3. MECANISMUL DE OPERARE ÎN PROLOG

3.1. Satisfacerea clauzelor

Vom explica în acest capitol modul în care se realizează satisfacerea scopului unui program în PROLOG. Convenim să numim mulțimea de fapte dintr-un program ***baza de fapte***, iar mulțimea de reguli ***baza de reguli***. În prima etapă vom considera că programul conține doar fapte (deci nu conține reguli). Presupunem că scopul are forma p_1, p_2,…, p_n. Dacă scopul nu conține variabile atunci clauzele sunt satisfăcute în ordinea în care sunt enumerate. Dacă baza de fapte poate satisface întregul scop, PROLOG răspunde cu "yes", iar în caz contrar răspunde cu "no". Dacă scopul conține variabile, atunci obiectivul este acela de a găsi, toate legăturile posibile pentru variabile, care să satisfacă scopul. Să considerăm următorul program:

```
domains
        nume_pers = symbol
predicates
        debitor(nume_pers, nume_pers)
        creditor(nume_pers, nume_pers)
clauses
        debitor(popescu, ionescu).
        debitor(georgescu, ionescu).
        creditor(cristescu, popescu).
        creditor(vasilescu, ionescu).
```

Prezentăm modul în care PROLOG răspunde la următoarele interogări:

```
Goal : debitor(X, ionescu)
X = popescu
X = georgescu
2 Solutions

Goal : debitor(popescu, X), creditor(vasilescu, X)
X = ionescu
1 Solution

Goal : debitor(popescu, X), creditor(cristescu, X)
No Solution
```

În cazul primei interogări cele două soluții corespund celor două fapte asociate predicatului debitor care au al doilea argument identic cu obiectul ionescu. Pentru satisfacerea acestui scop, la întâlnirea faptului debitor(popescu, ionescu) are loc legarea variabilei X la obiectul popescu. După aceasta se aplică următorul principiu de bază al programării în PROLOG:

Baza de cunoştinţe este inspectată în vederea satisfacerii scopului. După fiecare satisfacere a scopului, în cazul în care scopul conține nume de variabile, se încearcă o resatisfacere a acestuia în vederea aflării tuturor soluțiilor care satisfac scopul. Pe măsură ce faptele sunt utilizate în satisfacerea scopului, acestea sunt "marcate". Înainte de a resatisface scopul, variabilele sunt dezlegate de valorile obținute, după care se inspectează baza de cunoştinţe pornind de la ultimul semn de marcaj, în vederea obținerii unei alte legături posibile.

Prezența variabilei anonime pe locul unui argument precizează că nu interesează valoarea legată de argument ci numai existența unei asemenea valori. Pentru a face rolul variabilei anonime mai explicit vom considera două interogări relative la programul precedent:

```
Goal : debitor(_, ionescu)
Yes
Goal : creditor(ionescu, _)
No
```

Revenind la cazul general, dacă scopul $p_1, p_2, \ldots, p_n$ conține variabile, atunci se parcurge baza de fapte şi se satisface p_1. Se trece apoi la satisfacerea lui p_2. Dacă p_2 poate fi satisfăcută, se trece la clauza p_3. Dacă p_2 nu poate fi satisfăcută, atunci se dezleagă variabilele lui p_1 şi se inspectează baza de fapte pentru a găsi alte valori care legate la variabilele lui p_1 să satisfacă p_1, după care se încearcă din nou satisfacerea lui p_2. Are loc un fenomen de revenire în lista clauzelor din scop, fenomen numit ***backtracking***. Algoritmul de satisfacere a scopului $p_1, p_2, \ldots, p_n$ în cazul în care programul conține doar fapte poate fi descris de procedura prezentată mai jos. În această procedură V_i desemnează mulțimea variabilelor care apar în p_i şi nu apar în $p_1, p_2, \ldots, p_{i-1}$. Cu $\varnothing$ se notează mulțimea vidă.

Procedura alg_satisfacere_scop($p_1, p_2, \ldots, p_n$)
k : = 0
i : = 1
cât timp (i > 0)
- ***dezleagă variabilele din V_i***

j : = 1
cât timp (j = 1)
- ***inspectează baza de fapte în continuare pentru a găsi obiecte care legate de elementele lui V_i să satisfacă p_i. Fie B_i mulțimea acestor obiecte.***

dacă ($B_i = \varnothing$) atunci i : = i –1
j : = 0
altfel dacă (i < n) atunci i : = i + 1
altfel k : = k+1
- ***afişează valorile variabilelor din $V_1, V_2, \ldots, V_n$***

j : = 0
dacă (k = 0) atunci scrie : No Solution
altfel scrie : k Solutions

stop

Conform algoritmului de mai sus dacă variabilele clauzelor $p_1, \ldots, p_{i-1}$ sunt legate astfel încât aceste clauze să fie satisfăcute, se inspectează baza de fapte în vederea satisfacerii (resatisfacerii) lui p_i. Dacă nu se pot identifica legături noi ale variabilelor care apar în p_i şi nu au apărut în $p_1, \ldots, p_{i-1}$ (i.e. $B_i = \emptyset$), atunci prin backtracking se identifică acea clauză p_j cu $j \leq i-1$, care are proprietatea că variabilele care apar în p_j şi nu apar în $p_1, \ldots, p_{j-1}$ pot fi legate la alte obiecte care să satisfacă clauza p_j. În cazul în care se găseşte o astfel de clauză p_j se continuă cu satisfacerea clauzelor $p_{j+1}, p_{j+2}, \ldots$. În caz contrar (i.e. cazul în care $p_{i-1}, \ldots, p_1$ nu pot fi resatisfăcute), procesul se opreşte. Dacă nu s-a obținut nici o soluție PROLOG afişează mesajul "no".

Să presupunem că avem un program care conține şi reguli. Procedeul prin care PROLOG realizează satisfacerea clauzelor este ***unificarea***. Desemnăm prin ***termen*** un obiect elementar o variabilă sau un obiect complex. Procedura de unificare respectă următoarele reguli:

- O variabilă liberă se unifică cu orice termen. În urma unificării variabila devine legată la acel termen.
- O constantă se poate unifica cu ea însăşi sau cu o variabilă. După unificare variabila devine legată la acea constantă.
- O variabilă liberă se poate unifica cu o altă variabilă liberă. După unificare cele două variabile lucrează ca una şi aceeaşi variabilă. Dacă una din ele devine legată la o valoare, atunci şi cealaltă se consideră legată la aceeaşi valoare.
- Un termen complex se unifică cu alt termen complex dacă
  - amândoi reprezintă obiecte de acelaşi tip
  - toți subtermenii unuia din termeni unifică cu subtermenii asociați din celălalt termen.
- Un predicat se unifică cu alt predicat dacă
  - cele două predicate au acelaşi nume
  - toți termenii asociați din cele două predicate unifică.

Putem descrie acum algoritmul de satisfacerea scopului, $p_1, \ldots, p_n$. Presupunem că $p_1, \ldots, p_{i-1}$ au fost satisfăcute. În vederea satisfacerii lui p_i se inspectează baza de fapte şi baza de reguli. Dacă p_i se satisface cu un element al bazei de fapte, atunci se trece la p_{i+1}. În caz contrar se

inspectează baza de reguli şi se identifică prima din regulile neluate în considerare până în acel moment, al cărui cap se poate unifica cu p_i. Clauzele care compun corpul regulii identificate se consideră componente ale scopului. Dacă una din clauzele corpului nu poate fi satisfăcută, atunci se identifică o altă regulă al cărui cap să unifice cu p_i. În cazul în care nu există o asemenea regulă, prin backtracking se încearcă resatisfacerea clauzelor p_{i-1}, p_{i-2}, ….

Pentru a exemplifica aplicarea acestui algoritm, să adăugăm la faptele existente în programul de la începutul capitolului următoarele reguli:

```
are_obligatii(X, Y) :- debitor(Y, X).
are_obligatii(X, Y) :- creditor(X, Y).
```

care au semnificația evidentă: "dacă X este debitorul lui Y, atunci X are obligații față de Y" şi respectiv, "dacă Y este creditorul lui X, atunci X are obligații față de Y". Să urmărim în continuare răspunsul la câteva interogări.

```
Goal : are_obligatii(X, popescu).
X = ionescu
X = cristescu
2 Solutions
Goal : are_obligatii(X, ionescu).
X = vasilescu
1 Solution
Goal : are_obligatii(X, _).
X = ionescu
X = ionescu
X = cristescu
X = vasilescu
4 Solutions
```

3.2. Controlul revenirii pe urme – tăietura şi predicatul fail.

După cum am arătat mai înainte pentru satisfacerea scopului în PROLOG se utilizează tehnica backtracking. Procesul de resatisfacere prin backtracking poate fi oprit de programator cu

ajutorul predicatului ***cut***, simbolizat prin caracterul ***!***. Cut este un predicat fără argumente. În poziția în care apare interzice revenirea, blocând astfel căutarea de noi soluții. Semnul "!" aşezat între clauzele p_i şi p_{i+1} ale scopului blochează resatisfacerea clauzei p_i. Un efect asemănător se obține utilizând tăietura (semnul "!") în corpul unei reguli, deoarece pentru satisfacerea corpului unei reguli PROLOG utilizează backtracking.

Facem o comparație între programul anterior şi un program ale cărui reguli conțin tăieturi. Să presupunem că scopul programului este are_obligatii(X, Y). Pentru a putea explica mecanismul de căutare a soluțiilor să etichetăm clauzele din program.

(1) debitor(popescu, ionescu).

(2) debitor(georgescu, ionescu).

(3) creditor(cristescu, popescu).

(4) creditor(vasilescu, ionescu).

(5) are_obligatii(X, Y) :- debitor(Y, X).

(6) are_obligatii(X, Y) :- creditor(X, Y).

Reprezentăm arborele de căutare corespunzător scopului are_obligatii(X, Y). Săgețile indică modul în care se avansează şi respectiv, se revine la puncte deja explorate, în căutarea tuturor soluțiilor posibile.

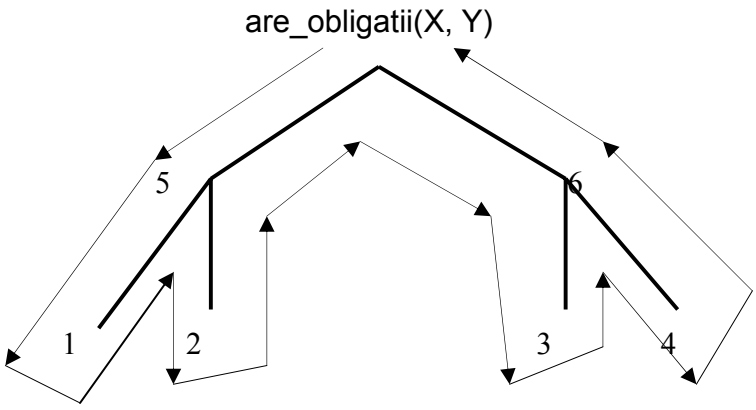


figura 1

Răspunsul la interogare este:

X = ionescu, Y = popescu

X = ionescu, Y = georgescu

X =cristescu, Y = popescu

X = vasilescu , Y = ionescu

4 Solutions

Să modificăm, cele două reguli anterioare, introducând tăietura

(5) are_obligatii(X, Y) :- debitor(Y, X), !.

(6) are_obligatii(X, Y) :- creditor(X, Y), !.

şi să analizăm efectul său asupra modului în care se va parcurge arborele de căutare.

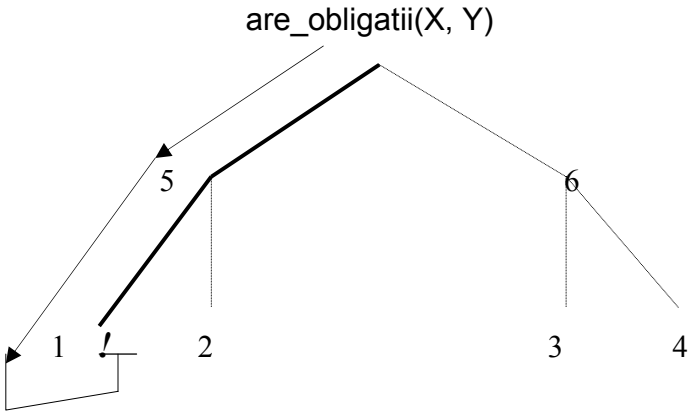


figura 2

După parcurgerea nodurilor 1 şi 3, se întâlneşte tăietura care blochează revenirea. Rezultatul obținut este

X = ionescu, Y = popescu

1 Solution

Un scop eşuează în PROLOG dacă toate încercările posibile de al satisface prin revenire pe urme (backtracking) eşuează. Eşuarea poate fi exprimată în PROLOG printr-un predicat predefinit, care are valoarea de adevăr "fals", şi anume predicatul ***fail***. Predicatul fail este utilizat pentru forțarea revenirii pe urme. Următoarele exemple vor pune în evidență forțarea backtrackingului cu ajutorul acestui predicat. Pentru a înțelege exemplele este necesar să

precizăm semnificaţia predicatelor “nl“ şi “write”. Predicatul “write” este un predicat predefinit a cărui valoare logică este “adevărat” şi care nu se resatisface niciodată. Clauza “write(X)” determină afişarea pe ecran a valorii legate de variabila X. Predicatul “nl” este un predicat predefinit care determină trecerea la linie nouă.

```
/*  Program 1 */
domains
        nume = symbol
predicates
        client(nume)
        afis_clienti
clauses
        client(popescu).
        client(ionescu).
        client(vasilecu).
afis_clienti : - client(X), write(X), nl.
```

```
/*  Program 2 */
domains
        nume = symbol
predicates
        client(nume)
        afis_clienti
clauses
        client(popescu).
        client(ionescu).
        client(vasilecu).
afis_clienti : - client(X), write(X), nl, fail.
```

La interogarea

Goal : afis_clienti

primul program va răspunde

```
popescu
Yes
```

în timp ce al doilea program va răspunde

```
popescu
ionescu
vasilescu
No
```

În primul program, pentru scopului afis_clienti se leagă variabila X la primul obiect care satisface clauza client(X), după care se afişează pe ecran obiectul respectiv (popescu). Deoarece predicatul write nu se resatisface nu se mai încearcă nici resatisfacerea scopului. Utilizarea, în cel de-al doilea program al predicatului fail în corpul clauzei determină dezlegarea variabilei X şi legarea ei la următorul obiect care face clauza client(X) adevărată. Mesajul "no" de la sfârşit este determinat de faptul că scopul eşuează. Putem evita obținerea acestui mesaj modificând programul în felul următor:

```
/*  Program 3 */
domains
        nume = symbol
predicates
        client(nume)
        afis_clienti
clauses
        client(popescu).
        client(ionescu).
        client(vasilecu).
```

```
afis_clienti : - client(X), write(X), nl, fail.
afis_clenti.
```

Răspunsul la aceeaşi interogare este

```
popescu
ionescu
vasilescu
Yes
```

3.3. Negaţia în PROLOG

Cu ajutorul predicatului fail se poate programa în PROLOG negaţia. Să presupunem că vrem să exprimăm următoarea regulă: "Dacă cererea pentru un produs creşte şi producţia nu creşte, atunci preţul produsului creşte". Aceasta se poate realiza cu programul următor:

```
domains
        produs=symbol
predicates
        creste_cerere(produs)
        creste_productie(produs)
      np(produs)
     creste_pret(produs)
clauses
        creste_cerere(mat_de_constructie).
        creste_cerere(confectii).
        creste_productie(confectii).
        np(X):- creste_productie(X),!,fail.   /*1*/
        np(_).                          /*2*/
        creste_pret(X):-creste_cerere(X),np(X).
```

Predicatul np(X) definit în acest program este satisfăcut dacă şi numai dacă preţul produsului X nu creşte. Folosim tăietura , !, pentru a evita activarea clauzei notate cu 2. Dacă pentru un anumit produs producţia creşte, tăietura va împiedica PROLOG să revină pe urme la regula 2, iar predicatul fail va duce la eşuarea capului regulii 1, deci va rezulta că pentru acel produs anume preţul nu creşte. Să considerăm un produs pentru care se menţionează în baza de fapte că cererea creşte. Dacă pentru acel produs nu se menţionează în baza de fapte că producţia creşte, atunci regula 1 va eşua, iar PROLOG va reveni pe urme, activând regula 2, al cărui cap va fi satisfăcut. Iată şi răspunsurile la câteva interogări:

```
Goal: creste_pret(confectii)
No
Goal: creste_pret(mat_de_constructii)
Yes
Goal: creste_pret(X)
X = mat_de_constructii
1 Solution
```

O încercare diferită de exprima negaţia în PROLOG o constituie predicatul special ***not***. Negaţia predicatului p(arg_1, arg_2,…, arg_n) se exprimă prin not(p(arg_1, arg_2,…, arg_n)). Predicatul ***not(p(arg_1, arg_2,…, arg_n))*** are valoarea logică "adevărat" dacă predicatul p(arg_1, arg_2,…, arg_n) nu se poate satisface. În caz contrar, are valoarea fals. Rescriem programul de mai sus folosind predicatul not.

```
domains
        produs=symbol
predicates
        creste_cerere(produs)
        creste_productie(produs)
        creste_pret(produs)
clauses
        creste_cerere(mat_de_constructie).
        creste_cerere(confectii).
        creste_productie(confectii).
```

```
creste_pret(X):-creste_cerere(X), not(creste_productie(X)).
```

Să considerăm un alt exemplu şi să analizăm răspunsurile sistemului PROLOG:

```
predicates
        patrat_perfect(integer)
clauses
        patrat_perfect(4).
        patrat_perfect(25).

Goal: patrat_perfect(16)
No
Goal: not(patrat_perfect(9))
Yes
```

Răspunsul "no" la prima interogare nu trebuie interpretat ca "16 nu este pătrat perfect". Ceea ce se înțelege de fapt prin răspunsul "no" este că, după ce s-au verificat toate clauzele acestui program, sistemul PROLOG nu poate considera 16 pătrat perfect. Răspunsul la a doua interogare este "yes" deoarece clauza patrat_perfect(9) eşuează neputând fi dedusă din cauzele programului. Acest mecanism de "gândire" al sistemului PROLOG se bazează pe ipoteza lumii închise (***Closed World Assumption***) : ***Dacă o afirmație A nu este exprimată nici direct, nici indirect într-un program P, ceea ce înseamnă că ea nu este nici fapt şi nici concluzie bazată pe datele unui program P, atunci acceptăm că este valabilă negația sa.***

3.4. Recursivitatea în PROLOG

Intuitiv recursivitatea se naşte în clipa în care o schiță este construită prin reproducerea ei însăşi, la o altă scară sau la un alt nivel. În cadrul programării recursivitatea este o tehnică care constă în posibilitatea de a include într-o procedură apeluri la ea însăşi. În PROLOG, acesta se traduce prin posibilitatea ca un nume de predicat să apară atât în corpul unei reguli cât şi în capul

ei. De exemplu, să presupunem că vrem să definim relația “strămoş” în vederea examinării următoarei părți a unui arbore de familie:

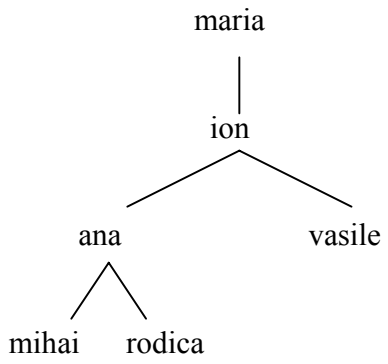


figura 3

Fiecare nod desemnează o persoană iar legătura directă între noduri reprezintă relația “părinte” (de exemplu ion este părintele anei şi al lui vasile). Soluția nerecursivă impune definirea de reguli pentru fiecare grad de descendență:

```
stramos(X, Y) : - parinte(X, Y).
stramos(X, Y) : - parinte(X, Z),  parinte( Z, Y).
stramos(X, Y) : - parinte(X, Z1),  parinte( Z1, Z2), parinte( Z1, Y).
```

De fiecare dată când mergem mai adânc în arborele de familie, trebuie să definim o nouă regulă. acest lucru este fără îndoială ineficient din punct de vedere al programării, deoarece un program este util şi eficient numai dacă rezolvă cazuri generale şi nu doar cazuri particulare ale unei probleme. Există o cale mai eficientă de a defini relația “strămoş”. Observăm că pentru orice X şi Y din arborele de familie, X este strămoş pentru Y dacă există un Z astfel încât X este părinte al lui Z şi Z este strămoş al lui Y. Astfel, putem scrie următoarea clauză:

```
stramos(X, Y) : - parinte(X, Z),  stramos( Z, Y).
```

În această definiție predicatul "stramos" apare şi în corpul regulii şi în corpul ei. Definițiile de acest fel se numesc ***definiții recursive***, iar relații pe care le desemnează se numesc ***relații recursive***. În exemplu nostru controlul relației "stramos" dintre maria şi mihai este ilustrat de nivelurile diagramei următoare:

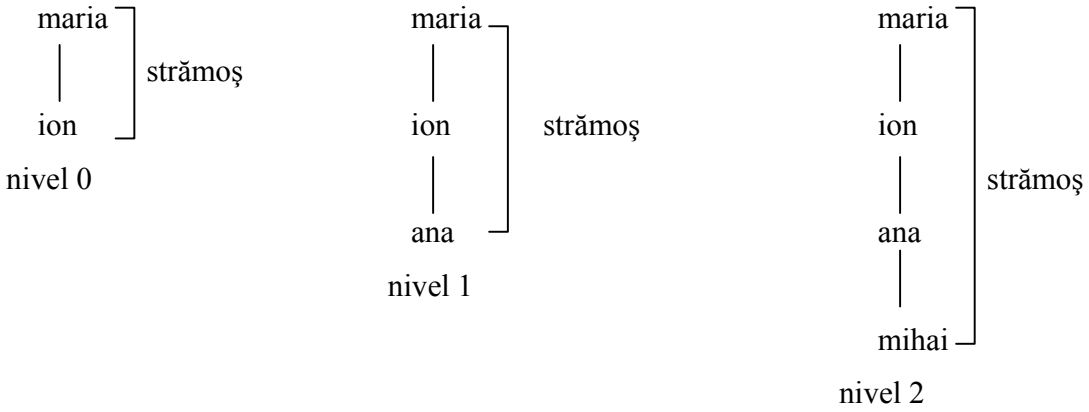


figura 4

Regula introdusă nu ajunge pentru definirea relației "strămoş". Deşi sub aspect logic exprimă ceea ce vrem să definim, ea nu dă următorului programului informația necesară pentru răspunsul la întrebări privind relația "strămoş":

```
domains
        persoana = symbol
predicates
        parinte(persoana, persoana)
        stramos(persoana, persoana)
clauses
        parinte(maria, ion).
        parinte(ion, ana).
        parinte(ion, vasile).
        parinte(ana, mihai).
        parinte(ana, rodica).
        stramos(X, Y) : - parinte(X, Z),  stramos( Z, Y).
```

Să presupunem că se pune următoare întrebare:

Goal: stramos(maria, ana)

PROLOG va încerca să răspundă căutând în spațiul stărilor descris de arbore următor:

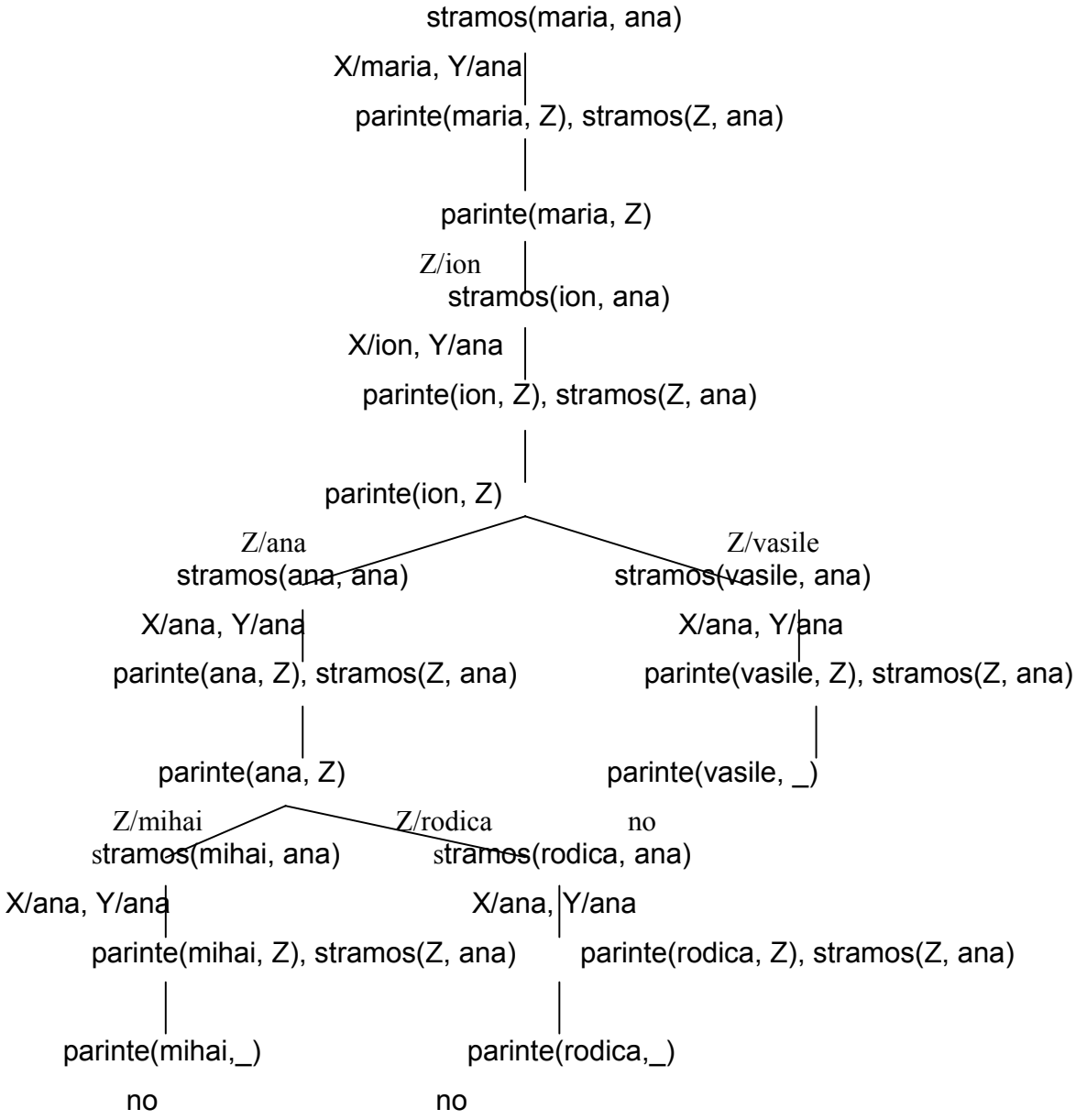


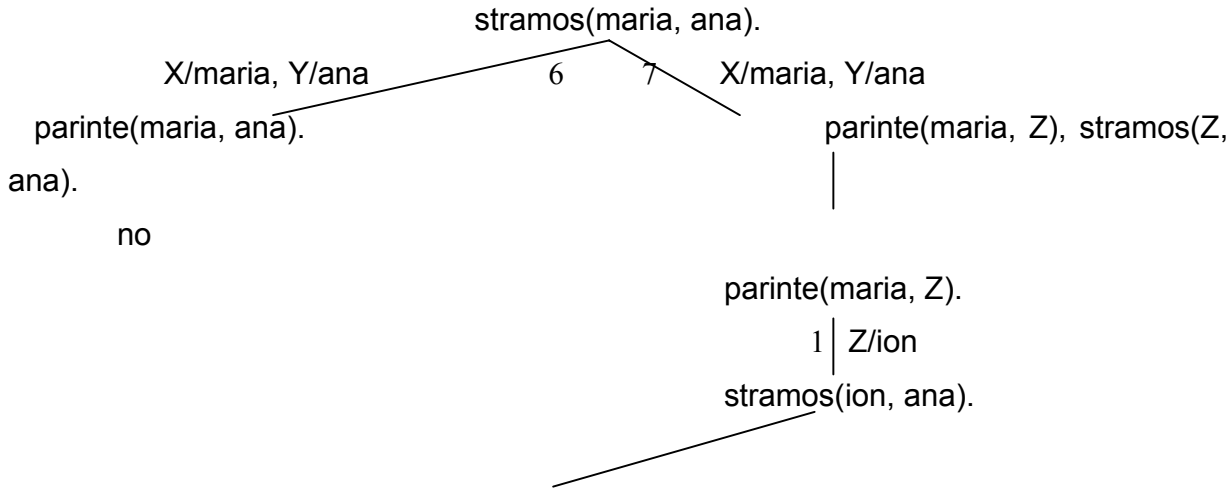
figura 5

PROLOG nu reuşeşte să termine corect căutarea în spațiul stărilor problemei, deoarece recurența nu are o condiție de oprire. Oprirea programului este provocată de terminarea parcurgerii arborelui de familie sau de limitările echipamentului pe care lucrează versiunea de PROLOG în

cauză dacă arborele este foarte adânc. Aşadar ne trebuie o metodă care garantează terminarea execuţiei unei proceduri recursive date. O astfel de metodă poate fi implementată folosind o clauză care enunţă ***condiţiile de margine*** în care o relaţie este validă. În exemplul nostru această condiţie este relaţia "părinte", care are sensul că o relaţie de "strămoş" între două persoane este valabilă la margine, când una dintre persoane este părintele celeilalte. De aceea trebuie să adăugăm în baza de reguli o clauză. Noul program este

```
domains
        persoana = symbol
    predicates
        parinte(persoana, persoana)
        stramos(persoana, persoana)
    clauses
        parinte(maria, ion).                    /*1*/
        parinte(ion, ana).                      /*2*/
        parinte(ion, vasile).                   /*3*/
        parinte(ana, mihai).                    /*4*/
        parinte(ana rodica).                    /*5*/
        stramos(X, Y) : - parinte(X, Y).            /*6*/
        stramos(X, Y) : - parinte(X, Z),  stramos( Z, Y).  /*7*/
```

Clauza 6 acţionează ca o condiţie de frontieră a relaţiei definite. Spaţiul stărilor care corespunde răspunsului la întrebarea stramos(maria, ana) este descris de arborele:



6

parinte(ion, ana).

yes

Astfel prin introducerea clauzei 6 PROLOG termină căutarea în spațiul stărilor şi răspunde pozitiv la întrebare.

În general, pentru rezolvarea recursivă a unei probleme, împărțim problema în două subgrupe (cf. []):

- Prima subgrupă conține instanța "simplă" a problemei (limita recurenței);
- A doua problemă conține instanțele "generale" ale problemei, ale căror soluții se găsesc reducându-le la versiuni mai simple ale problemei (recurență).

Considerăm util să încheiem acest capitol cu paragraf de concluzii:

- Toate clauzele care se referă la acelaşi nume de predicat trebuie grupate. Ordinea clauzelor în cadrul aceluiaşi grup influențează modul de execuție, deoarece ele sunt inspectate în vederea resatisfacerii în ordinea în care sunt trecute. Ordinea în care sunt aşezate grupurile de clauze nu prezintă importanță.
- Procedeul prin care PROLOG realizează satisfacerea clauzelor este unificarea. În urma unificării variabilele libere devin legate. Legarea variabilelor este locală (numele unei variabile dintr-o clauză nu are nici o legătură cu acelaşi nume dintr-o altă clauză).
- Pentru satisfacerea scopului, sau satisfacerea corpului unei reguli PROLOG utilizează tehnica backtrackingului. Forțarea backtrackingului se face predicatul fail. Procesul de resatisfacere prin backtracking poate fi oprit cu ajutorul tăieturii, !.
- Negația în PROLOG poate fi exprimată utilizând o combinație tăietură-eşuare (!,fail) sau utilizând predicatul not. Clauza not(p(arg_1, arg_2,…, arg_n)) este satisfăcută dacă şi numai dacă clauza p(arg_1, arg_2,…, arg_n) nu se poate satisface.
- PROLOG admite recursivitatea în sensul că acelaşi nume de predicat poate apare şi în capul unei reguli şi în corpul ei. În situația în care se utilizează definiții recursive este necesar să se definească condiții de margine (condiții de oprire a apelurilor recursive).