Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp şi Prolog. Aplicaţii în Inteligenţa Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

Capitolul 14

Controlarea backtracking-ului

14.1. Backtracking

Fie programul următor:

```
domains
        child = symbol
        age = integer
predicates
        player(child, age)
clauses
        player(peter, 9).
        player(paul, 10).
        player(chris, 9).
        player(susan, 9).
```

Dorim să aflăm toate perechile de copii în vârstă de 9 ani. Adresăm următoarea întrebare mediului Turbo Prolog:

```
player(P1, 9), player(P2, 9), P1 <> P2.
```

Iată mecanismul prin care Turbo Prolog găseşte soluţiile:

1. Prolog găseşte soluţie la primul sub-goal, mergând de sus în jos: P1 va fi legat de peter;
2. Prolog găseşte soluţie la al doilea subgoal: P2 va fi legat de peter;
3. Prolog încearcă să verifice al treilea subgoal: P1 şi P2 sunt ambele legate de peter, deci eşec;
4. Prolog revine la al doilea subgoal. Următoarea soluţie: P2 e legat de chris;
5. Prolog verifică condiţia a treia: P1 şi P2 sunt diferite, rezultă o soluţie:
   P1=peter, P2=chris
6. Prolog revine la al doilea subgoal. Următoarea soluţie: P2 e legat de susan;
7. Prolog verifică condiţia a treia: P1 şi P2 sunt diferite, rezultă o soluţie:
   P1=peter, P2=susan
8. Prolog revine la al doilea subgoal, dar nu mai găseşte alte soluţii;
9. Prolog revine la primul subgoal. Următoarea solutie: P1 e legat de chris;
10. Prolog trece la al doilea subgoal: P2 e legat de peter;
11. Prolog verifică condiţia a treia: P1 şi P2 sunt diferite, rezulta o soluţie:
    P1=chris, P2=peter

Procesul continuă în acelaşi mod. În final, Turbo Prolog găseşte şase soluţii:

```
P1=peter, P2=chris
P1=peter, P2=susan
P1=chris, P2=peter
```

```
P1=chris, P2=susan
P1=susan, P2=peter
P1=susan, P2=chris
6 Solutions
```

Observaţie. Turbo Prolog a găsit soluţii redundante. Mecanismul backtracking face de data aceasta căutare nenecesară.

Pentru alterarea mecanismului backtracking, Turbo Prolog oferă două predicate:

- **fail**, care forţează backtracking-ul;
- **!** (**cut**, tăietură), care împiedică backtracking-ul.

14.2. Utilizarea predicatului fail

Valoarea lui fail este eşec. Prin aceasta el încurajează backtracking-ul. Efectul lui este acelaşi cu al unui predicat imposibil, de genul 2 = 3. Fie următorul exemplu:

```
domains
        parametru = symbol
predicates
        predicat(parametru, parametru)
        toate
        toate1
clauses
        predicat(a, b).
        predicat(c, d).
        predicat(e, f).
        toate :-
                predicat(X, Y),
                write(X, " şi ", Y, "\n"),
                fail.
        toate1 :-
                predicat(X, Y),
                write(X, " şi ", Y, "\n").
```

Răspunsul la întrebarea

```
        toate1.
```

va fi

```
        a şi b
        Yes
```

iar răspunsul la întrebarea

```
        toate.
```

va fi

```
        a şi b
        c şi d
        e şi f
        No
```

Faptul că apelul predicatului toate se termină cu fail (care eşuează întotdeauna) obligă Turbo Prolog să înceapă backtracking prin corpul regulii toate. Prolog va reveni până la ultimul

apel care poate oferi mai multe soluţii. Un astfel de apel se numeşte apel nedeterminist. Predicatul write nu poate da alte soluţii, deci revine la apelul lui father.

Observaţie. Acel No de la sfârşitul soluţiilor semnifică faptul că predicatul toate nu a fost satisfăcut.

Observaţie. Dupa fail nu are rost să puneţi nici un predicat, deoarece Prolog nu va ajunge să-l execute niciodată.

Sistemul Prolog va răspunde în acelaşi fel la întrebările toate şi toate1 de mai sus, indiferent de modul în care sunt puse acestea: ca goal intern sau extern. Am precizat într-un capitol anterior că Turbo Prolog va răspunde la un goal intern cu prima soluţie, iar la un goal extern cu toate soluţiile. Dar, predicatele toate şi toate1 sunt fără parametri, şi ca atare sistemul va trebui să răspundă dacă există vre-un X şi Y astfel încât aceste predicate să aibă loc, şi ca atare este normal ca răspunsul să nu fie afectat de tipul goal-ului.

Să observăm însă că dacă punem întrebarea compusă

predicat(X, Y), write(X, " şi ", Y, "\n").

atunci sistemul va răspunde diferit în cele două cazuri. În cazul unui goal intern sistemul va răspunde cu

a şi b
Yes

iar în cazul unui goal extern, sistemul va răspunde cu

a şi b
c şi d
e şi f
Yes

Notă. Secvenţa Pseudocod echivalentă cu predicatul toate este următoarea:

cât timp predicat(X, Y) are soluţii execută write(...)

Deci, secvenţele următoare sunt echivalente:

cât timp *condiţie* execută	predicat :-
corp	*condiţie*,
	corp,
	fail.

14.3. Utilizarea predicatului ! (cut)

Turbo Prolog conţine predicatul cut (!) folosit pentru a preveni backtracking-ul. Când se procesează predicatul !, apelul reuşeşte imediat şi se trece la subgoalul următor. O dată ce s-a trecut peste o tăietură, nu este posibilă revenirea la subgoal-urile plasate înaintea ei şi nu este posibil backtracking-ul la alte reguli ce definesc predicatul în execuţie.

Există două utilizări importante ale tăieturii:

1. Când ştim dinainte că anumite posibilităţi nu vor duce la soluţii, este o pierdere de timp să lăsăm sistemul să lucreze. În acest caz, tăietura se numeşte **tăietură verde**.
2. Când logica programului cere o tăietură, pentru prevenirea luării în consideraţie a subgoal-urilor alternative, pentru a evita obţinerea de soluţii eronate. În acest caz, tăietura se numeşte **tăietură roşie**.

14.3.1. Prevenirea backtracking-ului la un subgoal anterior

În acest caz tăietura se utilizează astfel: r1 :- a, b, !, c.

Aceasta este o modalitate de a spune că suntem multumiţi cu primele soluţii descoperite cu subgoal-urile a şi b. Deşi Turbo Prolog ar putea găsi mai multe soluţii prin apelul la c, nu este autorizat să revină la a şi b. De-asemenea, nu este autorizat să revina la altă clauză care defineşte predicatul r1.

14.3.2. Prevenirea backtracking-ului la următoarea clauză

Tăietura poate fi utilizată pentru a-i spune sistemului Turbo Prolog că a ales corect clauza pentru un predicat particular. De exemplu, fie codul următor:

```
r(1) :- !, a, b, c.
r(2) :- !, d.
r(3) :- !, e.
r(_) :- write("Aici intră resul apelurilor").
```

Folosirea tăieturii face predicatul r determinist. Aici, Turbo Prolog apelează predicatul r cu un argument întreg. Să presupunem că apelul este r(1). Turbo Prolog caută o potrivire a apelului. O găseşte la prima clauză. Faptul că imediat după intrarea în clauză urmează o tăietură, împiedică Turbo Prolog să mai caute şi alte potriviri ale apelului r(1) cu alte clauze.

Observaţie. Acest tip de structură este echivalent cu o instrucţiune de tip case unde condiţia de test a fost inclusă în capul clauzei. La fel de bine s-ar fi putut spune şi

```
r(X) :- X = 1, !, a, b, c.
r(X) :- X = 2, !, d.
r(X) :- X = 3, !, e.
r(_) :- write("Aici intra resul apelurilor").
```

Notă. Deci, următoarele secvenţe sunt echivalente:

```
case X of
    v1: corp1;                  predicat(X) :- X = v1, !, corp1.
    v2: corp2;                  predicat(X) :- X = v2, !, corp2.
    ...                         ...
    else corp_else.             predicat(X) :- corp_else.
```

De-asemenea, următoarele secvenţe sunt echivalente:

```
if cond1 then                   predicat(...) :-
    corp1                           cond1, !, corp1.
else if cond2 then              predicat(...) :-
    corp2                           cond2, !, corp2.
...                             ...
else                            predicat(...) :-
    corp_else.                      corp_else.
```

Ca exemplu, să scriem un predicat pentru calculul numărului X la puterea N. Algoritmul este cel tradiţional:

1. orice număr ridicat la puterea 0 este 1;

2. X la puterea N este X la puterea N-1 înmulţit cu X.

Sursa Prolog este următoarea:

```
predicates
        power(real, integer, real)
clauses
        power(_, 0, 1) :- !
        power(X, N, R) :-
                N1 = N – 1,
                power(X, N1, R1),
                R = R1 * X.
```

Să remarcăm aici necesitatea utilizării predicatului cut. Fără acesta, predicatul power va face backtracking şi dincolo de zero, la valori negative ale lui N. Dacă dorim să evităm această situaţie şi în cazul în care valoarea lui N a fost negativă de la bun început, atunci între cele două clauze de mai sus trebuie inserată clauza:

```
power(X, N, R) :-
        N < 0,
        !,
        N1 = -N,
        X1 = 1 / X,
        power(X1, N1, R).
```

Să mai remarcăm prezenţa celor două predicate = din clauza principală (cea cu calculul efectiv). La intrarea în subclauza N1 = N – 1, N este variabilă legată, iar N1 este variabilă liberă. Ca atare rolul subclauzei este de a lega variabila liberă N1 la valoarea expresiei din partea dreaptă. La ieşirea din subclauză, N1 va fi legată la valoarea lui N minus 1. Orice altă formă a subclauzei, sau orice alt loc în care ar fi plasată va produce o eroare. In prima situaţie, forma N = N1 + 1 va genera eşec, deoarece N este variabilă legată şi ca atare se va verifica dacă valorile expresiilor din cele două părţi ale semnului egal sunt egale. In a doua situaţie, dacă subclauza N1 = N – 1 se va plasa după subclauza cu apelul lui power, atunci la intrarea în power N1 va fi variabilă liberă, ca şi R1 de-altfel, şi la intrarea (în cadrul apelului recursiv) în noua clauză N1 = N – 1, cele două variabile vor fi ambele libere, deci iarăşi eşec. O discuţie asemănătoare se poate face în ceea ce priveşte cealaltă clauză egal, R = R1 * X.

14.4. Predicatul not

Sintaxa predicatului not este următoarea:

not(subgoal(Arg1, ..., ArgN))

Predicatul **not** reuşeşte atunci când subgoal-ul care este argument al lui not nu poate fi dovedit adevărat. Dacă subgoal-ul este apelat cu variabile libere, Turbo Prolog va emite mesaj de eroare: Variabilele libere nu sunt autorizate în not. Să remarcăm că not este singurul predicat Prolog care are ca argument un predicat Prolog.

De exemplu, fie secvenţa:

```
place(ilie, Cineva) :-
```

```
place(mariana, Cineva),
not(urăste(ilie, Cineva)).
```

Acest enunţ este echivalent cu: lui Ilie îi place de cineva dacă şi Marianei îi place, şi daca Ilie nu îl urăşte. De observat că, la intrare, Cineva este variabilă liberă, dar la ieşirea din primul subgoal, Cineva este variabilă legată. Astfel, la intrarea în not, Cineva este legat, şi totul va funcţiona bine.

Dacă se inversează ordinea clauzelor, rezultatul nu va mai fi corect, deoarece la intrarea în not Cineva va fi variabilă liberă. Folosirea variabilei anonime (_) în not în locul lui Cineva tot nu va rezolva problema.

14.5. Liste şi recursivitate

14.5.1. Ce este o listă?

În Prolog, o listă este un obiect care conţine un număr arbitrar de alte obiecte. Listele corespund vectorilor din alte limbaje, dar, spre deosebire de aceştia, lungimea lor nu trebuie declarată. Listele se construiesc folosind parantezele drepte. Elementele acestora sunt separate de virgulă.

Iată câteva exemple:

```
[1, 2, 3]
[dog, cat, canary]
[“valerie ann”, “jennifer caitlin”, “benjamin thomas”]
```

14.5.2. Declararea listelor

Pentru a declara domeniul pentru o listă de întregi se foloseşte o declaraţie de domeniu de tipul următor

```
domains
    integerlist = integer*
```

Notă. cuvântul ‘list’ din numele ‘integerlist’ nu are nici o semnificaţie pentru lista în sine. Domeniul s-ar fi putut numi absolut oricum, ceea ce contează fiind caracterul asterisc de după numele tipului de bază.

Toate elementele dintr-o listă trebuie să aparţină unui acelaşi domeniu, şi pentru fiecare tip de elemente trebuie să fie o declaraţie de forma

```
domains
    elementlist = element*
    element = . . .
```

Să remarcăm că Turbo Prolog nu permite definirea în mod direct a listelor eterogene. Vom vedea într-un capitol viitor cum poate fi rezolvată parţial această restricţie.

14.5.3. Capul şi coada unei liste (head & tail)

O listă este un obiect realmente recursiv. Aceasta constă din două părţi: **capul**, care este primul element al listei şi **coada**, care este restul listei. Ca şi în Lisp, capul listei este element, iar coada listei este listă.

Iată câteva exemple:

Capul listei [a, b, c] este a
Coada listei [a, b, c] este [b, c]

Capul listei [c] este c
Coada listei [c] este []

Lista vida [] nu poate fi împărţită în cap şi coada.

14.5.4. Procesarea listelor

Prolog oferă o modalitate de a face explicite capul şi coada unei liste. În loc să separăm elementele unei liste cu virgule, vom separa capul de coadă cu caracterul '|'.

De exemplu, următoarele liste sunt echivalente:

[a, b, c] [a | [b, c]] [a | [b | [c]]] [a | [b | [c | []]]]

De asemenea, înaintea caracterului '|' pot fi scrise mai multe elemente, nu doar primul. De exemplu, lista [a, b, c] de mai sus este echivalentă cu

[a | [b, c]] [a, b | [c]] [a, b, c | []]

14.5.5. Utilizarea listelor

Deoarece o listă este o structură de date recursivă, pentru procesarea ei este nevoie de algoritmi recursivi. Modul de bază de procesare a listei este acela de a lucra cu ea, executând anumite operaţii cu fiecare element al ei, până când s-a atins sfârşitul.

Un algoritm de acest tip are nevoie în general de două clauze. Una dintre ele spune ce să se facă cu o lista ordonată (care se poate descompune în cap şi coadă). Cealaltă spune ce să se facă cu o listă vidă.

14.5.6. Exemple de programe Prolog de tratare a listelor

(1) De exemplu, pentru scrierea listelor am avea nevoie de predicatul următor:

```
domains
        list = integer*
predicates
        writelist(list)
clauses
        writelist([]).          /*pentru a scrie o lista vidă, nu făa nimic */
```

```
        writelist([H|T]) :-     /* pentru a scrie o listă, */
                write(H),       /* scrie capul listei */
                nl,
                writelist(T).   /* apoi scrie coada listei */
goal
        writelist([1, 2, 3]).
```

(2) Care este lungimea unei liste? Iată o definiţie logică:

1. Lungimea listei [] este 0.
2. Lungimea oricărei alte liste este 1 plus lungimea cozii.

Iată sursa în Prolog:

```
domains
        list = integer*
predicates
        length(list, integer)
clauses
        length([], 0).
        length([_|T], L) :-
                length(T, L1),
                L = L1 + 1.
```

(3) Următorul program ia o listă de întregi şi adună 1 la fiecare element al ei:

```
domains
        list = integer*
predicates
        add1(list, list)
clauses
        add1([], []).
        add1([H|T], [H1|T1]) :-
                H1 = H + 1,
                add1(T, T1).
```

(4) Următorul program copiază dintr-o listă în altă listă doar elementele nenegative.

```
domains
        list = integer*
predicates
        copneneg(list, list)
clauses
        copneneg([], []).
        copneneg([H|T], TNou) :-
                H < 0,
                !,
                copneneg(T, TNou).
        copneneg([H|T], [H|TNou]) :-
                copneneg(T, TNou).
```

Să remarcăm aici rolul tăieturii: pentru cazurile în care capul H al primei liste este număr negativ, sistemul nu va face backtracking la ultima clauză. Fără tăietură, se va intra în ultima clauză indiferent de valorile lui H. Ca atare, se poate renunţa la tăietură dacă se poate introduce în ultima clauză condiţia H >= 0. Să observăm că pentru ca un astfel de program să funcţioneze corect fără tăieturi este necesar ca toate condiţiile puse să fie exclusive. Dacă condiţiile nu sunt exclusive va exista posibilitatea ca într-o anumită situaţie să se intre în două clauze diferite, ori nu aceasta este intenţia.

De exemplu, întrebarea

```
copneneg([2, -45, 3, 468], X).
```

produce soluţia X = [2, 3, 468].

(5) Următorul predicat copiază o listă dublând fiecare element.

```
domains
        list = integer*
predicates
        dublează(list, list)
clauses
        dubleaza([], []).
        dubleaza([H|T], [H, H| DubluT]) :-
                dubleaza(T, DubluT).
```

(6) Pentru a descrie apartenenţa la o listă vom contrui predicatul member(element, list) care va investiga dacă un anume element este membru al listei. Algoritmul de implementat ar fi (din punct de vedere declarativ) următorul:

1. E este membru al listei L dacă este capul ei.
2. Altfel, E este membru al listei L dacă este membru al cozii lui L.

Din punct de vedere procedural,

1. Pentru a gasi un membru al listei L, găseşte-i capul;
2. Altfel, găseşte un membru al cozii listei L.

Programul Prolog este:

```
domains
        element = integer
        list = element*
predicates
        member(element, list)
clauses
        member(E, [E|_]).
        member(E, [_|T]) :-
                member(E, T).
```

Observaţie. Acest program va putea răspunde atât la întrebări de confirmare a apartenenţei unui element la o listă (de exemplu, member(2, [1,2,3]), cu răspuns afirmativ), cât şi la întrebări de identificare a membrilor unei liste (de exemplu, member(X, [1,2,3]), cu trei soluţii).

(7) Pentru combinarea listelor L1 si L2 pentru a forma lista L3 vom utiliza un algoritm recursiv de genul:

1. Dacă L1 = [] atunci L3 = L2.
2. Altfel, capul lui L3 este capul lui L1 şi coada lui L3 se obţine prin combinarea cozii lui L1 cu lista L2.

Să explicăm puţin acest algoritm. Fie listele L1 = [a_1, ..., a_n] şi L2 = [b_1, ..., b_m]. Atunci lista L3 va trebui să fie L3 = [a_1, ..., a_n, b_1, ..., b_m] sau, dacă o separăm în cap şi coadă, L3 = [a_1 | [a_2, ..., a_n, b_1, ..., b_m]]. De aici rezultă că:

1. capul listei L3 este capul listei L1;
2. coada listei L3 se obţine prin concatenarea cozii listei L1 cu lista L2.

Mai departe, deoarece recursivitatea constă din reducerea complexităţii problemei prin scurtarea primei liste, rezultă că ieşirea din recursivitate va avea loc odată cu epuizarea listei L1, deci când L1 este []. De remarcat că condiţia inversă, anume dacă L2 este [] atunci L3 este L1, este inutilă.

Programul Prolog este următorul:

```
domains
        list = integer*
predicates
        append(list, list, list)
clauses
        append([], L, L).
        append([X|L1], L2, [X|L3]) :-
                append(L1, L2, L3).
```

Se observă că şi acest program funcţionează cu mai multe modele de flux. De exemplu, încercaţi întrebările

```
append([1, 2], [3], L), append(L, L, LL).       /* model de flux (i,i,o) */
append(L1, L2, [1, 2, 3]).                      /* model de flux (o,o,i) */
append(L, [3, 4], [1, 2, 3, 4]).          /* model de flux (o,i,i) sau (i,o,i) */
```

In al doilea caz Prolog va afişa toate soluţiile ecuaţiei L1 concatenat cu L2 egal cu [1, 2, 3]. Acelaşi lucru se va întâmpla cu a treia întrebare.

(8) Pentru determinarea permutărilor unei liste [E|L], care are capul E şi coada L, vom proceda în felul următor:

1. determină o permutare L1 a listei L;
2. plasează elementul E pe toate poziţiile listei L1 şi produce în acest fel lista X care va fi o permutare a listei iniţiale [E|L].

Programul Prolog este următorul:

```
domains
        elem = integer
        lista = elem*
predicates
        elimin(elem, lista, lista)
        perm (lista, lista)
```

```
clauses
        elimin(E, L, [E|L]).
        elimin(E, [A|L], [A|X]) :-
                elimin(E, L, X).
        perm([], []).
        perm([E|L], X) :-
                perm(L, L1),
                elimin(E, L1, X).
```

Să observăm că predicatul elimin funcţionează cu mai multe modele de flux. Astfel, următoarele întrebări sunt valide:

- elimin(E, L, [1, 2, 3]), cu modelul de flux (o, o, i) şi soluţiile
  E=1, L = [2, 3]
  E=2, L = [1, 3]
  E=3, L = [1, 2]
- elimin(1, [2, 3], L), cu modelul de flux (i, i, o) şi soluţiile
  L = [1, 2, 3]
  L = [2, 1, 3]
  L = [2, 3, 1]
- elimin(1, L, [1, 2, 3]), cu modelul de flux (i, o, i) şi soluţia
  L = [2, 3]
- elimin(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) şi soluţia
  E = 2

În cazul problemei noastre predicatul **elimin** va fi folosit ca în exemplul al doilea. Ca urmare a apelului perm([1, 2, 3]) se vor genera şase soluţii, şi anume cele şase permutări ale listei [1, 2, 3]. Încercaţi o execuţie pas cu pas, cu creionul în mână, pentru a vedea modul în care Prolog foloseşte backtracking-ul pentru a genera cele şase soluţii.

Un alt mod de a descrie generarea permutărilor (dar pe baza aceleiaşi idei) ar fi următorul:

```
domains
        elem = integer
        lista = elem*
predicates
        inserez(elem, lista, lista)
        perm (lista, lista)
clauses
        inserez(E, L, [E|L]).
        inserez(E, [A|L], [A|X]) :-
                inserez(E, L, X).
        perm([], []).
        perm([E|L], X) :-
                perm(L, L1),
                inserez(E, L1, X).
```

Să observăm că predicatul **inserez** funcţionează cu mai multe modele de flux. Astfel, următoarele întrebări sunt valide:

- inserez(E, L, [1, 2, 3]), cu modelul de flux (o, o, i) şi soluţiile
  E=1, L = [2, 3]
  E=2, L = [1, 3]
  E=3, L = [1, 2]
- inserez(1, [2, 3], L), cu modelul de flux (i, i, o) şi soluţiile
  L = [1, 2, 3]
  L = [2, 1, 3]
  L = [2, 3, 1]
- inserez(1, L, [1, 2, 3]), cu modelul de flux (i, o, i) şi soluţia
  L = [2, 3]
- inserez(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) şi soluţia
  E = 2

În cazul problemei noastre predicatul inserez va fi folosit ca în exemplul al doilea.

14.6. Găsirea tuturor soluţiilor în acelaşi timp

Turbo Prolog oferă o modalitate de a găsi toate soluţiile unui predicat în acelaşi timp: predicatul **findall**, care colectează într-o listă toate soluţiile găsite. Acesta are următoarele argumente:

- primul argument specifică argumentul din predicatul considerat care trebuie colectat în listă;
- al doilea argument specifică predicatul de rezolvat;
- al treilea argument specifică lista în care se vor colecta soluţiile.

Pentru a ne lămuri, vom vedea un exemplu. Programul care urmează determină media vârstelor unor persoane date.

```
domains
        name, address = string
        age = integer
        list = age*
predicates
        person(name, address, age)
        sumlist(list, age, integer)

goal
        findall(Age, person(_, _, Age), L),     /* construieşte lista vârstelor */
        sumlist(L, Sum, N),                     /* calculează suma şi lungimea */
        Ave = Sum / N,                          /* calculează media */
        write("Average = ", Ave),               /* scrie rezultatul */
        nl.
clauses
        sumlist([], 0, 0).
        sumlist([H|T], S, N) :-
                sumlist(T, S1, N1),
                S = H + S1,
                N = 1 + N1.
        person("Sherlock Holmes", "22B baker Street", 42).  /* aici sunt definite */
        person("Pete Spiers", "Apt. 22, 21st Street", 36).  /* persoanele */
        person("Mary Darrow", "Suite 2, Omega Home", 51).
```

Observaţie. Să facem diferenţa între obţinerea tuturor soluţiilor unei anumite probleme şi obţinerea listei tuturor soluţiilor unei anumite probleme. În primul caz se va construi un predicat nedeterminist, iar în al doilea caz fie se va regândi problema, fie se va folosi predicatul findall pentru colectarea tuturor soluţiilor predicatului nedeterminist contruit în cazul anterior.

14.7. Alte exemple de programe Prolog

1. Să se scrie un predicat Prolog care să determine suma primelor **N** numere naturale.

Goal: suma(3,S)
va produce S = 6

Formula recursivă de rezolvare este următoarea

$$suma(N) = \begin{cases} 0 & daca \quad N = 0 \\ N + suma(N-1) & altfel \end{cases}$$

```
predicates
        suma(integer, integer)
clauses
        suma(0, 0) :- !.
        suma(N, S) :-
                N1 = N - 1,
                suma(N1, S1),
                S = S1 + N.
```

2. Să se scrie un predicat Prolog care să determine suma numerelor naturale pare din intervalul [a, b].

Goal: suma(3, 1, S)
va produce S = 0

Goal: suma(1, 5, S)
va produce S = 6

Formula recursivă de rezolvare:

$$suma(A,B) = \begin{cases} 0 & daca \ A > B \\ A + suma(A+1,B) & daca \quad A \ e \ par \\ suma(A+1,B) & altfel \end{cases}$$

```
predicates
        suma(integer,i nteger, integer)
clauses
        suma(A, B, 0) :- A>B, !.
        suma(A, B, S) :-
                A mod 2 = 0,
                !,
                A1 = A + 1,
```

```
        suma(A1, B, S1),
        S = S1 + A.
suma(A, B, S) :-
        A1 = A + 1,
        suma(A1, B, S1),
        S = S1 + A.
```

3. Să se scrie un predicat Prolog care să adauge un element la sfârşitul unei liste.

Goal: adaug(3, [1, 2], L)
va produce L = [1, 2, 3]

Formula recursivă:

$$adaug(E, L_1 L_2 \dots L_n) = \begin{cases} (E) & daca \quad L\ e\ lista\ vida \\ L_1 \cup adaug(E, L_2 \dots L_n) & altfel \end{cases}$$

```
domains
        elem = integer
        lista = elem*
predicates
        adaug(elem, lista, lista)
clauses
        adaug(E, [], [E]).
        adaug(E, [A|L], [A|X]) :-
                adaug(E, L, X).
```

4. Să se scrie un predicat Prolog care să producă inversa unei liste

Goal: invers([1, 2], L)
va produce L = [2, 1]

Observaţie: Vom folosi predicatul **adaug** din exemplul anterior (care adauga un element la sfârşitul unei liste).

Formula recursivă de rezolvare:

$$invers(L_1 L_2 \dots L_n) = \begin{cases} \emptyset & daca \quad L\ e\ lista\ vida \\ adaug(L_1, invers(L_2 \dots L_n)) & altfel \end{cases}$$

```
domains
        elem = integer
        lista = elem*
predicates
        invers(lista, lista)
clauses
        invers([], []).
        invers([A|L], X) :-
                invers(L, Y),
```

```
adaug(A, Y, X).
```

Pentru restul problemelor, lăsăm cititorul să descrie formulele recursive de rezolvare.

5. Să se scrie un predicat Prolog care produce suma numerelor pare dintr-o listă cu elemente liste de numere întregi.

Goal: suma([[1, 2], [3,4]], S)
va produce S = 6

Observaţie: Vom folosi un predicat auxiliar **sumal** care determină suma numerelor pare dintr-o listă de numere întregi.

```
domains
        elem = integer
        lista = elem*
        listad = lista*                         % tipul listei duble
predicates
        suma(listad, integer)
        sumal(listad, integer)
clauses
        sumal([], 0).
        sumal([A|L], S) :-
                A mod 2 = 0,
                !,
                sumal(L, S1),
                S = S1 + A.
        sumal([_|L], S) :-
                sumal(L, S).
        suma([], 0).
        suma([L|Ld], S) :-
                sumal(L, S1),
                suma(Ld, S2),
                S = S1 + S2.
```

6. Să se scrie un predicat Prolog care produce reuniunea a două mulţimi reprezentate sub formă de liste

Goal: reun([[1, 3, 2], [3,4]], L)
va produce L = [1, 3,2, 4]

Observaţie: Vom folosi predicatul auxiliar **member** descris în exemplul (6) din secţiunea 13.5.6.

```
domains
        elem = integer
        lista = elem*
predicates
        member(elem, lista)
        reun(lista, lista, lista)
clauses
```

```
reun([], L, L).
reun([A|L1], L2, [A|L3]) :-
        not(member(A, L2)),
        !,
        reun(L1, L2, L3).
reun([_|L1], L2, L3) :-
        reun(L1, L2, L3).
```

7. Să se scrie un predicat Prolog care să fie adevărat dacă elementele unei liste numerice sunt ordonate crescător.

Goal: crescator([1, 3, 2])
va produce No
Goal: crescator([1, 2, 3])
va produce Yes

```
domains
        elem = integer
        lista = elem*
predicates
        crescator(lista)
clauses
        crescator([_]).
        crescator([A|[B|L]]) :-
                A <= B,
                crescator([B|L]).
```

8. Dându-se o listă numerică, se cere un predicat Prolog care determină lista perechilor de elemente strict crescătoare din listă.

Goal: perechi([2, 1, 3, 4], L)
va produce L = [[2, 3], [2, 4], [1, 3], [1, 4], [3, 4]]

Goal: perechi([5, 4, 2], L)
va produce No solution

Observaţie: Vom folosi următoarele predicate:

- predicatul nedeterminist **pereche**(element, lista) (model de flux (i, o)), care va produce perechi în ordine crescătoare între elementul dat şi elemente ale listei argument
  Goal: pereche(2, [1, 3, 4], L)
  va produce L = [2, 3]
  L = [2, 4]

- predicatul nedeterminist **per**(lista, lista) (model de flux (i, o)), care va produce perechi în ordine crescătoare între elementele listei argument
  Goal: per([2, 1, 4], L)
  va produce L = [2, 4]

L = [1, 4]

- predicatul principal **perechi**(lista, listad) (model de flux (i, o)), care va colecta toate soluţiile predicatului nedeterminist **per**.

```
domains
        elem = integer
        lista = elem*
        listad = lista*
predicates
        pereche(elem, lista)
        per(lista, lista)
        perechi(lista, listad)

clauses
        pereche(A, [B|_], [A, B]) :-
                A < B.
        pereche(A, [_|L], X) :-
                pereche(A, L, X).

        per([A|L], X) :-
                pereche(A, L, X).
        per([_|L], X) :-
                per(L, X).
        perechi(L, Ld) :-
                findall(X, per(L, X), Ld).
```

9. Să se scrie un predicat Prolog care să fie adevărat dacă o listă numerică reprezintă o mulţime (elementele sale sunt distincte).

Goal: multime([1, 3, 2])
va produce Yes

Goal: multime([1, 2, 1])
va produce No

Observaţie: Vom folosi predicatul auxiliar **member** descris în exemplul (6) din secţiunea 13.5.6.

```
domains
        elem = integer
        lista = elem*
predicates
        multime(lista)
clauses
        multime([]).
        multime([A|L]) :-
                not(member(A,L)),
                multime(L).
```

10. Să se scrie un predicat Prolog care să calculeze suma primelor **N** elemente dintr-o listă cu elemente numerice (dacă **N** depăşeşte lungimea listei, atunci rezultatul va fi suma elementelor listei).

Goal: suma([1, 3, 2], 2, S)
va produce S = 4

Goal: suma([1, 3, 2], 4, S)
va produce S = 6

```
domains
        elem = integer
        lista = elem*
predicates
        suma(lista, integer, integer)
clauses
        suma(_, 0, 0) :- !.
        suma([], _, 0).
        suma([A|L], N, S) :-
                N1 = N - 1,
                suma(L, N1, S1),
                S = A + S1.
```

11. Să se scrie un predicat Prolog care având ca argument o listă numerică, va produce lista iniţială cu elementele din **N** în **N** scrise de două ori.

Goal: dublare([1, 2, 3, 4, 5], 2, L)
va produce L = [1, 2, 2, 3, 4, 4, 5]

Observaţie: Vom folosi un predicat auxiliar **inserare(L, I, N, L1)** care furnizează lista L1 obţinută dublând elementele listei **L** din **N** în **N**, începând cu elementul de pe poziţia **I**.

```
domains
        elem = integer
        lista = elem*
predicates
        dublare(lista, integer, lista)
        inserare(lista, integer, integer, lista)
clauses
        dublare(L, N, L1) :- inserare(L, N, N, L1).
        inserare([], _, _, []).
        inserare([A|L], 1, N, [A|[A|L1]]) :-
                !,
                inserare(L, N, N, L1).
        inserare([A|L], M, N, [A|L1]) :-
                M1 = M - 1,
                inserare(L, M1, N, L1).
```

12. Să se scrie un predicat Prolog care având ca argument o listă numerică, produce ca rezultat lista iniţială cu elementele din **N** în **N** eliminate.

Goal: eliminare([1, 2, 3, 4, 5], 2, L)
va produce L = [1, 3, 5]

Observaţie: Vom folosi un predicat auxiliar **stergere(L, I, N, L1)** care furnizează lista L1 obţinută eliminând elementele listei **L** din **N** în **N**, începând cu elementul de pe poziţia **I**.

```
domains
        elem = integer
        lista = elem*
predicates
        eliminare(lista, integer, lista)
        stergere(lista, integer, integer, lista)
clauses
        eliminare(L, N, L1) :- stergere(L, N, N, L1).
        stergere([], _, _, []).
        stergere([_|L], 1, N, L1) :-
                !,
                stergere(L, N, N, L1).
        stergere([A|L], M, N, [A|L1]) :-
                M1 = M - 1,
                stergere(L, M1, N, L1).
```

13. Să se scrie un predicat Prolog care având ca argument o listă formată din simboluri, produce ca rezultat al **N**-lea element din listă.

Goal: select([a, b, c, d], 2, E)
va produce E = b

Goal: select([a, b, c, d], 5, E)
va produce **nu exista element**

Observaţie: Vom folosi următoarele predicate auxiliare:

- predicatul **length**(lista) care determină lungimea unei liste
- predicatul principal **run**(lista) care va rezolva cerinţa problemei, citind în prealabil valoarea **N** (poziţia de inserare)

```
domains
        elem = symbol
        lista = elem*
predicates
        run(lista)
        select(lista, integer, elem)
        tipar(lista, integer)
        length(lista, integer)
clauses
        length([],0):-!.
        length([_|L],M) :-
                length(L,M1),
                M = M1 + 1.
        run(L) :-
```

```
        write("N="),
        readint(N),
        tipar(L,N).
tipar(L,N) :-
        length(L,M),
        N>M,
        !,
        write("nu exista element"),
        nl.
tipar(L,N) :-
        select(L,N,S),
        write("ELEMENT=",S),
        nl.
select([A|L],1,A) :- !.
select([_|L],N,S) :-
        N1 = N - 1,
        select(L,N1,S).
```

La o întrebare de forma

```
Goal:   run ([a,b,c,d])
        N=2
```

se va afisa

```
        ELEMENT=b
```

14. Să se scrie un predicat Prolog care având ca argument o listă numerică produce ca rezultat lista sortată (folosind sortarea prin inserare).

Goal: sortare([3, 1, 4, 2], L)
va produce L = [1, 2, 3, 4]

Observaţie: Ideea sortării prin inserare este următoarea: dacă lista e vidă, rezultatul este lista vidă; dacă nu, se inserează primul element al listei în lista obţinută în urma sortării restului listei. De aceea vom folosi predicatul auxiliar **inserează**(element, lista) care determină lista obţinută prin inserarea elementului în lista (sortată) argument.

```
domains
        elem = integer
        lista = elem*
predicates
        sortare(lista, lista)
        insereaza(elem, lista, lista)
clauses
        sortare([], []).
        sortare([A|L], L1) :-
                sortare(L, L2),
                insereaza(A, L2, L1).
        insereaza(A, [], [A]).
        insereaza(A, [B|L], [A|[B|L]]) :-
                A < B,
                !.
      insereaza(A, [B|L], [B|L1]) :-
```

insereaza(A, L, L1).