâng şi cheia nodului rădăcină rămân nemodificate.

2.3. Obiecte terminate în variabilă (incomplete)

2.3.1. Listă terminată în variabilă

Dorim să scriem un predicat care să verifice dacă un element este membru într-o listă: dacă da, să răspundă afirmativ, dacă nu, să-l insereze fără o nouă parcurgere şi fără a genera o nouă listă (inserarea în aceeaşi listă). Acest lucru ar fi posibil dacă lista în care se face căutarea nu ar fi o constantă. Putem reprezenta lista altfel decât ca pe o constantă, punând ca ultim element al listei o variabilă nelegată. Dacă la parcurgerea listei elementul căutat a fost găsit, parcurgerea se opreşte. Dacă nu a fost identificat până ajungem la variabila liberă de pe poziţia finală, înlocuim variabila cu elementul de inserat urmat de o variabilă liberă.

```
inserLTV(H,L):-var(L),!,L=[H|_].
inserLTV(H,[H|_]):-!.
inserLTV(H,[_|T]):-inserLTV(H,T).
```

Se observă că clauzele 1 şi 2 se pot unifica într-una singură (şi anume a 2-a): dacă a fost identificat elementul, ne oprim; dacă a fost identificată variabila liberă de la sfârşitul listei, o înlocuim cu elementul de inserat urmat de variabila liberă, ajungându-se la aceeaşi structură ([H|_]) pentru parametrul listă. Deci putem renunţa la clauza 1.

Observaţii:

1.Obiectele terminate în variabilă se numesc şi obiecte referenţiate (declaraţia tipului domeniului este precedat de cuvântul cheie reference).

2.Întrebarea pentru o astfel de problemă se pune:

?-inserLTV(5,[1,5|_]) cu răspunsul DA.

?-inserLTV(5,[1,3|_]) cu răspunsul DA şi lista sursă "crescută" la [1,3,5|_].

2.3.2. Arbore terminat în variabile

Dacă definim arborii având variabile libere pe poziţia frunzelor, atunci inserarea se poate face chiar în arborele sursă.

```
tree=reference t(id,tree,tree)

inser(I,t(I,_,_)):-!.
inser(I,t(I1,L,_)):-I<I1,!,inser(I,L).
inser(I,t(I1,_,R)):-inser(I,R).
```

Punând întrebarea:

```
?-inser(7,T),inser(3,T),inser(5,T).
```

arborele va avea formele intermediare:

```
1.t(7,_,_).
2.t(7,t(3,_,_),_).
3.t(7,t(3,_,t(5,_,_)),_).
```

Ultima formă este şi rezultatul returnat de scop.

OBSERVAȚII

1. Deoarece o variabilă nelegată se transmite de la un subscop la altul, domeniile list, tree se declară referințe ale unor obiecte (simple respectiv compuse). Intern, aceasta se materializează prin transmiterea de adrese şi nu valori).
2. Dacă nu se declară referințele şi totuşi întrebările se pun pentru nişte obiecte terminate cu variabile, la execuție se afişează un avertisment de transmitere de referințe (oricum rezultatul se obține corect).

3. Desfăşurarea lucrării

Se vor executa toate programele pentru diferite date de intrare.

4. Întrebări şi probleme

4.1. Să se scrie un predicat care tipăreşte arbori binari.

4.2. Să se "crească" un arbore pornind de la arborele vid, după fiecare inserare tipărindu-se arborele intermediar obținut.

4.3. Plecând de la arborele generat la punctul 3 să se şteargă chei aleatoare, tipărind arborii obținuti.

4.4. Cum răspunde sistemul dacă la III.1. punem întrebarea
?-inserLTV(7,[1,5,-]). Explicați.

4.5. Să se "crească" un arbore terminat în variabile, pornind de la nodul rădăcină=variabilă liberă, tipărind toate formele intermediare.

4.6. Cum este mai eficient: să reprezentăm arborii terminați în variabile şi să inserăm chiar în ei sau terminați în nil şi să inserăm returnând un nou arbore?

4.7. Cum se şterge o cheie dintr-un arbore terminat în variabilă?

LUCRAREA DE LABORATOR NR. 8

ARBORI AVL

1. Scopul lucrării

Se prezintă o soluţie cunoscută pentru îmbunătăţirea performanţelor în operaţiile de bază asupra unui arbore binar. Aceasta se bazează pe menţinerea sa într-o anumită stare de "echilibru" (arbori AVL) în raport cu operaţiile de bază. Specificarea predicativă prezentată în lucrare se bazează pe efecte laterale şi tehnici de rescriere.

2. Consideraţii teoretice

2.1. Arbori AVL

În lucrarea precedentă am făcut cunoştinţă cu noţiunea de functor în Prolog. Functorii Prolog au fost folosiţi pentru reprezentarea obiectelor recursive de tip listă şi arbore binar. Au fost discutaţi arborii de căutare. Au fost definite predicate pentru inserarea şi ştergerea de noduri dintr-un arbore de căutare. Metodele prezentate însă, nu permit menţinerea unui control asupra structurii arborelui, care, se poate depărta oricât de starea de "echilibru". În starea de "echilibru" subarborii stânga/dreapta au aceeaşi înălţime şi găsirea unei chei într-un arbore binar cu N noduri cere în medie $\log_2 N$ comparaţii. În cazul cel mai defavorabil însă - cazul arborelui degenerat în listă - sunt necesare în medie N/2 comparaţii. Această din urmă peformanţă nu justifică menţinerea structurii de arbore.

O soluţie pentru îmbunătăţirea performanţelor operaţiilor de bază asupra arborelui de căutare constă în menţinerea sa în stare de "echilibru". Trebuie luat însă în considerare şi efortul de menţinere a "echilibrului". O soluţie eficientă constă în formularea unei definiţii mai puţin stricte a "echilibrului" în arbore. Un asemenea "echilibru imperfect" ar trebui să poată fi păstrat cu uşurinţă, dar în acelaşi timp să aducă o îmbunătăţire reală a operaţiilor de bază asupra structurii de arbore. O asemenea definiţie a echilibrului a fost formulată de Adelson-Velski şi Landis. Criteriul de echilibru este următorul:

Un arbore de căutare este echilibrat dacă şi numai dacă pentru fiecare nod înălţimile celor doi subarbori ai săi diferă prin cel mult 1.

Arborii care satisfac condiţia de mai sus se numesc arbori AVL (după iniţialele celor care i-au propus). Definiţia este simplă, cere un efort mic de păstrare a echilibrului, şi adâncimea căii de căutare în arbore este practic identică cu cea dintr-un arbore perfect echilibrat.

Operaţiile de căutare, inserare şi ştergere asupra unui arbore AVL se pot efectua cu un efort de ordinul $\log_2 N$, chiar şi în situaţia cea mai defavorabilă. Această afirmaţie se bazează pe o teoremă demonstrată de Adelson-Velski şi Landis, teoremă care garantează că un arbore AVL nu va depăşi înălţimea arborelui corespunzător perfect echilibrat cu mai mult de 45%, indiferent de numărul de noduri.

Prezentăm mai jos structura arborilor AVL în cazul cel mai defavorabil (dezechilibru maxim cu păstrarea însă a proprietăţii din definiţie). Fie T_h arborele AVL cu număr minim de noduri, şi cu înălţime h. Evident T_0 este arborele vid şi T_1 este arborele cu un singur nod. Pentru a construi arborele T_h, cu $h>1$, vom adăuga la rădăcină doi subarbori care au la rândul lor un număr minim de noduri. Unul din arbori va trebui să aibă înălţimea h-1 iar celalalt h-2. Prezentăm mai jos arborii T_0,T_1,T_2,T_3 şi T_4 într-o notaţie functorială de tip Prolog. Arborii vor fi încărcaţi cu cheile 1,2,3,... Fie:

tree=t(tree,integer,tree);nil

definiția domeniului "tree" (arbore). Notăm cu $T_i\{k_1,k_2,...\}$ arborele AVL de înălțime i, cu număr minim de noduri şi cu cheile $k_1,k_2,...$ Atunci:

$T_0\{\} = nil$
$T_1\{1\} = t(nil , 1 , nil)$
$T_2\{1,2\} = t(T_1\{1\} , 2 , nil)$
$T_3\{1,2,3,4\} = t(T_2\{1,2\} , 3 , T_1\{4\})$
$T_4\{1,2,3,4,5,6,7\} = t(T_3\{1,2,3,4\} , 5 , T_2\{6,7\})$

Criteriul de construcție a sugerat denumirea acestei secvențe de arbori definită inductiv: arbori Fibonacci. Astfel, numărul de noduri N_k în arborele T_k este:

$N_0=0$, $N_1=1$, $N_k=N_{k-1}+1+N_{k-2}$

2.2. Predicatele *assert* şi *retract*

Programele Prolog sunt extensibile (la nivel de interpretor limbajul furnizează compilare incrementală). Reprezentarea predicatelor se face în mod unitar sub formă de colecții de clauze Horn. Limbajul furnizează predicate predefinite pentru modificarea dinamică a programelor. Predicatele **asserta(<clauza>)** /**assertz(<clauza>)** permit adăugarea unei clauze la începutul /sfârşitul setului de clauze care definesc un predicat. În acest mod baza de cunoştințe poate fi extinsă în general atât cu fapte (clauze fără corp) cât şi cu reguli (clauze cu corp). Există de asemenea posibilitatea eliminării clauzelor. Aceasta se poate face cu ajutorul predicatului **retract(<clauza>)**.

Modificarea dinamică a comportamentului programelor este necesară în special în aplicatii de Inteligență Artificială. Dacă însă se au în vedere numai fapte (nu şi reguli), predicatele assert şi retract permit conceperea de algoritmi cu "efecte laterale". Mecanismul de "negație ca eşec" din Prolog determină pierderea legărilor de variabile efectuate în procesul de rezoluție. Datele memorate cu assert nu sunt însă afectate de acest proces (sunt "persistente" la eşec). În plus ele sunt global accesibile la nivelul întregului program. Spre deosebire de parametri şi datele locale, ele pot fi create, modificate sau consultate de orice predicat.

2.3. Inserarea în arbori AVL

Vom prezenta în continuare specificarea predicativă a inserării unui nod într-un arbore AVL. Construirea unei definiții de predicat Prolog solicită efectuarea unei clasificări, sau a unei împărțiri pe cazuri a problemei. Fiecare caz admite apoi o definiție (predicativă) în general recursivă. În Prolog, reprezentarea datelor este simbolică. Reprezentarea functorială introdusă în lucrarea 7 permite raționarea pe structuri arborescente în general.

Să presupunem acum că inserarea unui nod într-un arbore AVL t(S,_,D) (reamintim aici că un arbore AVL este un arbore de căutare) se face în subarborele stânga S (cazul inserării în subarborele dreapta D este simetric) şi că înălțimea subarborelui creşte cu 1. Atunci avem trei cazuri:

1. $h_S=h_D$: subarborii S şi D devin de înălțimi inegale dar criteriul de echilibru nu este violat.
2. $h_S<h_D$: echilibru este îmbunătățit.
3. $h_S>h_D$: criteriul de echilibru este violat şi este necesară restructurarea arborelui.

O analiză atentă relevă existența a numai două situații care necesită restructurare în urma inserării în subarborele stânga, respectiv două situații simetrice pentru inserarea în subarborele dreapta. Pentru aceste situații sunt prezentate mai jos restructurările corespunzătoare, cu păstrarea condițiilor de definiție pentru un arbore de căutare şi pentru echilibru în arbori AVL. **Restructurările** sunt marcate cu semnul "➢", iar **inserările** cu ">>". Cu steluță (*) sunt marcați subarborii în care s-a făcut inserarea elementului şi a căror înălțime creşte (cu 1), alterând echilibrul. Dacă doi subarbori sunt marcați simultan cu steluță, atunci inserarea (cu creşterea înălțimii) s-a făcut asupra unuia (oarecare) dintre ei. Există în total patru tipuri posibile de restructurare, notate mai jos cu LL, LR, RR şi RL.

1. t(t(SS,NodA,DS),NodB,D) >>
   t(t(SS*,NodA,DS),NodB,D) (unde $h_{SS}=h_{DS}=h_D$)

LL: t(t(SS*,A,DS),B,D) ➢ t(SS*,A,t(DS,B,D))

2. t(t(SS,NodA,t(SDS,NodB,SDD)),NodC,D) >>
   t(t(SS,NodA,t(SDS*,NodB,SDD*)),NodC,D)
   (unde $h_{SS}=h_D=h_{SDS}+1=h_{DDS}+1$)

LR: t(t(SS,NodA,t(SDS*,NodB,SDD*)),NodC,D) ➢
   t(t(SS,NodA,SDS*),NodB,t(SDD*,NodC,D)

3. t(S,NodA,t(DS,NodB,DD)) >>
   t(S,NodA,t(DS,NodB,DD*)) (unde $h_S=h_{DS}=h_{DD}$)

RR: t(S,NodA,t(DS,NodB,DD*)) ➢ t(t(S,NodA,DS),NodB,DD*))

4. t(S,NodA,t(t(DSS,NodB,DSD),NodC,DD)) >>
   t(S,NodA,t(t(DSS*,NodB,DSD*),NodC,DD))
   (unde $h_S=h_{DD}=h_{DSS}+1=h_{DSD}+1$)

RL: t(S,NodA,t(t(DSS*,NodB,DSD*),NodC,DD)) ➢
   t(t(S,NodA,DSS*),NodB,t(DSD*,NodC,DD))

Pentru menținerea structurii de arbore AVL se foloseşte următoarea reprezentare functorială (definitia de domenii este prezentată în maniera Turbo Prolog):

```
nod,factor_echilibru_AVL=integer
tree=t(nod,factor_echilibru_AVL,tree,tree);nil
```

Factorul de echilibru poate avea una din următoarele valori:

-1 - dacă subarborele stânga este mai înalt (cu 1);
 0 - dacă cei doi subarbori au aceeaşi înălțime;
+1 - dacă subarborele dreapta este mai înalt;

Deasemenea, se utilizează un predicat în baza de date:

```
h(bool)
```

"h(true)" semnifică faptul că "arborele a crescut în înălțime", iar "h(false)" că "arborele şi-a păstrat înălțimea" în urma precedentei inserări. Inițial are loc "h(false)".

Prezentăm mai jos specificarea predicatului de inserare:

```
ins (nod, arbore_AVL, arbore_AVL_rezultat_in_urma_inserãrii)
```

Inserarea se poate face într-un arbore vid (clauza 1), caz în care înălțimea sa creşte deci se asertează h(true), sau ne-vid (clauzele 2-4). Dacă arborele este ne-vid nodul de inserat poate coincide cu rădăcina (deci poate fi deja în arbore - clauza 2), caz în care arborele nu se modifică deci se asertează h(false), sau poate să difere, caz în care se va insera într-unul din subarborii stânga/dreapta. Această inserare determină un proces de restructurare conform celor prezentate mai sus. Oricum, inserarea se face printr-un apel recursiv al predicatului "ins()" pe subarborele stânga/dreapta, deci subarborele respectiv va rămâne un arbore AVL.

```
ins(I,nil,t(I,0,nil,nil)):-!,
            retract(h(_)),asserta(h(true)).
ins(I,t(I,E,L,R),t(I,E,L,R)):-!,
            retract(h(_)),asserta(h(false)).
ins(I,t(I1,E,L,R),t(I2,NE,NL,NR)):-I<I1,!,
            ins(I,L,NL1),retract(h(H)),!,
            rs(H,t(I1,E,L,R),t(I2,NE,NL,NR),NL1).
ins(I,t(I1,E,L,R),t(I2,NE,NL,NR)):-
            ins(I,R,NR1),retract(h(H)),
            rd(H,t(I1,E,L,R),t(I2,NE,NL,NR),NR1).
```

Procesul de restructurare este prezentat pentru cazurile de inserare în subarborele stânga (procesul de inserarea în subarborele dreapta poate fi înțeles în acelaşi mod datorită simetriei problemei). Restructurarea este efectuată de către predicatul:

rs (daca_arborele_a_crescut_?,
arbore_AVL=A,
intreg_arborele_AVL_dupa_inserare_si_restructurare,
rezultatul_inserarii_in_subarborele_stânga_al_A)

in care există un singur parametru de ieşire (parametrul al treilea din listă).

Prezentăm acum clasificarea problemei pe cazuri specificate prin clauze Horn pe baza parametrilor de intrare. Dacă subarborele stânga nu a crescut (clauza 1) se asertează "h(false)". Altfel, dacă arborele a crescut fără însă să se altereze echilibrul cerut în definiția arborelui AVL (clauzele 2 şi 3) se marchează păstrarea ("h(false)") sau creşterea ("h(true)") înălțimii arborelui. Altfel, creşterea subarborelui stânga violează condiția din definiția arborelui AVL şi se va efectua restructurarea LL sau LR de către predicatele r1s, respectiv r2s.

```
rs(false,t(I1,E,L,R),t(I1,E,NL,R),NL):-!,asserta(h(false)).
rs(true,t(I1,1,L,R),t(I1,0,NL,R),NL):-!,asserta(h(false)).
rs(true,t(I1,0,L,R),t(I1,-1,NL,R),NL):-!,asserta(h(true)).
rs(true,t(I1,-1,L,R),t(I,NE,NL,NR),NL1):-NL1=t(_,-1,_,_),!,
                        r1s(t(I1,-1,L,R),t(I,NE,NL,NR),NL1),
                        asserta(h(false)).
rs(true,t(I1,-1,L,R),t(I,NE,NL,NR),NL1):-
                        r2s(t(I1,-1,L,R),t(I,NE,NL,NR),NL1),
                        asserta(h(false)).
```

Predicatele r1s şi r2s stabilesc următoarele relații între parametri (parametrul al doilea este de ieşire în fiecare caz):

r1s(arbore_AVL=A,

```
        intreg_arborele_AVL_dupa_inserare_şi_restructurare_LL,
        rezultatul_inserarii_in_subarborele_stânga_al_A)

r2s( arbore_AVL=A,
        intreg_arborele_AVL_dupa_inserare_şi_restructurare_LR,
        rezultatul_inserarii_in_subarborele_stânga_al_A)
```

Predicatul r1s este definit printr-o singură clauză.

```
r1s(t(I1,-1,_,R),t(IS,0,LS,t(I1,0,RS,R)),t(IS,-1,LS,RS)).
```

Pentru predicatul r2s se disting trei cazuri. Două din ele se datorează inserării (cu creşterea înălţimii) pe stânga sau pe dreapta în ramura din dreapta a subarborelui stânga (fie el SD) Aceste cazuri sunt specificate clauzele 1 şi 2 (vezi şi definiţia regulii LR). Mai există însă cazul când arborele SD era vid. În acest caz prin inserare el va creşte în înălţime însă factorul său de echilibru va fi 0.

```
r2s(t(I1,-1,_,R),t(IR,0,t(IS,0,LS,LR),t(I1,1,RR,R)),
      t(IS,1,LS,t(IR,-1,LR,RR))).
r2s(t(I1,-1,_,R),t(IR,0,t(IS,-1,LS,LR),t(I1,0,RR,R)),
      t(IS,1,LS,t(IR,1,LR,RR))).
r2s(t(I1,-1,_,R),t(IR,0,t(IS,0,LS,LR),t(I1,0,RR,R)),
      t(IS,1,LS,t(IR,0,LR,RR))).
```

Clauzele predicatelor rd, r1d şi r2d exprimă acelaşi raţionament pentru restructurările RR şi RL.

```
rd(false,t(I1,E,L,R),t(I1,E,L,NR),NR):-!,asserta(h(false)).
rd(true,t(I1,-1,L,R),t(I1,0,L,NR),NR):-!,asserta(h(false)).
rd(true,t(I1,0,L,R),t(I1,1,L,NR),NR):-!,asserta(h(true)).
rd(true,t(I1,1,L,R),t(I,NE,NL,NR),NR1):-NR1=t(_,1,_,_),!,
                        r1d(t(I1,1,L,R),t(I,NE,NL,NR),NR1),
                        asserta(h(false)).
rd(true,t(I1,1,L,R),t(I,NE,NL,NR),NR1):-
                        r2d(t(I1,1,L,R),t(I,NE,NL,NR),NR1),
                        asserta(h(false)).

r1d(t(I1,1,L,_),t(IS,0,t(I1,0,L,LS),RS),t(IS,1,LS,RS)).

r2d(t(I1,1,L,_),t(IR,0,t(I1,-1,L,LR),t(IS,0,RR,RS)),
      t(IS,-1,t(IR,1,LR,RR),RS)).
r2d(t(I1,1,L,_),t(IR,0,t(I1,0,L,LR),t(IS,1,RR,RS)),
      t(IS,-1,t(IR,-1,LR,RR),RS)).
r2d(t(I1,1,L,_),t(IR,0,t(I1,0,L,LR),t(IS,0,RR,RS)),
      t(IS,-1,t(IR,0,LR,RR),RS)).
```

Prezentăm acum o trasare de probă pentru o mai bună înţelegere a mecanismului. În acest scop construim un predicat de tipărire (prin parcurgere în inordine) a arborelui AVL.

```
tip(L,nil):-!.
tip(L,t(I,E,S,R)):-NL=L+1,tip(NL,R),tipch(L,I,E),tip(NL,S).

tipch(0,I,E):-write('[',I,',',E,']'),!,nl.
tipch(L,I,E):-NL=L-1,write("   "),tipch(NL,I,E).

?-    asserta(h(false)),
      write("\n---> 4"),ins(4,nil,T1),nl,tip(0,T1),
      write("\n---> 5"),ins(5,T1,T2),nl,tip(0,T2),
      write("\n---> 7"),ins(7,T2,T3),nl,tip(0,T3),
      write("\n---> 2"),ins(2,T3,T4),nl,tip(0,T4),
```

```
      write("\n---> 1"),ins(1,T4,T5),nl,tip(0,T5),
      write("\n---> 3"),ins(3,T5,T6),nl,tip(0,T6),
      write("\n---> 6"),ins(6,T6,T7),nl,tip(0,T7).
```

```
---> 4
[4,0]

---> 5
    [5,0]
[4,1]

---> 7
    [7,0]
[5,0]
    [4,0]

---> 2
    [7,0]
[5,-1]
    [4,-1]
        [2,0]

---> 1
    [7,0]
[5,-1]
        [4,0]
    [2,0]
        [1,0]

---> 3
        [7,0]
    [5,1]
[4,0]
        [3,0]
    [2,0]
        [1,0]

---> 6
        [7,0]
    [6,0]
        [5,0]
[4,0]
        [3,0]
    [2,0]
        [1,0]
```

3. Desfăşurarea lucrării

Studenții vor testa predicatele prezentate pe diverse date de intrare. Se vor studia în special predicatele de restructurare efectuată în urma unei inserări. Se vor efectua trasări în mediul Turbo Prolog.

4. Întrebări şi probleme

4.1. Dintre operațiile de bază asupra arborilor AVL a fost studiată numai operația de inserare. Operația de căutare este de fapt cea prezentată pentru arbori de căutare în lucrarea 7. Să se scrie un predicat pentru ştergere element dintr-un arbore AVL.

4.2. Pe baza predicatului de inserare prezentat în lucrare, să se scrie un program care să construiască o secvență de arbori Fobonacci. Să se generalizeze problema prin construirea unui predicat care să permită construirea secvenței de arbori $T_1,T_2,T_3,...$ Fibonacci până la o înălțime primită ca parametru.

LUCRAREA DE LABORATOR NR. 9

GRAFURI. DRUMURI ÎN GRAF

1. Scopul lucrării

Se prezintă modalități de găsire a drumului în graf de la un nod inițial la un nod terminal.

2. Considerații teoretice

Găsirea drumului într-un graf se poate face pe baza unui algoritm cu revenire (backtracking). Principiul algoritmilor cu revenire este căutarea soluției, la fiecare pas impunînd o condiție suplimentară pe care să o îndeplinească soluția. În cazul nostru, condiția suplimentară constă în introducerea unui nou nod în drumul căutat. După fiecare condiție nou impusă se cercetează dacă sunt îndeplinite cerințele globale ale problemei (în cazul nostru dacă s-a ajuns la nodul terminal). Indiferent de rezultatul testului (am ajuns/n-am ajuns la nodul terminal), se reface starea anterioară impunerii ultimei condiții (se elimină ultimul nod introdus în drum) şi se încearcă satisfacerea condiției altfel (respectiv introducerea unui alt nod următor în drum). Algoritmul se termină cînd la nici unul din paşii la care s-au luat decizii impunându-se condiții nu se mai poate îndeplini condiția (respectiv la nici unul din paşi n-a mai rămas nici un nod neales).

Algoritmii scrişi în Prolog, dacă nu specificăm altfel, conțin un backtracking implicit (datorită implementării). Acesta realizează automat refacerea stării anterioare (prin ştergerea ultimului nod din arborele de execuție) şi încercarea celorlalte alternative (altă satisfacere a nodului anterior şters). Dacă vrem să inhibăm backtrackingul, trebuie să ne oprim la prima soluție validă. Aceasta se realizează cu o tăiere (!) imediat ce se obține soluția.

Cu ajutorul algoritmilor cu revenire se pot rezolva următoarele tipuri de probleme:

i) găsirea tuturor soluțiilor unei probleme (aşa funcționează TurboPrologul dacă nu se specifică altfel).
ii) găsirea unei soluții oarecare, de obicei prima soluție (! după ajungerea la soluție).
iii) găsirea soluției optime, ceea ce s-ar putea face:
   - comparând toate soluțiile de la i) şi alegând soluția optimă
   - după găsirea unei prime soluții, o considerăm un optim parțial, după care la fiecare pas vom căuta soluții mai bune decît optimul parțial (pe care de fiecare dată îl actualizăm).

3. Drumuri în graf

Vom reprezenta graful neorientat prin arcele existente între noduri: e_arc(nod,nod) sau e_usa(nod,nod).

3.1. Drum între noduri date

Este echivalentul găsirii drumului prin labirint.

Graful fiind neorientat înseamnă că existența unui arc între nodurile A şi B ar putea fi materializată prin e_usa(A,B) sau prin e_usa(B,A) (în funcție de modul cum a fost reprezentat arcul în graf). De aceea ne trebuie un predicat care testează existența uneia din cele două arce (trecerea între încăperile A şi B poate fi specificată fie prin uşă între A şi B fie prin uşă între B şi A).

```
e_trecere(X,Y):-e_usa(X,Y);e_usa(Y,X).
```

Căutăm drumul de la un nod de start transmis ca parametru la nodul obiectiv. O variabilă suplimentară va înmagazina rezultatul parţial ("firul" căii parcurse). După găsirea primei soluţii backtackingul este inhibat(!).

```
cauta(X,Y,Drum):-incearca(X,Y,[X],Drum).
incearca(X,X,F,F):-e_obiectiv(X),!.
incearca(X,Y,Fir,Drum):- e_trecere(X,Z),
                         not(member(Z,Fir)),
                         incearca(Z,Y,[Z/Fir],Drum).
```

Cu observaţia că trebuie să existe în program un predicat care ne specifică faptul că unul dintre nodurile din graf (o încăpere din labirint) e obiectiv (adică e încăperea în care trebuie să se ajungă).

3.2. Drum restricţionat

Acum căutăm un drum la obiectiv dar care să treacă prin anumite noduri intermediare.

```
arc(X,Y):- e_arc(X,Y); e_arc(Y,X).
e_drum(Nod,Nod,Drum,Drum).
e_drum(Nx,Ny,Drum_partial,Drum_final):-
      arc(Nx,Nz),
      not(member(Nz,Drum_partial)),
      e_drum(Nz,Ny,[Nz|Drum_partial],Drum_final).
e_drum_obiectiv(Nx,Ny,Drum):-
      bound(Nx),
      e_drum(Nx,Ny,[Nx],Drum),
      e_obiectiv(Ny).
e_drum_restrictionat(Nx,Ny,Lrestrictii,Drum):-
      e_drum_obiectiv(Nx,Ny,Drum),
      e_inclus_in_ordine(Lrestrictii,Drum).
```

După găsirea unui drum la obiectiv se testează dacă nodurile restricţie sunt incluse în ordine în soluţie. Testul se face cu predicatul

```
e_inclus_in_nod([],_):-!.
e_inclus_in_ordine([H|T],[H|T1]):-!, e_inclus_in_ordine(T,T1).
e_inclus_in_ordine(L,[_|T]):- e_inclus_in_ordine(L,T).
```

Observaţii:

1. Drumul de la start la obiectiv se află în lista Drum, dar este inversat (pe prima poziţie este obiectivul). Pentru testul de inclus în ordine ar trebui ca soluţiile să fie inversate.
2. Deoarece drumurile la obiectiv sunt multiple, o variantă mai eficientă ar fi inversarea listei restricţiilor şi testarea dacă inversa restricţiei se află în inversa soluţiei.
3. O variantă mai eficientă ar fi testarea existenţei restricţiilor în soluţie pe măsură ce aceasta se generează.

3.3. Drum optim

Presupunem că ne încadrăm în cerinţa iii) de la consideraţii teoretice. În faza actuală, prin optim vom înţelege drumul ce trece printr-un număr minim de noduri intermediare până la obiectiv. Vom utiliza un optim parţial (prima soluţie găsită), pe care îl vom actualiza la fiecare nouă soluţie care trece printr-un număr de noduri mai mic decât cel al optimului parţial (lungime fir se incrementează cu 1 la fiecare nou nod prin care se trece).

```
cauta(X,Drum):-asserta(sol_part([],20)),
      incearca(X,[X],Drum,1).
```

```
cauta(_,Drum):-sol_part(Drum,_).

incearca(X,Fir,Fir,Lfir):-e_obiectiv(X),
      retract(sol_part(_,_)),!,
      asserta(sol_part(Fir,Lfir)),
      fail.
incearca(X,Fir,Drum,Lfir):-e_trecere(X,Z),
      not(member(Z,Fir)),
      Lf1=Lfir+1,sol_part(_,Lopt),
      Lf1<Lopt,
      incearca(Z,[Z|Fir],Drum,Lf1).
```

Se utilizează o bază de date ce va conține predicatul sol_part(drum,lung_drum), reprezentând soluția parțială a problemei, conținând calea de la nodul inițial la obiectiv şi lungimea căii. La terminarea execuției, va conține rezultatul problemei. Se inițializează cu un drum vid şi o lungime inițială (peste a cărei valoare nu ne interesează nici o soluție). De câte ori se ajunge la obiectiv înseamnă că s-a identificat un drum ce trece prin mai puține noduri decât soluția parțială şi deci noul drum devine soluție parțială. Eşecul provocat la sfârşitul clauzei1 a predicatului incearca ne asigură că se încearcă toate celelalte alternative, iar când nu mai există, se execută clauza2 a predicatului cauta. Eşecul nu provoacă pierderea soluției deoarece s-a utilizat efect lateral.

3.4. Ciclu Hamiltonian

Prin ciclu Hamiltonian se înțelege un drum închis într-un graf (având nodul de pornire indentic cu nodul final) care trece o dată şi numai o dată prin toate nodurile grafului. Nu orice graf are ciclu Hamiltonian. Vom căuta ciclul într-un graf neorientat ale cărui arce au ataşată lungimea. Predicatele au următoarele argumente:

```
hamilton(noduri_in_ciclu,nod_start,drum,lungime_drum)
incearca(nr_noduri_ramase_de_parcurs,nod_initial,nod_curent,
      drum_partial,drum, cost_partial, cost)

hamilton(N,X,Drum,Cost):-N1=N-1,
      incearca(N1,X,X,[X],Drum,0,Cost).

inceaca(0,X,Y,Dp,[X|Dp],Cp,Cost):-arc(Y,X,D),!,Cost=Cp+D.
incearca(N,X,Y,Dp,Drum,Cp,Cost):-N>0,arc(Y,Z,D),
      not(member(Z,Dp)),N1=N-1,Cp1=Cp+D,
      incearca(N1,X,Z,[Z|Dp],Drum,Cp1,Cost).
```

4. Desfăşurarea lucrării

4.1. Să se execute toate programele.
4.2. Se modifică programele asfel încât să poată fi vizualizate rezultatele intermediare (în cazul problemei de optim).
4.3. Să se modifice programul de la 3.2. în spiritul obsevațiilor făcute la paragraful respectiv.
4.4. Se modifică 3.3. asfel încât costul să fie minim relativ la lungimea drumului nu la numărul de noduri.

LUCRAREA DE LABORATOR NR. 10

STRATEGII DE CĂUTARE ÎN GRAF

1. Scopul lucrării

Se prezintă specificări predicative pentru strategiile cunoscute de căutare în graf. Se utilizează algoritmi cu revenire (backtracking), liste terminate în variabilă şi efecte laterale. Datele se reprezintă sub formă de fapte (clauze) Prolog.

2. Consideraţii teoretice

Algoritmii prezentaţi în continuare se încadrează în clasa mai largă a problemelor de căutare (problematica aferentă "căutării" va fi discutată pe larg în cadrul disciplinei de Inteligenţă Artificială). În algoritmii prezentaţi în lucrare se caută o cale într-un graf între un nod de start şi un nod ţintă. În cele ce urmează vom specifica predicativ căutarea "în adâncime" şi "în lăţime". În final vom prezenta deasemenea algoritmul "best-first". Trebuie menţionat aici că algoritmii de căutare sunt dependenţi în general de reprezentarea utilizată pentru structura de graf. Pentru a nu simplifica din start problema dăm mai jos un set de reprezentări posibile pentru unul şi acelaşi graf (neorientat):

a) Ca listă de vecini în baza de date.

```
v(a,[b,c]). v(b,[a,c]). v(c,[a,b]).
```

b) Ca listă de vecini în reprezentare simbolică de tip listă.

```
G1=[[a,b,c],[b,a,c],[c,a,b]]
```

c) Ca listă de vecini în reprezentare functorială.

```
G2=l(v(a,[b,c]),l(v(b,[a,c]),l(v(c,[a,b]),nil)))
```

d) Ca listă de arce în baza de date.

```
arc(a,b). arc(a,c). arc(b,c).
```

În cele ce urmează vom considera graful reprezentat ca listă de arce în baza de date. Memorarea structurii de graf în baza de date permite exploatarea mecanismului implicit de căutare prin backtracking a limbajului Prolog. Specificarile predicative pentru traversările prezentate în lucrare asupra celorlalte reprezentări posibile pentru structura de graf rămân ca probleme propuse studenţilor spre rezolvare.

2.1. Căutare în adâncime (Depth-First)

Mecanismul de căutare Prolog se suprapune direct peste acest tip de căutare. Deoarece aceasta este strategia de lucru utilizată (şi dezvoltată pe larg) în lucrarea nr. 9, ne vom mărgini la încadrarea sa în problematica aferentă lucrării. Dacă vecinii unui nod pot fi inspectaţi într-o anumită ordine, ideea căutării în adâncime se bazează pe faptul că trecerea la următorul vecin se face numai după considerarea vecinului curent. Căutarea pe o anumită cale se abandonează fie la

detectarea nodului țintă, fie la detectarea unui ciclu. În aceste situații se revine în puncte de reluare, în care au rămas alternative (căi în graf) neinspectate.

```
cautare_in_adincime(Start,Tinta,Drum):-
      incearca(Start,Tinta,[Start],Drum).

incearca(Tinta,Tinta,Drum,Drum).
incearca(NodCurent,Tinta,DrumPartial,Drum):-
      e_arc(NodCurent,Vecin),not(member(Vecin,DrumPartial)),
      incearca(Vecin,Tinta,[Vecin|DrumPartial],Drum).

e_arc(Nod1,Nod2):-arc(Nod1,Nod2);arc(Nod2,Nod1).
```

2.2. Căutare în lățime (Breadth-First)

Algoritmul de căutare "în lățime" a unui drum între un nod de start şi un nod țintă preferă luarea în considerare a tuturor vecinilor (expandare) înaintea avansării pe una din căi spre țintă. Se poate considera că vecinilor (candidați ai) unui nod li se acordă aceeaşi şansă în a fi aleşi pentru găsirea unui drum spre țintă. Se operează cu doua liste: o listă Candidat (contine nodurile care "candidează" pentru "expandare"), şi o listă Expandat (conține nodurile care au fost deja inspectate şi expandate). Pentru uşurința manipulării lista Candidat o vom reprezenta ca o listă terminată în variabilă. Astfel, la fiecare pas, se preia un nod din capul listei Candidat, se găsesc toți vecinii săi (expandare) şi se introduc în coada listei Candidat (cu predicatul cunoscut de inserare într-o listă terminată în variabilă: inserLTV (lucrarea de laborator nr. 7)). Nodul astfel expandat este mutat în lista Expandat. În acest mod în lista Expandat se vor găsi nodurile inspectate în procesul de căutare a unui drum spre țintă. Când în lista Candidat este întâlnit nodul țintă se raportează lista Expandat. Lăsăm cititorul să înțeleagă ce se întâmplă dacă nu există nici un drum între nodurile start şi țintă considerate.

```
cautare_in_latime(Start,Tinta,Expandat):-
      trav(Tinta,[Start|_],[],Expandat).

trav(Tinta,[Tinta|_],Exp,Exp):-nonvar(Tinta).
trav(Tinta,[NodCurent|Cand],Exp1,Exp):-nonvar(NodCurent),!,
      expandare(NodCurent,Cand,Exp1),
      trav(Tinta,Cand,[NodCurent|Exp1],Exp).

expandare(Nod,_,Exp):-e_arc(Nod,Vecin),
      not(member(Vecin,Exp)),assertz(vecin(Vecin)),fail.
expandare(_,Cand,_):-assertz(vecin(eof)),colecteaza(Cand).

colecteaza(Cand):-retract(vecin(Vecin)),Vecin<>eof,!,
      inserLTV(Vecin,Cand),colecteaza(Cand).
colecteaza(_).
```

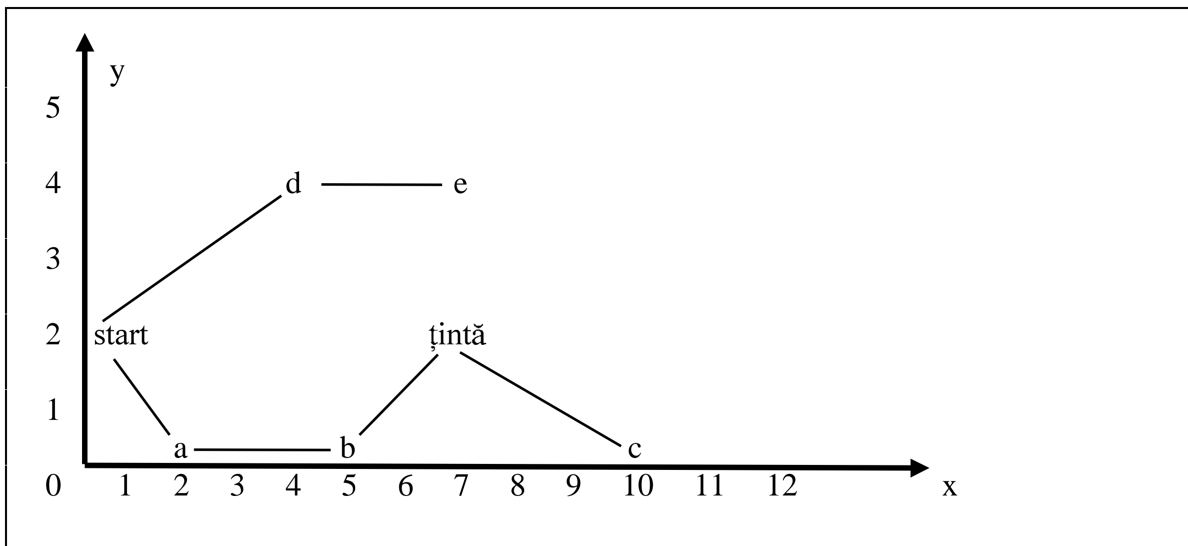
Pentru expandare se folosesc "efecte laterale". Vecinii unui nod se regăsesc prin mecanismul de backtracking al Prolog. De aceea ei sunt memorați în predicatul baza de date "vecin". Predicatul "colecteaza" preia toate nodurile memorate ca şi fapte ale predicatului "vecin" şi le inserează în coada listei Candidat.

Cititorul va remarca faptul că în final în lista Expandat nu se va găsi propriu-zis o cale între nodurile start şi țintă, ci o "istorie a căutării în lățime". Este posibil ca la expandare pentru fiecare vecin al nodului curent să se construiască o nouă listă conținând o cale candidată (parțială). În acest mod se va opera practic cu o listă de liste. La întâlnirea nodului țintă în capul unei asemenea căi parțiale se poate raporta soluția care de această dată va conține un drum între nodurile start şi țintă. Pentru a se permite o "trasare" mai lizibilă a programului se va folosi această din urmă strategie în prezentarea algoritmului Best-First.

2.3. Algoritmul Best-First

Algoritmul Best-First presupune că între oricare două noduri din graf există o "distanţă" calculabilă. Spre deosebire de agoritmul Breadth-First, se va alege spre expandare nodul aflat cel mai "aproape" de ţintă. În acest scop la fiecare pas lista căilor candidate pentru expandare se va sorta după distanţa rămasă de parcurs până la ţintă.

Considerăm graful din figura de mai jos unde poziţia nodurilor şi distanţa dintre ele sunt definite faţă de sistemul de coordonate xOy.



Acest graf îl reprezentăm ca o colecţie de fapte ale predicatului "pos_vec".

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

Distanţa între două noduri din graf poate fi calculată cu ajutorul predicatului "dist":

```
dist(Nod1,Nod2,Dist):-
      pos_vec(Nod1,X1,X2,_),pos_vec(Nod2,X2,Y2,_),
      Dist=(X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2).
```

Algoritmul Best-First expandează o cale parţială prin adăugarea vecinilor ce nu au fost încă inspectaţi. Apoi sortează lista căilor parţiale în ordinea apropierii lor de ţintă. În acest scop putem folosi algoritmul de sortare rapidă ("Quick-Sort" - vezi lucrarea de laborator nr. 6) în care predicatul care stabileşte relaţia de ordine între două căi parţiale (utilizat în predicatul "partitionare") va fi:

```
order([Nod1|_],[Nod2|_]):-e_obiectiv(Tinta),
      dist(Nod1,Tinta,Dist1),dist(Nod2,Tinta,Dist2),
      Dist1<Dist2.
```

Cu aceste precizări algoritmul Best-First este:

```
best([],[]):-!.
best([[Tinta|Rest]|_],[Tinta|Rest]):-e_obiectiv(Tinta),!.
best([[H|T]|Rest],Best):-
      pos_vec(H,_,_,Vec),expand(Vec,[H|T],Rest,Exp),
      q_sort(Exp,SortExp,[]),best(SortExp,Best).

expand([],_,Exp,Exp):-!.
expand([E|R],Cale,Rest,Exp):-not(member(E,Cale)),!,
      expand(R,Cale,[[E|Cale]|Rest],Exp).
expand([_|R],Cale,Rest,Exp):-expand(R,Cale,Rest,Exp).
```

Dacă se inserează un apel de predicat de tipărire a listei expandate şi sortate (SortExp) înainte de apelul recursiv al predicatului "best" (din clauza 3), se obţine următoarea trasare:

```
?-Exp=[[start]],nl,write(Exp),best(Exp,Best),nl,write("->",Best).

 [[start]]
 [[d,start],[a,start]]
 [[e,d,start],[a,start]]
 [[a,start]]
 [[b,a,start]]
 [[tinta,b,a,start],[c,b,a,start]]
 ->[tinta,b,a,start]
```

În urma primei expandări se adaugă vecinii nodului "start". Deoarece "d" este mai aproape de ţintă decât "a" (vezi figura) calea sa apare prima în lista expandată după sortare. În acest mod expandarea continuă întotdeauna de le calea parţială cea mai apropiată de ţintă.

3. Desfăşurarea lucrării

Studenţii vor testa predicatele prezentate în secţiunile 2.1., 2.2. şi 2.3. Se vor efectua trasări în mediu Turbo Prolog. Pentru algoritmul Best-First se va completa corespunzător codul pentru predicatul de sortare rapidă din lucrarea 6.

4. Întrebări şi probleme

4.1. Să se rescrie algoritmii de căutare în adâncime şi în lăţime pentru alte reprezentări posibile ale grafului.

4.2. Să se rescrie algoritmul de căutare în lăţime prin păstrarea integrală a căilor parţiale (asa cum s-a procedat la punctul 2.3. cu algoritmul Best-First).

4.3. Să se rescrie algoritmul Best-First fără păstrarea integrală a căilor parţiale, după modelul implementat pentru algoritmul de căutare în lăţime.

4.4. Deoarece în algoritmul "Best_First" în urma sortării interesează întotdeauna numai primul element (deci minimul listei) să se rescrie algoritmul utilizând predicatul "elimmin" introdus în lucrarea 5.

LUCRAREA DE LABORATOR NR. 11

DETERMINAREA COMPONENTELOR BICONEXE ÎNTR-UN GRAF

1. Scopul lucrării

Se prezintă specificarea predicativă pentru determinarea subgrafelor conexe dintr-un graf.

2. Considerații teoretice

Există în practică o serie de probleme a căror rezolvare conduce la reprezentarea datelor sub formă de graf. O problemă fundamentală în studiul grafelor o reprezintă determinarea proprietății de conexitate a grafului şi în particular, determinarea componentelor biconexe.

Un graf neorientat este **biconex** dacă între orice două noduri din graf există cel puțin un drum (nu are noduri **izolate**) şi nu are puncte de articulație. Un nod este **punct de articulație** dacă indexul nodului la traversarea în adâncime (specificând al câtelea nod vizitat este) este mai mic sau cel mult egal cu indexul strămoş al nodului fiu (într-un graf neorientat fiul unui nod este orice vecin al nodului dat care este parcurs ulterior nodului dat la traversare).

Pentru un nod într-un graf, indexul strămoş se defineşte după cum urmează:

```
stramos(Nod) = min{ df_num(Nod),
                    min( stramos(Fiu)),
                    min( df_num(Prim_Stramos))
               }                                        (1)
```

deci reprezintă minimul a trei componente:

- indexul nodului la traversarea în adâncime,
- minimul între indecşii strămoş ai tuturor fiilor nodului,
- minimul între indecşii la traversarea în adâncime a tuturor nodurilor prim_strămoş.

Prim strămoş într-un graf neorientat este un nod vecin cu nodul dat dar care a fost parcurs la traversarea în adâncime anterior nodului dat. În arborele de apel a traversării, legătura între nod şi prim strămoş apare sub forma unui **arc invers**.

Pentru a determina componentele biconexe ale unui graf se calculează:

- indexul traversării în adâncime al fiecărui nod,
- indexul strămoş al fiecărui nod,
- pentru fiecare nod se stabileste valoarea de adevăr a inegalității:

  index traversare în adâncime <= index strămoş fiu (2)

pentru fiecare fiu în parte, şi acele noduri pentru care inegalitatea este adevarată sunt declarate puncte de articulație ale grafului inițial,

-se extrag componentele biconexe. Punctele de articulație vor face parte din cel puțin două componente. În general un punct de articulatie va face parte din "n + 1" componente, unde "n" reprezintă numărul de fii pentru care inegalitatea (2) este îndeplinită.

3. Componente biconexe

În continuare este descris algoritmul de determinare a componentelor biconexe prin specificarea predicativă a sa. Graful este reprezentat prin perechi ordonate vecin(nod,lista_vecini).

Specificarea predicativă pentru determinarea indexului strămoş nu urmăreşte întocmai specificarea formală (1).

Dacă la (1) se determină minimul indecşilor strămoş ai tuturor fiilor, respectiv minimul indecşilor nodurilor prim strămoş la traversare, în final determinându-se minimul celor trei componente rezultate, la implementarea predicativă se inițializează indexul strămoş cu indexul traversării în adâncime şi de fiecare dată când în traversarea grafului se avansează la un nou vecin, se calculează noul minim între minimul parțial (inițializat sau rezultat din calcule anterioare) existent şi valoarea dată de vecin (indexul strămoş dacă vecinul este un fiu şi legătura tată-fiu un arc direct, respectiv indexul traversării în adâncime dacă fiul este un prim strămoş şi legătura arc invers).

- pentru vecinii fiu (arce directe):

```
pune_stramos_fiu(Nod,Fiu):-
                            retract(stramos(Nod,N)),!,
                            stramos(Fiu,Nf),min(N,Nf,Ns),
                            asserta(stramos(Nod,Ns)).
```

- pentru vecinii prim strămoş (arce inverse):

```
pune_prim_stramos(Nod,Prim_Stramos):-
                            retract(stramos(Nod,N)),!,
                            df_num(Prim_Stramos,Nps),min(N,Nps,Ns),
                            asserta(stramos(Nod,Ns)).
```

Determinarea indexului traversării este făcută în predicatul de traversare în adâncime:

```
trav(Nod,Tata):-
      assertz(df_num(Nod,N)),assertz(stramos(Nod,N)),
      arc_o(Nod,Vecin),Vecin<>Tata,
      pune_arc_inv(Nod,Vecin),continua(Nod,Tata,Vecin),fail.
trav(_,_).
```

unde predicatul continuă asigură avansarea în adâncime către vecinul specificat atât cât este posibil. Când nu se mai poate avansa în adâncime, predicatul continua se termină cu succes permițând reluarea traversării în adâncime pe o altă ramură (alt vecin al nodului curent; clauza 1 a predicatului trav).

Predicatul pune_arc_inv asigură realizarea arcelor inverse între nodul curent şi noduri prim_strămoş (în ipoteza că astfel de noduri există, Nv < N în predicat; dacă nu există, rezultatul evaluării inegalității anterioare provoacă eşec, deci arcul invers nu se mai adaugă).

```
pune_arc_inv(Nod,Vecin):-
                            df_num(Nod,N),df_num(Vecin,Nv),
                            Nv<N,!,asserta(stiva(Nod,Vecin)).
pune_arc_inv(_,_).
```

Arcele inverse se adaugă pe stivă şi reprezintă constituenți ai componentelor biconexe. Avansarea în adâncime pe o ramură dată este asigurată de predicatul continua.

```
continua(Nod,_,Vecin):-
      not(df_num(Vecin,_)),!,
      Fiu=Vecin,
      asserta(stiva(Nod,Fiu)),
      trav(Fiu,Nod),
      extrage_biconex(Nod,Fiu),
      pune_stramos_fiu(Nod,Fiu).
continua(Nod,_,Vecin):-
      Prim_Stramos=Vecin,
      pune_prim_stramos(Nod,Prim_Stramos).
```

Arcul direct se adaugă pe stivă (pentru a fi extras la componente biconexe), după care se continuă traversarea (nodul fiu devenind nod curent; clauza 1). Arcul invers se adaugă în cazul în care vecinul fusese vizitat la traversarea în adâncime (indexul traversării există deja în baza de date petru nodul vecin; clauza 2).

La revenirea din traversare (terminarea unei ramuri în adâncime) se extrage componenta biconexă dacă nodul curent este punct de articulație (N <= Nf)

```
extrage_biconex(Nod,Fiu):-
                    df_num(Nod,N),
                    stramos(Fiu,Nf),
                    N<=Nf,      /* pct. articulatie */
                    retract(stiva(X,Y)),
                    assertz(biconex(X,Y)),
                    eq(Nod,Fiu,X,Y),!,
                    assertz(biconex(-1,-1)),
                    fail.
extrage_biconex(_,_).
```

şi se actualizează indexul strămoş al nodului curent (pune_stramos_fiu) în conformitate cu arcul direct (cazul în care vecin = fiu; clauza 1). Dacă arcul este invers (cazul vecin = prim_stramoş; clauza 2) se actualizează indexul strămoş al nodului curent în conformitate cu acesta (pune_prim_stramos).

Extragerea componentei biconexe are loc când la revenirea de pe o ramură a traversării în adâncime nodul curent este depistat ca punct de articulație. În acest caz se extrag arcele de pe stivă până când se întâlneşte un arc în a cărui componentă intră punctul de articulație.

Actualizarea indexului strămoş a nodurilor se face în post-ordine (astfel încât valoarea finală a acestui index pentru un nod dat este determinată doar după ce au fost determinate valorile finale ale tuturor nodurilor fiu şi prim_stramoş).

Specificarea componentelor biconexe se face sub formă de arce neorientate. Dacă se reprezintă aceste componente se observă că fiecare punct de articulație apare în cel puțin două componente.

4. Desfăşurarea lucrării

Se va testa algoritmul urmărind modificarea indexului strămoş al fiecărui nod pe parcursul executiei, depistarea punctelor de articulație şi desprinderea componentelor biconexe.

Se va desena un arbore de apel, marcând toate arcele (directe şi inverse) precum şi indecşii df şi strămoş (cu valorile intermediare) pentru fiecare nod în parte. Se vor reprezenta componentele biconexe rezultate în ordinea apariției lor.

5. Întrebări şi probleme

5.1. Explicați comportarea în cazul în care graful conține puncte izolate.

5.2. Rescrieți algoritmul pentru reprezentarea grafului sub formă de arce.

5.3. Rescrieți algoritmul astfel încât componentele biconexe să aibă aceeaşi reprezentare cu a grafului inițial.

LUCRAREA DE LABORATOR NR. 12

METAPROGRAMARE

1. Scopul lucrării

Se va prezenta conceptul de metaprogramare şi principalele predicate predefinite care se folosesc în metaprogramare precum şi exemple semnificative. Se va folosi suportul pentru metaprogramare oferit de SICStus Prolog 3.5.

2. Consideraţii teoretice

În lucrările precedente au fost prezentate o serie de exemple, unele chiar cu un grad ridicat de complexitate, dar toate predicatele care au apărut au avut o caracteristică comună: s-au încadrat în logica predicatelor de ordinul întâi. Aceasta presupune limitarea ca un argument al unui predicat să nu poată fi un alt predicat.

În contextul teoretic mai larg al logicii predicatelor de ordin superior, acest lucru este însă posibil şi este numit generic metaprogramare. Prin metaprogramare vom avea posibilitatea ca un program să raţioneze despre el însuşi putându-şi testa anumite caracteristici, ba mai mult, un program se poate automodifica dinamic pe parcursul execuţiei.

Predicatele din tabelul de mai jos sunt metalogice, efectuând operaţii care necesită raţionarea despre instanţele unor termeni sau descompunerea unor termeni în constituenţii lor. Astfel de operaţii nu pot fi exprimate folosind predicate cu un număr finit de clauze.

Predicat	Acţiune
var(?X)	Verifică dacă variabila X este neinstanţiată(liberă). O variabilă neinstanţiată este o variabilă care nu a fost legată la nimic, decât eventual la o altă variabilă neinstanţiată (liberă).
nonvar(?X)	Verifică dacă variabila X este instanţiată. Este opusul lui var.
ground(?X)	Verifică dacă X este complet instanţiat, adică nu conţine nici o variabilă neinstanţiată (liberă)
atom(?X)	Verifică dacă variabila X este instanţiată cu un atom (adică un termen de aritate 0, care nu este număr)
float(?X)	Verifică dacă variabila X este instanţiată cu un număr real.
integer(?X)	Verifică dacă variabila X este instanţiată cu un număr întreg.
number(?X)	Verifică dacă variabila X este instanţiată cu un număr.
atomic(?X)	Verifică dacă variabila X este instanţiată cu un atom sau un număr.
functor(+Term, ?Name, ?Arity)	Furnizează numele principalului functor şi aritatea termenului.
functor(?Term, +Name, +Arity)	Furnizează cel mai general termen având functorul principal şi aritatea indicate
arg(+ArgNo, +Term, ?Arg)	Furnizează argumentul de pe poziţia ArgNo în termenul Term. Argumentele sunt numerotate începând cu 1.
+Term =.. ?List	Furnizează o listă al cărei cap este functorul principal al termenului Term şi a cărei coadă este alcătuită din argumentele termenului Term

?Term =.. +List	Furnizează un termen al cărui functor principal este capul listei, şi ale cărui argumente sunt coada listei List

Se poate observa că predicatul "=.." nu este neaparat necesar, deoarece funcționalitatea sa poate fi asigurată prin intermediul predicatelor "functor" şi "arg".

Predicatele din tabelul următor permit modificarea unui program chiar în timpul execuției sale, permițând adăugarea sau ştergerea unor clauze. De notat că predicatul care este modificat pe parcursul execuției trebuie declarat "dinamic". Deasemenea, trebuie ținut seama de faptul că ":-" este operator inifix şi deci clauzele de forma (Head :- Body) trebuie incluse între paranteze.

Predicat	Acțiune
assert(:Clause)	Adaugă clauza "Clause" la programul curent.
asserta(:Clause)	La fel ca assert dar clauza adăugată devine prima clauză a predicatului la care se referă
assertz(:Clause)	La fel ca assert dar clauza adăugată devine ultima clauză a predicatului la care se referă
clause(:Head, ?Body)	Verifică existența unei anumite clauze şi returnează corpul clauzei
retract(:Clause)	Prima clauză a programului care corespunde şablonului "Clause" este ștearsă
retractall(:Head)	Şterge toate clauzele care corespund şablonului "Head"
abolish(:Spec)	Şterge toate clauzele care corespund predicatului "Spec"
abolish(:Name, +Arity)	Şterge toate clauzele care corespund predicatului dat de "Name" şi "Arity"

2.1. Interpretor Prolog scris în Prolog

Folosind metaprogramarea, se poate scrie un interpretor Prolog în Prolog. Presupunem că clauzele programului au fost încărcate dintr-un fişier în memorie şi sunt de forma:

```
my_clause((Head:-Body)).   /* reguli */
```

sau

```
my_clause((Head:-true)).    /* axiome (fapte)*/
```

Interpretorul constă practic din doar două clauze:

```
execute((P,Q)) :- !, execute(P), execute(Q).
execute(P) :- my_clause((P:-Q)), execute(Q).
```

Deoarece predicatele predefinite (care au proprietatea "built_in" adevărată) trebuie tratate separat, mai trebuie adăugată o clauză şi interpretorul devine:

```
execute((P,Q)) :- !, execute(P), execute(Q).
execute(P) :- predicate_property(P, built_in), ! , P.
execute(P) :- my_clause((P:-Q)), execute(Q).
```

2.2. Determinarea variabilelor unui termen

Următorul exemplu ilustrează folosirea metaprogramării pentru determinarea listei tuturor variabilelor care apar în cadrul unui termen:

```
variabile(X, [X]) :- var(X), !.
variabile(T, L) :- functor(T, _, A), variabile(0, A, T, L).

variabile(A, A, _, []) :- !.
variabile(A0, A, T, L) :-
      A1 is A0 + 1,
      arg(A1, T, X),
      variabile(X, L0),
      variabile(A1, A, T, L1),
      append(L0, L1, L).
```

Se observă faptul că apar două predicate cu acelaşi nume dar care diferă ca număr de parametri (supraîncărcare), respectiv variabile/2 şi variabile/4 care se apelează reciproc.

Predicatul variabile/2 verifică dacă primul argument este variabilă şi dacă este întoarce o listă care îl conține (clauza 1) respectiv obține aritatea principalului functor al primului argument şi apelează predicatul variabile/4 care va returna lista variabilelor.

Predicatul variabile/4 parcurge toate argumentele termenului care apare ca al treilea argument şi apelează variabile/2 pentru fiecare dintre acestea. Rezultatul final este colectat printr-un append.

Append-ul din final se poate elimina folosind un mecanism bazat pe liste diferență şi predicatul devine:

```
variabile(X, [X|L], L) :- var(X), !.
variabile(T, L0, L) :- functor(T, _, A), variabile(0, A, T, L0, L).

variabile(A, A, _, L, L) :- !.
variabile(A0, A, T, L0, L) :-
      A1 is A0 + 1,
      arg(A1, T, X),
      variabile(X, L0, L1),
      variabile(A1, A, T, L1, L).
```

2.3. Produs cartezian generalizat

Prezentăm în continuare o altă aplicație a metaprogramării, în calculul produsului cartezian. Pentru calculul produsului cartezian a unei mulțimi prin ea însăşi, avem următoarea variantă elegantă de rezolvare:

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

pc2(L, _) :- member(X1,L), member(X2,L), assert(p2([X1, X2])), fail.
pc2(_, R) :- findall(X, p2(X), R), retractall(p2(_)).
```

Am obținut deci produsul cartezian $L^2 = L \times L$ prin folosirea predicatului member şi a mecanismului de backtracking din Prolog. Se remarcă folosirea metapredicatului "findall" pentru colectarea tuturor elementelor produsului cartezian asertate anterior în baza de date.

Pentru a calcula $L^N = L \times L ... \times L$ (de N ori) într-o manieră similară observăm că am avea nevoie de un predicat similar "pcn" care însă ar trebui să apeleze de N ori în prima clauză predicatul "member". Vom urmări deci să construim un predicat de forma:

```
pcn(N,L,R):-member(L1,L),...,member(LN,L),assert(t([L1,...,LN])),fail.
pcn(N,L,R):-findall((X),t(X),R),retractall(t(_)),retractall(pcn(_,_,_)))).
```

Pentru a obține un asemenea predicat îl vom construi dinamic folosind metaprogramarea, în modul următor:

```
:-dynamic pn/3.

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

append([], X, X).
append([H|T], B, [H|Y]) :- append(T, B, Y).

genlist(0, []).
genlist(N, [H|T]) :- N1 is N-1, genlist(N1, T).

genmember([], [], L).
genmember([H|T], [HH|TT], L) :- HH =.. [member, H, L],
      genmember(T, TT, L).

tr([H, HH], (H, HH)).
tr([H|T], (H, TT)) :- tr(T, TT).

creare(N, L, R) :- genlist(N, A),
     genmember(A, B, L),
     append(B, [assert(t(A)), fail], C),
     tr(C, CC),
     DD =.. [:-, pcn(Nr, L, R), CC],
     assert(DD),
     assertz((pcn(Nr,L,R):-
findall(X,t(X),R),retractall(t(_)),retractall(pcn(_,_,_)))).

pc(N,Lista,R):-creare(N,L,R),pcn(N,Lista,R).
```

Construirea clauzei a doua nu prezintă probleme, ea poate fi asertată direct. Construirea primei clauze însă ridică două probleme principale:

- construirea unei liste de N variabile libere [L1, ... , LN]
- construirea unei liste de N apeluri ale predicatului member

Aceste probleme se rezolvă respectiv de către predicatele “genlist” şi “genmember”. Predicatul “creare” este cel care crează predicatul “pcn” şi se observă cum după apelul lui “genlist” , “genmember” şi “append”, se aplică transformarea “tr” şi se obţine în final predicatul dorit. Rolul transformării “tr” este de a transforma o listă de forma [E_1,..., E_k] într-o listă echivalentă de forma (E_1,...,E_k) acest lucru fiind necesar pentru compunerea corectă a primei clauze a predicatului “pcn”.

3. Desfăşurărea lucrării

Se vor testa toate exemplele prezentate în lucrare pe diferite seturi de test şi se va trasa execuția pentru fiecare exemplu pentru cel puțin un set de date de intrare.

4. Întrebări şi probleme

4.1. Să se scrie un predicat care determină toate aparițiile de variabile care apar în definiția unui predicat Prolog.

4.2. Să se modifice programul care calculează produsul cartezian astfel încât să genereze tuple de forma (e_1, ..., e_n) în locul tuplelor de forma [e_1, ..., e_n].

4.3. Scrieți un program original care să ilustreze facilitățile oferite de metaprogramare.

BIBLIOGRAFIE

1. **Kowalski**, **R.,** *Logic for Problem Solving*, Elsevier/North Holland, Amsterdam, 1979.
2. **Clocksin, W.F., Mellish**, **C.S.,** *Programming in Prolog*, Springer Verlag, New York, 1981.
3. **Lloyd, J.W.,** *Foundations of Logic Programming*, Springer Verlag, New York, 1984.
4. **Sterling, L., Shapiro, E.,** *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, Ma, 1986.
5. **Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjoland, T.,** *SICStus Prolog User's Manual,* SICS, 1996.
6. ***, *Turbo Prolog 2.0, User's Guide*, Borland, 1988.