7

Structuri de date

7.1. Tehnici de sortare a secvențelor

Sortarea este una din cele mai importante activități de prelucrare a datelor. Există mulți algoritmi de sortare a secvențelor. Aici, însă, vom considera patru algoritmi diferiți implementați în Prolog: sortarea naivă, sortarea cu inserție, sortarea prin metoda bulelor și sortarea rapidă.

Fiecare program va utiliza predicatul *precede(X,Y)* ce este adevărat pentru numere, dacă $X \leq Y$. Deci, schimbând numai acest predicat, vom putea sorta diferite tipuri de elemente.

7.1.1. Sortarea naivă

Sortarea naivă generează permutările elementelor din listă şi apoi verifică dacă ea este sortată, fie ascendent.

Să aducem programul de generare a unei permutări a elementelor unei liste. Pentru aceasta se utilizează predicatul *eliminare_elem/3*. Să observăm că o permutare a unei liste se obține prin permutarea cozii listei şi apoi prin inserția capului listei într-o poziție posibilă.

```
permutare([],[]).
permutare([Cap|Coada],L):-
       permutare(Coada,L1),
       eliminare_elem(Cap,L,L1).
```

Combinând predicatele de mai sus, se obține programul de sortare naivă.

```
sortare_naiva(L,ListaSortata):-
       permutare(L,ListaSortata),
       ascendenta(ListaSortata).
```

Programul descris e uşor de înțeles. Însă, el este iremediabil de ineficient. Motivul este că sunt *n!* permutări posibile a *n* elemente. De exemplu, *5!=120*, adică, Prologul poate verifica până la *120* posibilități. Dar *10!=3628800* şi *25!=15511210043330985984000000* - este clar că Prologul nu le va putea examina. Să observăm că permutarea se face înainte de a verifica dacă lista e sortată ascendent. Dacă măcar o sublistă nu e sortată, ar fi bine ca permutarea să fie respinsă.

7.1.2. Sortarea prin inserție

Sortarea prin inserție este, probabil, cea mai naturală metodă care deseori se utilizează şi manual. Capul unei liste este eliminat, coada se sortează, apoi capul se inserează în poziția corespunzătoare a sublistei sortate.

```
sortare_insertie([],[]).
sortare_insertie([C|Cd],LSort):-
       sortare_insertie(Cd,LSort1),
       inserare(C,LSort1,LSort).
```

Inserarea este prezentată de următoarele clauze:

```
inserare(Elem,[],[Elem]).
inserare(Elem,[C|Cd],[C|Cd1]):-
       precede(C,Elem),
       inserare(Elem,Cd,Cd1).
inserare(Elem,[C|Cd],[Elem,C|Cd]):-
       precede(Elem,C).
```

Însă, acest program de sortare a secvenței are o particularitate puțin stranie:

```
?-sortare_insertie([4,1,3,1,7,2],L).
L=[1,1,2,3,4,7];
L=[1,1,2,3,4,7]
```

Prologul la întrebarea de mai sus produce două soluții. Cauza constă în prezența în listă a celor două elemente 1 care se inserează în ambele posibile locuri. Aceasta ar putea fi evitată, dacă s-ar utiliza o altă definiție a predicatului *precede/2*.

7.1.3. Metoda bulei

Sortarea prin metoda bulei este o altă tehnică de sortare care presupune examinarea perechilor de elemente adiacente, schimbându-le cu locul, dacă nu corespund ordinii impuse de predicatul *precede/2*. După ce toate perechile sunt examinate, lista rezultată devine sortată. Următorul program, ce realizează această idee, este un exemplu de programare proastă în Prolog.

```
sortare_bubble(Lista,ListaSortata):-
       concatinare(X,[A,B|Y],Lista),
       precede(B,A),
       concatenare(X,[B,A|Y],M),
       sortare_bubble(M,ListaSortata).
sortare_bubble(Lista,Lista):-
       ascendenta(Lista).
```

Acest program este extrem de ineficient

```
?-sortare_bubble([5,4,3,2,1],Lista).
Lista=[1,2,3,4,5]
```

şi are două dezavantaje. Primul dezavantaj este că programul nu e scris în stil declarativ. El rău utilizează cunoştințele despre ordinea în care Prologul încearcă să efectueze sortarea. Al doilea dezavantaj este că scopul *precede(B,A)* întotdeauna va reuşi, dacă *A=B* şi deci, programul va avea o buclă infinită, dacă două elemente în listă sunt egale. Motivul e în faptul că sortarea prin metoda bulelor descrisă mai sus este un concept procedural. De aceea el este ideal pentru implementarea într-un limbaj procedural, cum ar fi C. Deci, prin acest exemplu se vede că este posibilă scrierea în Prolog a unui algoritm procedural, rezultatul, însă, nu este, bineînțeles, elegant.

7.1.4. Sortarea rapidă

Algoritmul de sortare rapidă e mult mai potrivit pentru implementare în Prolog. Algoritmul se bazează pe următoarea idee inteligentă. Considerăm lista *[Cap|Coada]*. Lista *Coada* se poate diviza în două liste *L* şi *M,* în aşa fel ca *L* să conțină elementele mai mici decât *Cap*, iar *M* - elementele mai mari decât *Cap*. Atunci lista sortată va fi compusă din versiunea

sortată a listei *L*, apoi *Cap* şi apoi versiunea sortată a listei *M*. La rândul său, listele *L* şi *M* sunt sortate recursiv prin sortarea rapidă.

Mai întâi, considerăm algoritmul de divizare a listei. Vom utiliza predicatul *divizare(Cap,Coada,L,M)* pentru împărţirea listei *[Cap|Coada]* în listele *L* şi *M*.

```
divizare(Cap,[A|X],[A|Y],Z):-
       precede(A,Cap),
       divizare(Cap,X,Y,Z).
divizare(Cap,[A|X],Y,[A|Z]):-
       precede(Cap,A),
       divizare(Cap,X,Y,Z).
divizare(Cap,[],[],[]).
```

Atunci algoritmul de sortare rapidă e reprezentat de programul

```
sortare_rapida([C|Cd],LSort):-
       divizare(C,Cd,L,M),
       sortare_rapida(L,L1),
       sortare_rapida(M,M1),
       concatenare(L1,[C|M1],LSort).
sortare_rapida([],[]).
```

Acest algoritm este foarte eficient şi lucrează bine pe secvenţe foarte mari.

7.2. Tratarea mulţimilor

Mulţimile în programare sunt una din structurile de bază. Multe probleme sunt legate de căutarea soluţiilor folosind mulţimile. Unele limbaje de programare, de exemplu, Pascal şi Modula-2, încorporează mulţimile şi operaţiile standarde asupra lor ca parte a limbajului. Alte limbaje, inclusiv Prolog, nu au mulţimi în calitate de structuri predefinite. Responsabilitatea este a programatorilor să definească mulţimi şi să elaboreze predicate ce imit operaţiile pe mulţimi, utilizând facilităţile disponibile din limbaj.

7.2.1. Reprezentarea explicită prin liste

Această tehnică utilizează listele pentru reprezentarea mulţimilor şi este cea mai simplă şi deci, mai frecvent aplicată. Mulţimile se reprezintă şi se manipulează explicit, scriind elementele într-o listă. De exemplu, mulţimea *{2, 3, 5}* e reprezentată în Prolog cu *[2, 3, 5]*. Mai există şi alte variante ale acestei reprezentări, dar mai întâi să argumentăm necesitatea ei. Motivul principal constă în faptul că în Prolog nu există nici o facilitate predefinită pentru reprezentarea mulţimilor. De aceea mulţimile se substituie deseori cu liste. Mulţimile şi listele, însă, sunt structuri de date diferite. Pentru a implementa o mulţime prin intermediul listei, ultimei se impune unele caracteristici specifice mulţimilor.

Precum s-a mai spus, sunt două deosebiri majore între o listă şi o mulţime. Una este că ordinea elementelor într-o listă e esenţială, pe când într-o mulţime - nu. De exemplu, *[2, 3, 5]* şi *[3, 5, 2]* sunt liste diferite, dar ele reprezintă aceeaşi mulţime. A doua deosebire este că mulţimea este o colecţie de obiecte distincte. Astfel, când se concatenează listele *[2, 3, 5]* şi *[5, 6]*, obţinem *[2, 3, 5, 5, 6]*. Din altă parte, uniunea mulţimilor *[2, 3, 5]* şi *[5, 6]* este *[2, 3, 5, 6]*.

Reprezentarea prin liste nesortate, de exemplu, *[3, 5, 2]*, este, probabil, cea mai obişnuită formă de reprezentare a mulţimilor în Prolog. În secţiunile precedente se vorbea despre predicatul *apartine/2*. În baza lui poate fi definit predicatul *submultime/2,* care este adevărat, dacă primul argument este o submulţime a argumentului doi.

```
submultime([],_).
submultime([X|Xs],Ys):-
       apartine(X,Ys),
       submultime(Xs,Ys).
```

Predicatul *echivalente(Xs,Ys)* este adevărat, dacă mulţimile *Xs* şi *Ys* sunt echivalente.

```
echivalente(Xs,Ys):-
       submultime(Xs,Ys),
       submultime(Ys,Xs).
```

Uniunea a două mulţimi poate fi exprimată prin următoarele clauze:

```
uniune([],Ys,Ys).
uniune([X|Xs],Ys,[X|Zs]):-
       not(apartine(X,Ys)),!,
       uniune(Xs,Ys,Zs).
uniune([_|Xs],Ys,Zs):-
       uniune(Xs,Ys,Zs).
```

iar intersecţia a două mulţimi prin

```
intersectie([],_,[]).
intersectie([X|Xs],Ys,[X|Zs]):-
       apartine(X,Ys),!,
       intersectie(Xs,Ys,Zs).
intersectie([_|Xs],Ys,Zs):-
       intersectie(Xs,Ys,Zs).
```

Tehnica reprezentării prin liste sortate, fiind de fapt o variantă a tehnicii de mai sus, impune o restricţie internă - elementele listei sunt sortate. Să notăm, că aici nu se schimbă regula de bază a mulţimilor, care ne spune că ordinea elementelor nu este esenţială. Noi impunem această restricţie doar în reprezentarea internă.

Acum vom aduce definiţiile operaţiilor pe mulţimi, reprezentate de liste sortate: uniunea, intersecţia şi diferenţa. Aici se foloseşte predicatul *precede/2,* care este adevărat, dacă primul argument precede în listă elementul al doilea.

```
uniune([],Ys,Ys) .
uniune(Xs,[],Xs).
uniune([X|Xs],[X|Ys],[X|Zs]):-!,
       uniune(Xs,Ys,Zs).
uniune([X|Xs],[Y|Ys],[X|Zs]):-
       precede(X,Y),
       uniune(Xs,[Y|Ys],Zs).
uniune([X|Xs],[Y|Ys],[Y|Zs]):-
       precede(Y,X),
       uniune([X|Xs],Ys,Zs).

intersectie([],_,[]).
intersectie(_,[],[]).
intersectie([X|Xs],[X|Ys],[X|Zs]):-!,
       intersectie(Xs,Ys,Zs).
intersectie([X|Xs],[Y|Ys],Zs):-
       precede(X,Y),
       intersectie(Xs,[Y|Ys],Zs).
intersectie([X|Xs],[Y|Ys],Zs):-
       precede(Y,X),
       intersectie([X|Xs],Ys,Zs).

diferenta([],_,[]).
diferenta(Xs,[],Xs).
diferenta([X|Xs],[X|Ys],Zs):-!,
       diferenta(Xs,Ys,Zs).
diferenta([X|Xs],[Y|Ys],[X|Zs]):-
       precede(X,Y),
       diferenta(Xs,[Y|Ys],Zs).
diferenta([X|Xs],[Y|Ys],Zs):-
       precede(Y,X),
       diferenta([X|Xs],Ys,Zs).
```

Dacă comparăm tehnicile descrise, principalele avantaje şi dezavantaje ţin de complexitatea algoritmilor. Tehnica de reprezentare prin liste sortate necesită mai mult timp decât tehnica de reprezentare prin liste nesortate pentru construirea mulţimilor şi mai puţin timp pentru operaţiile ce se aplică asupra submulţimilor. Trebuie să se ţină cont de următoarea regulă. Dacă mulţimile sunt mai mult sau mai puţin statice, adică nu se prea modifică, atunci se aplică tehnica listelor sortate, în caz contrar – tehnica listelor nesortate.

Fie mulţimea *Xs* are *m* elemente şi mulţimea *Ys* are *n* elemente. La început, întrucât mulţimea *Xs* se sortează, se va cere *O(m*log(m))* comparaţii, iar cealaltă metodă nu va necesita cheltuieli de timp. Inserarea unui element în *Xs,* folosind tehnica listelor sortate, necesită *O(m)* sau *O(log(m))* comparaţii în dependenţă de algoritmul utilizat. În cazul listelor nesortate e necesar doar *O(1)* timp, fiindcă un element poate fi inserat la începutul listei. Însă, pentru a face uniunea a două liste, tehnica listei sortate necesită *O(m+n)* timp, pe când tehnica listei nesortate - *O(m*n)* comparaţii.

Mulţimilor li se pot asocia un nume. În tehnicile precedente numele unei mulţimi nu era nicicum legat de ea, cum, spre deosebire, se face în matematică:

a={3, 5, 2}
b={5, 6}

Dar şi în unele aplicaţii putem dori ca mulţimile să fie legate de numele ce le posedă. O cale explicită este cea de utilizare a faptelor cu liste:

```
a([3,5,2]).
b([5,6]).
```

Adică, fiece fapt reprezintă o mulţime, iar functorul faptului este numele mulţimii.

Această tehnică poate fi considerată o variantă a tehnicii listelor nesortate. Aici orice mulţime e reprezentată *nume(Xs)*, dar în tehnica listelor nesortate se utilizează numai *Xs*. Deci, izolându-l pe *Xs,* putem aplica predicatele cunoscute. De exemplu,

```
apartine1(X,nume(Xs)):-
       apartine(X,Xs).
```

va determina dacă *X* este un element al mulţimii *Xs.* Similar

```
submultime1(nume1(Xs),nume2(Ys)):-
       submultime(Xs,Ys).
```

va determina dacă mulţimea *nume1(Xs)* este o submulţime a mulţimii *nume2(Ys).*

Dacă functorul faptelor este *mulţime*, atunci mulţimile *a* şi *b* sunt reprezentate astfel

```
multime(a,[3,5,2]).
multime(b,[5,6]).
```

Functorul *mulţime* este definit de utilizator, deci aici poate fi utilizat orice nume. Această formă de reprezentare indică explicit care fapte din baza de fapte reprezintă mulţimi. Ea, la fel, este o variantă a tehnicii de reprezentare prin liste nesortate. De exemplu,

```
apartine2(X,Nume):-
       multime(Nume,Xs),
       apartine(X,Xs).
```

va determina dacă *X* este un element al mulţimii cu numele *Nume*. Bineînţeles, putem considera aici încă o variantă, când lista din fapt, ce reprezintă mulţimea, este sortată.

7.2.2. Reprezentarea explicită a elementelor mulţimii prin fapte

Orice element al mulţimii e reprezentat de un fapt Prolog cu două argumente. Primul argument este numele mulţimii, iar al doilea este însuşi elementul. Deci, mulţimile *a={3,5,2}* şi *b={5,6}* pot fi reprezentate astfel

```
elem(a,3).
elem(a,5).
elem(a,2).
elem(b,5).
elem(b,6).
```

Reprezentările prin liste sunt mai compacte decât ultima. De asemenea, reprezentarea prin liste este mai potrivită pentru reprezentarea diferitor variante de mulţimi cum ar fi mulţimea vidă, *[]*, sau a unei mulţimi de mulţimi *[[2],[2,3]]*. Există, însă, situaţii când reprezentarea elementelor prin fapte este utilă. Momentul cheie este că această reprezentare este o simplă colecţie de fapte - componentă de bază a unui program Prolog.

Această reprezentare poate fi adusă la formele de reprezentare precedente, aplicând doi paşi. Primul pas transformă datele într-o listă. La pasul 2 se aplică operaţiile pe mulţimi asupra noii structuri, adică tehnicile cunoscute.

Există unele avantaje în utilizarea faptelor. Odată ce transformarea e făcută, fiind cunoscute tehnicile reprezentării prin liste, scrierea programului de mai departe devine simplă. Un alt, posibil, avantaj este că operaţiile asupra submulţimilor (sublistelor) sunt de obicei mai eficiente decât lucrul direct cu faptele.

Acum să încercăm să construim o listă din colecţia de fapte. De exemplu, din

```
elem(a,3).
elem(a,5).
elem(a,2).
```

dorim să obţinem un fapt nou

```
multime(a,[3,5,2]).
```

Cu toate că ordinea elementelor în mulţime e imaterială, e bine uneori, pentru comparaţie, să păstrăm aceeaşi ordine ca în faptele iniţiale.

În unele implementări Prolog există un predicat predefinit *findall/3*. Dacă se formulează întrebarea

```
?-findall(X,elem(a,X),Lista)
```

sistemul va răspunde

```
Lista=[3,5,2]
```

După unele operaţii cu mulţimi, uneori, se doreşte descompunerea mulţimilor înapoi în fapte. De exemplu, fiind dată

```
multime(c,[4,5]).
```

se cere generarea

```
elem(c,4).
elem(c,5).
```

Pentru aceasta putem utiliza predicatele

```
construire_elem(M):-
```

```
        multime(M,Es),
        generare(M,Es).

generare(_,[]):-!.
generare(M,[E|Es]):-
        assert(elem(M,E)),
        generare(M,Es).
```

Iar întrebarea

```
?-construire_elem(c)
```

va produce rezultatul aşteptat.

Să examinăm acum tehnica de operare direct cu fapte**.** Aici, pentru a verifica dacă un element *E* aparţine mulţimii cu numele *M,* e destul să se scrie

```
?-elem(M,E)
```

Să considerăm, acum, relaţia de submulţime. Fie faptele

```
elem(a,2).
elem(a,5).
elem(b,2).
elem(b,3).
elem(b,5).
```

Vom exprima noţiunea de submulţime prin noţiunea de a nu fi submulţime. Atunci, mulţimea reprezentată de *a* este o submulţime a mulţimii *b*, dacă predicatul ce exprimă relaţia de a nu fi submulţime dintre *a* şi *b* eşuează. Deci, trebuie de scris un predicat *submultime(A.B)* care reuşeşte, dacă *A* e submulţime a mulţimii *B* şi nu reuşeşte, în caz contrar. E evident că cazul special *submultime(A,A)* întotdeauna reuşeşte. Pentru exemplul de mai sus, *submultime(a,b)* trebuie să reuşească, în timp ce *submultime(b,a)* - să eşueze.

```
submultime(A,B):-
        not(nu_e_submultime(A,B)).

nu_e_submultime(A,B):-
        elem(A,E),
        not(elem(B,E)).
```

Două mulţimi sunt echivalente, dacă ele sunt submulţimi una alteia. Acest predicat seamănă cu cel din reprezentarea prin liste, numai că în calitate de argumente aici apar nume de mulţimi şi nu însăşi mulţimile.

```
echivalente(A,B):-
        submultime(A,B),
        submultime(B,A).
```

Predicatul *e_in_uniune(E,A,B)* este adevărat, dacă elementul *E* este în uniunea mulţimilor *A* şi *B*.

```
e_in_uniune(E,A,B):-elem(A,E),!.
e_in_uniune(E,A,B):-elem(B,E).
```

Predicatul *e_in_intersectie(E,A,B)* este adevărat, dacă elementul *E* este în intersecţia mulţimilor *A* şi *B*.

```
e_in_intersectie(E,A,B):-
        elem(A,E),
        elem(B,E).
```

Predicatul *e_in_diferenta(E,A,B)* este adevărat, dacă elementul *E* este în diferenţa mulţimilor *A* şi *B*.

```
e_in_diferenta(E,A,B):-
        elem(A,E),
        not(elem(B,E)).
```

Menţionăm că mai sus, realmente, nu s-au produs operaţiile uniunea, intersecţia şi diferenţa. Pentru a genera o nouă mulţime *C* în calitate de uniune a mulţimilor *A* şi *B,* ultimele trebuie să se transforme în liste, apoi să se utilizeze tehnicile respective şi, în sfârşit, lista să se desfacă iarăşi în fapte. Aceeaşi procedură poate fi aplicată, de asemenea, pentru intersecţie şi diferenţă.

Presupunem că mulţimea *A* are *m* elemente, iar mulţimea *B* are *n* elemente. Predicatele *submultime/2* şi *echivalente/2*, de exemplu, scanând faptele ce includ elementele mulţimilor *A* şi *B,* fac *O(m*n)* comparaţii. Predicatele *e_in_uniune/3, e_in_intersectie/3* şi *e_in_diferenta/3* necesită *O(m+n)* comparaţii.

7.2.3. Descrierea implicită a elementelor mulţimilor

Aici elementele mulţimilor se descriu de caracteristicile lor. Astfel, mulţimile în Prolog pot fi definite prin predicate. O serie de operaţii pe mulţimi pot fi aplicate asupra argumentelor predicatelor, chiar dacă nu se recunoaşte că predicatele reprezintă mulţimi.

Metodele implicite au unele avantaje şi dezavantaje. Un dezavantaj poate fi considerat faptul că nu întotdeauna pot fi uşor efectuate operaţiile pe mulţimi exprimate implicit. Dar îndată ce această dificultate este depăşită, apar unele avantaje. În primul rând, această reprezentare este foarte compactă. Dacă sunt necesare elementele în mod explicit, atunci acest lucru rămâne pe seama calculatorului. În al doilea rând, e clar că metoda explicită nu poate descrie o mulţime infinită. Pentru metoda implicită nu există astfel de restricţii. În al treilea rând, poate fi uşor văzută natura elementelor privind doar caracteristicile lor.

Să considerăm, de exemplu, reprezentarea implicită a numerelor Fibonacci. Pentru numărul *N*, unde *N≥0,* numărul Fibonacci, notat F_N, se defineşte

$F_N=0$, pentru $N=O$
$F_N=1$, pentru $N=1$
$F_N=F_{N-1}+F_{N-2}$, pentru $N\geq 2$

În Prolog, putem defini predicatul *fib(N,FN)* care primeşte valoarea adevăr, dacă F_N este al N^{ea} număr Fibonacci.

```
fib(0,0).
fib(1,1).
fib(N,FN):-
        N>=2,
        N1 is N-1, N2 is N-2,
        fib(N1,FN1),fib(N2,FN2),
        FN=FN1+FN2.
```

Acest predicat descrie caracteristica numerelor Fibonacci. El poate fi considerat o reprezentare implicită a lor.

Să ne oprim asupra a trei tehnici de manipulare a mulţimilor reprezentate implicit.

Una din tehnici presupune reducerea la liste a tuturor mulţimilor participante la operaţii. Această tehnică include doi paşi. La primul pas, utilizând caracteristica, se generează elementele mulţimilor în mod explicit. La pasul 2 se aplică operaţiile pe mulţimi conform tehnicilor de reprezentare prin liste. Este evident că această metodă nu poate fi aplicată, dacă din diferite motive nu pot fi generate toate elementele.

Astfel, utilizând predicatul *fib/2*, se poate genera mulțimea de numere Fibonacci reprezentate explicit. Predicatul *fibmultime(N,M)* reuşeşte, dacă *M* este mulțimea primelor *N* numere Fibonacci.

```
fibmultime(0,[0]).
fibmultime(N,M):-
        N>=1,
        fib(N,FN),
        N1 is N-1,
        fibmultime(N1,M1),
        concatinare(M1,[FN],M).
```

Menționăm că nu întotdeauna e necesară generarea tuturor elementelor în formă explicită. De exemplu, pentru a determina dacă un obiect este un element al unei definiții implicite, e de ajuns să se introducă obiectul în definiție. Dacă definiția reuşeşte, atunci obiectul e element al mulțimii reprezentate implicit. În caz contrar - nu este.

Pot fi aplicate operații când o mulțime este reprezentată explicit, iar alta implicit. Fie *A* o mulțime definită explicit, iar *B* o mulțime definită implicit. Pentru a determina dacă *A* este o submulțime a lui *B*, se verifică dacă orice element din *A* aparține şi lui *B*. Pentru a calcula mulțimea *C,* intersecție a mulțimilor *A* şi *B*, se lucrează cu orice element din *A* şi se verifică dacă el este şi în *B*. Dacă da, atunci elementul se include ca element în *C*. Pentru a face, însă, uniunea mulțimilor *A* şi *B*, trebuie generate toate elementele şi din *B*.

Bineînțeles că pot fi aplicate operații asupra mulțimilor reprezentate implicit fără a genera elementele în formă explicită. Aceasta e o tehnică atractivă. În general, însă, aplicarea operațiilor este mai dificilă decât în cazul reprezentării explicite.

Astfel, pot fi simulate operațiile uniunea şi intersecția, utilizând conjuncții şi disjuncții de subscopuri. Fie predicatele *a* şi *b* definesc implicit mulțimile *A* şi *B*, respectiv. Pentru a genera reprezentarea implicită a uniunii mulțimilor *A* şi *B,* se construieşte un predicat *c*:

```
c:-a.
c:-b.
```

Similar pentru reprezentarea intersecției mulțimilor date se va scrie predicatul

```
c:-a,b.
```

Aici trebuie de menționat că prelucrarea implicită a mulțimilor depinde mult de faptul dacă varianta Prolog, ce este la dispoziție, susține memorarea şi ştergerea dinamică a regulilor sau numai a faptelor.