3

Listele şi tratarea lor

3.1. Reprezentarea şi unificarea listelor

În Prolog lista este reprezentată printr-un termen structurat standard, predefinit, a cărui functor este caracterul "." şi are două componente: primul element al listei şi restul listei. Lista vidă este reprezentată prin atomul special *[]*. De exemplu, o listă cu un singur element *a* se reprezintă în Prolog *.(a,[]),* iar o listă cu trei elemente *a,b,c* se reprezintă *.(a,.(b,.(c,[]))).*

Ca şi orice termen structurat, o listă poate fi reprezentată în formă de arbore. De exemplu, listele *.(a,[])* şi *.(a,.(b,.(c,[])))* pot fi reprezentate ca în Fig.3.1 şi Fig. 3.2, corespunzător.

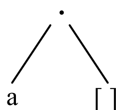

Fig. 3.1. O listă cu un singur element *a*

Menţionăm, că întrucât ordinea elementelor în listă este relevantă, lista *.(a,.(b,.(c,[])))* este diferită de lista *.(c,.(b,.(a,[]))).* Deoarece functorul "." este definit şi ca un operator infix, el poate fi utilizat în locul formei prefix. De exemplu, *(a.[])* şi *(a.(b.(c.[]))).* Parantezele pot fi omise, fiindcă "." este definit ca un operator asociativ în dreapta. Astfel, listele de mai sus au forma *a.[]* şi *a.b.c.[].*

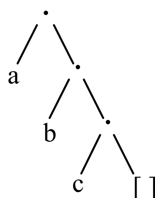

Fig. 3.2. O listă ce conţine atomii *a,b,c*

Dat fiind faptul, că lista e o structură intens utilizată, Prologul oferă şi o notaţie foarte comodă. Elementele listei se iau în paranteze pătrate şi se separă prin virgulă. Aşadar, lista ce conţine un singur atom *a* se scrie *[a]*, iar lista compusă din atomii *a, b* şi *c* se scrie *[a,b,c]*. Elemente ale listei pot fi atomi, numere, liste şi, în general, orice structuri. Listele, bineînţeles, pot conţine şi variabile. Următoarele liste sunt, de exemplu, corecte:

[a, sosit, [ziua, de, mâine]]
[a, X, b, [Y, c]]
[cartea(cugetari,autor(blaise,pascal)),
cartea(moby_dick,autor(herman,melville))]

Presupunem, că e necesar de specificat o listă cu primele două elemente cunoscute şi un număr nedefinit de alte elemente. O astfel de listă poate fi reprezentată, utilizând simbolul lexical "|". El separă elementele ce constituie capul listei de o variabilă ce reprezintă restul sau coada listei. Un exemplu poate fi lista *[unu, doi|X]*. Deci, "|" divizează lista în două părţi: capul listei – o enumerare de elemente înaintea lui "|" (elementele *unu, doi*) şi restul listei (variabila *X*). Capul listei este întotdeauna reprezentat explicit, iar coada poate fi reprezentată implicit printr-o variabilă în dreapta simbolului "|".

Lista	Cap	Coada
[a,b,c]	a	[b,c]
[a,b]	a	[b]
[a]	a	[ ]
[ ]		
[[a,b],c]	[a,b]	[c]
[a,[b,c]]	a	[[b,c]]
[a,[b,c],d]	a	[[b,c],d]
[[1,2,3],[2,3,4],[ ]]	[1,2,3]	[[2,3,4],[ ]]

Fig. 3.3. Exemple de separare a capului de coadă

Astfel, bara verticală "|" serveşte pentru separarea capului listei de coadă. Virgula se foloseşte pentru separarea unui element de elementul următor. În notaţia *[unu|X]* capul, *unu,* este un element (aici primul din listă), pe când coada, *X*, este restul listei, adică o altă listă mai scurtă cu un element. De exemplu, se poate scrie *[unu|[doi,trei]]*. Dar nu se poate scrie *[unu|doi,trei]*, fiindcă ceea ce urmează după bară nu este o listă. Înaintea barei suntem obligaţi să enumerăm explicit toate elementele capului. De exemplu, în Fig. 3.3 sunt evidenţiate notaţia listei, capul şi coada listei.

Listă	Semnificaţie
[X|L]	Toate listele nevide
[X, Y, Z |[ ]]	Toate listele cu trei elemente
[X,Y,Z,Coada]	Toate listele cu patru elemente
[X, X, Coada]	Toate listele cu trei elemente în care primul şi al doilea elemente reprezintă acelaşi termen
[41,X|Y]	Toate listele cu cel puţin două elemente primul din care este 41
[a,_,_,b]	Toate listele cu patru elemente în care primul şi al patrulea sunt atomii *a* şi *b*, iar celelalte pot fi orice termen

Fig. 3.4. Liste cu variabile şi semnificaţiile lor

În afară de aceasta, o listă poate fi reprezentată în mai multe feluri. De exemplu, notaţiile ce urmează sunt echivalente:

[1,2,3], [1, 2,3| [ ]], [1,2|[3|[ ]]], [1|[2,3|[ ]]], [1|[2|[3|[ ]]]]

iar în Fig. 3.4 este ilustrată semnificaţia listelor ce conţin variabile.

© Vitalie Cotelea

<table>
<tr><th>Lista 1</th><th>Lista 2</th><th colspan="3">Instanțieri</th></tr>
<tr><td>[X, Y, Z]</td><td>[a, b, c]</td><td>X=a</td><td>Y=b</td><td>Z=c</td></tr>
<tr><td>[a]</td><td>[X|Y]</td><td>X=a</td><td>Y=[]</td><td></td></tr>
<tr><td>[X|Y]</td><td>[a, b, c]</td><td>X=a</td><td>Y=[b,c]</td><td></td></tr>
<tr><td>[X,Y]</td><td>[a, b, c]</td><td>no</td><td></td><td></td></tr>
<tr><td>[X, Y|Z]</td><td>[a, b, c]</td><td>X=a</td><td>Y=b</td><td>Z=[c]</td></tr>
<tr><td>[X|[Y|Z]]</td><td>[a, b, c]</td><td>X=a</td><td>Y=b</td><td>Z=[c]</td></tr>
<tr><td>[X|Y]</td><td>[[1,2], a,b]</td><td>X=[1,2]</td><td>Y=[a,b]</td><td></td></tr>
<tr><td>[[a, Y]|Z]</td><td>[[X, b], [c, d]]</td><td>X=a</td><td>Y=b</td><td>Z=[[c, d]]</td></tr>
<tr><td>[X|Y]</td><td>[]</td><td>no</td><td></td><td></td></tr>
<tr><td>[X|Y]</td><td>[X|[Y]]</td><td>no</td><td></td><td></td></tr>
<tr><td>[X|Y]</td><td>[[1,2]|X]</td><td>X=Y; X=[1,2]</td><td></td><td></td></tr>
</table>

Fig. 3.5. Exemple de unificare a listelor

Considerăm câteva exemple (Fig. 3.5) pentru a vedea cum două liste (ce conțin variabile) se unifică. Aici se utilizează cunoscutele reguli de unificare a termenilor.

3.2. Tratarea listelor

Unul din motivele utilizării listelor rezidă în capacitatea lor de a exprima situații dinamice. Algoritmii de prelucrare, însă, trebuie să fie expliciți. Această cerință este remediată prin metode inteligente.

Majoritatea predicatelor asupra listei divizează lista în cap și coadă. Asupra capului se aplică o operație, apoi însăși coada se împarte în cap și coadă și operația e aplicată din nou. Procesul continuă până când nu se întâlnește o condiție de terminare. Cea mai evidentă condiție este lista vidă. Astfel, dacă avem lista *[unu,doi,trei]*, ea poate fi redusă în patru pași:

1. *Cap=unu, Coada=[doi,trei]*
2. *Cap=doi, Coada=[trei]*
3. *Cap=trei, Coada=[]*
4. Lista vidă (*[]*)

Prelucrarea acestei liste poate fi reprezentată ca în Fig. 3.6

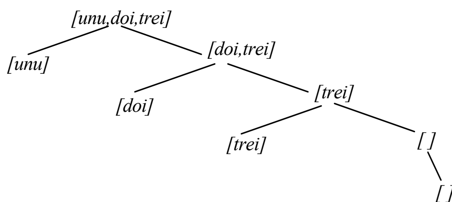

Fig. 3.6. Prelucrarea unei liste

Forma arborelui din Fig. 4.6 ne sugerează, că procedura de prelucrare a unei liste foarte mult seamănă cu programul de construire a unei căi din orașul *A* în orașul *B*. Programul conținea două reguli, dintre care prima era condiția de terminare, iar a doua clauza recursivă:

```
cale(OrasA, OrasB):-
       ruta(OrasA, OrasB).
cale(OrasA, OrasB):-
       ruta(OrasA, OrasC),
       cale(OrasC, OrasB).
```

După același model vom descrie, de exemplu, un predicat de afișare a unei liste. Prima clauză va avea forma

```
afisare_lista([]).
```

A doua clauză este o regulă recursivă, ce are forma

```
afisare_lista([Cap|Coada]):-
       write(Cap),nl,
       afisare_lista(Coada).
```

Punând ambele împreună, obținem

```
afisare_lista([]).
afisare_lista([Cap|Coada]):-
       nl,write(Cap),
       afisare_lista(Coada).
```

Predicatul *afisare_lista/1* are un singur argument (o listă) și el afișează elementele listei. Iar *write/1* e un predicat predefinit ce afișează argumentul său pe un dispozitiv standard. Predicatul *nl/0* este un predicat predefinit de trecere la un rând nou.

Predicatul *afisare_inver/1* e aproape identic cu *afisare_lista/1*, numai că el imprimă elementele listei în ordine inversă. Aceasta se face simplu, schimbând ordinea subscopurilor în regula recursivă.

```
afisare_inver([]).
afisare_inver([Cap|Coada]):-
       afisare_inver(Coada),
       nl, write(Cap).
```

Procesul de mai sus este destul de standard și programatorul începător îl poate urma pentru a scrie definiții recursive ale predicatelor de manipulare a listelor. Menționăm, că sunt și excepții care scapă de principiul general de recursie, dar acestea vor fi considerate în secțiunile următoare.

Criteriile de terminare sunt esențiale în prelucrarea listelor, fiindcă ele reprezintă condițiile fără care sistemul eșuează. Se cunosc trei criterii principale de terminare: terminarea când lista este vidă, când elementul specificat e găsit și când țelul specificat este atins.

3.3. Terminarea prelucrării când lista e vidă

Un astfel de exemplu deja a fost descris de predicatul *afisare_lista/2*. Punctul cheie constă în faptul că procesul ia sfârșit când este atinsă lista vidă - *[]*.

3.3.1. Predicatul *este_lista/1*

Predicatul *este_lista/1* este unul din cele mai simple predicate ce face parte din această familie. Idea constă în aceea, că un argument este o listă, dacă el poate fi redus la lista vidă.

```
este_lista([]).
este_lista([_|Coada]):-
       este_lista(Coada).
```

Să observăm, că nu are loc nici un proces, atât în prima clauză, cât și în a doua. În afară de aceasta, este utilizată variabila anonimă, fiindcă asupra capului nu se aplică *procesul_2/1* și deci, nu e nevoie să cunoaștem capul. Aici sunt interesante următoarele întrebări:

```
?-este_lista([])
yes
```

```
?-este_lista([a,b,c])
yes
```

3.3.2. Calcularea lungimii unei liste

O diferenţă evidentă dintre predicatul precedent şi cel de calculare a lungimii unei liste, *lungime/2,* este că ultimul are două argumente. Pentru cei ce utilizează alte limbaje de programare, care includ proceduri-funcţii, acest predicat poate trezi nedumeriri. Aici predicatul nu întoarce valoarea ca în cazul funcţiilor din limbajele imperative. Valoarea pe care dorim s-o obţinem se include în calitate de argument. Predicatul de mai jos are un singur ţel - calcularea numărului de elemente într-o listă dată. Să observăm, că acest predicat puţin se deosebeşte de modelul de bază.

```
lungime([],Total):-
       Total=0.
lungime([_|Coada],Total):-
       lungime(Coada,Cont),
       Total=Cont+1.
```

Principala diferenţă este că ordinea subscopurilor recursiv şi de proces din regula recursivă sunt schimbate cu locurile.

3.3.3. Concatenarea a două liste

Vom scrie un predicat de concatenare a două liste. Îl vom numi *concatenare/3* şi va avea trei argumente, toate liste. Acest predicat este foarte utilizat. Dacă avem listele *[1,2,3]* şi *[4,5]* putem să obţinem lista *[1,2,3,4,5]* .

Procesul de concatenare a unei liste cu alta va decurge, alipindu-se câte un element de la una la cealaltă. Să considerăm mai întâi condiţia de terminare. E evident, că dacă prima listă este vidă, atunci rezultatul concatenării va fi lista a doua.

```
concatenare([],L2,L3):-
       L2=L3.
```

Programatorii mai experimentaţi lasă o mare parte de lucru pe seama unificării argumentelor, micşorând astfel numărul de subscopuri. În acest exemplu se pot unifica argumentele *L2* şi *L3*, scriind acelaşi nume. Atunci regula de mai sus se transformă într-un fapt.

```
concatenare([],L,L).
```

Până a trece la regula recursivă, să vedem răspunsurile la nişte întrebări referitor la faptul de mai sus.

```
?-concatenare([],[4,5],[4,5])
yes
```

```
?-concatenare([],[4,5],L)
L=[4,5]
```

```
?-concatenare(Var,[4,5],L)
Var=[],L=[4,5]
```

```
?-concatenare([],[],L)
L=[]
```

```
?-concatenare(Var,[],L)
Var=[],L=[]
```

Primele două răspunsuri sunt clare. În cazul trei, primul argument al întrebării este o variabilă neinstanţiată care se unifică cu *[]*. Ultimele două întrebări sunt mai interesante. Din ele se vede că faptul nostru are dreptul la existenţă pentru orice valoare a argumentului doi, fie o listă vidă sau o listă nevidă.

Acum se poate scrie regula recursivă. Ea trebuie să lucreze cu primul argument. O definiţie a regulii recursive, ce se potriveşte modelului de bază, poate fi

```
concatenare([C1|Cd1],L2,[C3|Cd3]):-
       C1=C3,
       concatenare(Cd1,L2,Cd3).
```

Această regulă poate fi citită: O listă este adevărată, dacă este adevărat că se poate unifica capul primei liste cu capul listei trei şi se poate alipi coada primei liste la lista a doua, obţinând coada listei trei.

Ca şi în cazul condiţiei de terminare, se poate prezenta această regulă mai succint, doar renumind variabilele ce exprimă capetele listelor unu şi trei cu acelaşi nume.

```
concatenare([C|Cd1],L2,[C|Cd3]):-
       concatenare(Cd1,L2,Cd3).
```

Grupând aceste două clauze împreună, se obţine definiţia predicatului de concatenare a două liste.

```
concatenare([],L,L).
concatenare([C|Cd1],L2,[C|Cd3]):-
       concatenare(Cd1,L2,Cd3).
```

Să considerăm câteva întrebări referitor la acest predicat.

```
?-concatenare([1,2,3],[4,5],L3)
L3=[1,2,3,4,5]
```

```
?-concatenare([1,2,3],L2,[1,2,3,4,5])
L2=[4,5]
```

```
?-concatenare(L1,[4,5],[1,2,3,4,5])
L1=[1,2,3]
```

Interesantă este întrebarea "Care sunt perechile de liste, concatenarea cărora produc lista *[1,2,3]*?":

```
?-concatenare(L1,L2,[1,2,3])
L1=[], L2=[1,2,3];
L1=[1], L2=[2,3];
L1=[1,2], L2=[3];
L1=[1,2,3], L2=[]
```

Deci, în cazul când un singur argument în predicatul de concatenare este variabilă, există o singură soluţie, iar pentru cazul în care primele două argumente sunt variabile se obţin mai multe soluţii corespunzătoare primelor două liste care prin concatenare formează cea de a treia listă.

3.4. Terminarea prelucrării când elementul specificat e găsit

În calitate de exemplu pot fi considerate predicatele de căutare a unui element într-o listă sau de aplicare a unei operaţii asupra unui element, cum ar fi substituirea cu un alt element.

Diferenţa dintre acest model şi precedentul este că în cazul terminării cu listă vidă se examinează orice element (fie se numără sau fie se alipeşte la o listă). În cazul de faţă, ne interesează doar elementul pe care îl căutăm, celelalte fiind irelevante. Ca şi în cazul precedent avem de afacere doar cu un model.

3.4.1. Calitatea de membru al listei

Considerăm următoarea problemă: există o listă de termeni Prolog şi un termen despre care dorim să ştim dacă este sau nu în această listă. Ce metodă trebuie de utilizat pentru a soluţiona această problemă?

Predicatul ce defineşte această problemă este utilizat în multe aplicaţii, fiindcă el testează apartenenţa unui element la o listă. Ideea de realizare constă în următoarele. Mai întâi de toate,

© Vitalie Cotelea

se verifică dacă termenul căutat nu este însăşi capul listei. Dacă da, atunci el e găsit. În caz contrar, se verifică dacă termenul nu se găseşte în coada listei. Dacă s-a atins sfârşitul listei, atunci termenul nu aparține ei.

Să vedem acum cum se scrie în Prolog predicatul respectiv. Se defineşte predicatul *apartine(Elem, Lista),* care e adevărat, dacă termenul desemnat de *Elem* este membru al listei desemnate de *Lista*. Se vede, mai întâi, cum se verifică dacă *Elem* este capul listei *Lista*. Acest lucru este exprimat de faptul

```
apartine(Elem,[Elem|_]).
```

Într-a doilea rând, se defineşte că *Elem* este membru al unei liste, dacă el se găseşte în restul listei. Regula

```
apartine(Elem,[_|Rest]):-
       apartine(Elem,Rest).
```

exprimă exact acest fapt: *Elem* aparține listei, dacă *Rest* este coada listei şi *Elem* este în *Rest*.

Notăm, că lucrul, pe de o parte, îl face unificarea şi anume extragerea capului listei - în prima clauză şi extragerea cozii listei - într-a doua clauză, pe de altă parte, lucrul îl face variabila anonimă. Aici variabila anonimă se unifică cu partea listei care nu se consideră (coada în prima clauză şi capul în clauza a doua).

Se observă şi schema generală a unui program recursiv. În primul rând, este prezent cazul terminal exprimat de prima clauză. In al doilea rând, cazul general cu un apel recursiv exprimat de clauza secundă. Mai există, de fapt, un caz terminal suplimentar, implicit, atunci când lista în care se caută elementul este vidă. Aici căutarea eşuează, fiindcă argumentul al doilea din capetele ambelor clauze nu se pot nici cum unifica cu lista vidă.

Acum, să formulăm câteva întrebări, conținând acest predicat.

```
?-apartine(c,[a,b,c,d])
yes

?-apartine(e,[a,b,c,d])
no

?-apartine(X,[a,b,c,d])
X=a;
X=b;
X=c;
X=d
```

Pentru cazul în care primul argument este o variabilă, există patru soluții posibile ale scopului. Acest predicat pune în evidență o facilitate deosebit de interesantă a limbajului Prolog - puterea lui generativă. Predicatul *apartine/2* poate fi folosit atât pentru testarea apartenenței unui element la o listă, cât şi pentru generarea pe rând a elementelor unei liste prin resatisfacerea succesivă. În anumite contexte de utilizare această facilitate poate fi folositoare, iar în alte contexte ea poate genera efecte nedorite.

Să menționăm, că unii programatori neexperimentați deseori mai adaugă o clauză la cele două, care de fapt este incorectă:

```
apartine(_,[]).
```

Fiind adăugată această clauză la celelalte două, următoarea întrebare are răspuns afirmativ.

```
?-apartine(unu,[]).
yes
```

Dacă acesta este răspunsul aşteptat, atunci e bine. Dar majoritatea consideră că dacă un obiect nu este în lista examinată, atunci predicatul trebuie să eşueze. Deci, pentru a anunța eşecul nu trebuie de adăugat nimic la program. Prologul face aceasta singur.

De remarcat, că predicatul *apartine/2* este inclus în bibliotecile multor versiuni ale limbajului Prolog.

3.4.2. Eliminarea unui element din listă

În următorul exemplu vom elimina un element dintr-o listă. Acest predicat e similar predicatului *apartine/2*, dar mai are şi al treilea argument pentru a reprezenta lista fără elementul specificat.

```
eliminare_elem(E,[E|Cd],Cd).
eliminare_elem(E,[C|Cd1],[C|Cd2]):-
       eliminare_elem(E,Cd1,Cd2).
```

Să examinăm mai întâi condiția de terminare:

```
eliminare_elem(E,[E|Cd],Cd).
```

Ea este adevărată, dacă elementul specificat, *E*, este capul listei. În acest caz, al treilea argument este coada listei.

Regula recursivă seamănă mult cu cea a predicatului *concatinare/3*.

```
eliminare_elem(E,[C|Cd1],[C|Cd2]):-
       eliminare_elem(E,Cd1,Cd2).
```

În această regulă *C* al primei liste se unifică cu *C* al listei doi, care, de fapt, este prima listă fără elementul specificat. Subscopul din corpul regulii specifică că *E* este în *Cd1*, iar *Cd2* este lista căutată fără *E*.

Sunt interesante răspunsurile la următoarele întrebări referitor la predicatul *eliminare_elem/3*.

```
?-eliminare_elem(3,[1,2,3,4],Rez)
Rez=[1,2,4]

?-eliminare_elem(1,[1],Rez)
Rez=[]

?-eliminare_elem(4,[1,2,3],Rez)
no

?-eliminare_elem(1,[],Rez)
no

?-eliminare_elem(2,[1,2,3,2],Rez)
Rez=[1,3,2];
Rez=[1,2,3]
```

Acest program se poate utiliza pentru a găsi toate listele ce pot fi create din lista dată, adăugând la ea un element:

```
?-eliminare_elem(2,Lista,[1,3])
Lista=[2,1,3];
Lista=[1,2,3];
Lista=[1,3,2]
```

De asemenea, se pot afişa toate perechile dintr-un element şi o sublistă, formate dintr-o listă dată:

```
?-eliminare_elem(E,[1,2,3],Lista)
E=1, Lista=[2,3];
E=2, Lista=[1,3];
E=3, Lista=[1,2]
```

© Vitalie Cotelea

Este evident că predicatul *eliminare_elem/3* poate avea diverse utilizări. Una din ele este inserarea unui element într-o listă. Astfel, un sinonim pentru *eliminare_elem/3* poate fi definit

```
inserare(E,Lista,Rez):-
        eliminare_elem(E,Rez,Lista).
```

3.5. Terminarea prelucrării când țelul specificat este atins

În acest caz listele se prelucrează până nu se întâlnește un punct specificat după care se purcede la efectuarea unor operații.

3.5.1. Găsirea elementului fiind dată poziția lui în listă

Predicatul *pozitie/3* reprezintă exact modelul de mai sus. Idea constă în obținerea elementului concret când contorul se reduce la 1. Acest predicat găsește elementul, dacă este dată poziția sa în listă.

```
pozitie(1,E,[E|_]).
pozitie(Cnt,E,[_|Coada]):-
        Cnt1=Cnt-1,
        pozitie(Cnt1,E,Coada).
```

Dacă sunt formulate întrebările ce urmează, se obțin răspunsurile respective.

```
?-pozitie(0,E,[1,2,3]).
no

?-pozitie(2,E,[1,2,3]).
E=2

?-pozitie(4,E,[1,2,3]).
no
```

Să reproducem aici predicatul de calculare a numărului de elemente într-o listă, considerat mai sus

```
lungime([],0).
lungime([_|Coada],Total):-
        lungime(Coada,Cont),
        Total=Cont+1.
```

El utilizează o variabilă numită *Cont,* care joacă rolul de contor. În acest predicat, pentru a susține calcularea, se introduce un acumulator reprezentat de un alt argument. Atunci se scrie următoarea condiție nouă de terminare.

```
lungime_ac([],Total,Total).
```

Această condiție ne spune că dacă lista e vidă, atunci celelalte argumente au aceeași valoare.

Condiția recursivă este o rearanjare a versiunii vechi.

```
lungime_ac([_|Coada],Suma,Total):-
        Cont=Suma+1,
        lungime_ac(Coada,Cont,Total).
```

La fiece pas al recursiei regula adaugă câte o unitate la suma acumulată. Să descifrăm cum *Cont* crește și cum variabila *Total* este instanțiată atunci când este atinsă condiția terminală.

Aici trebuie de constatat că nu se pot formula întrebări în care se indică elementul și lista, iar în rezultat să se ceară obținerea poziției elementului în lista dată.

3.5.2. Eliminarea unui element indicat de poziția sa în listă

Acest predicat este adevărat, dacă se obține o listă din lista dată, dar fără elementul sub numărul de ordine *n*. Ca și în predicatul *pozitie/3,* în calitate de argument apare valoarea care indică numărul elementului în listă ce trebuie eliminat.

```
eliminare_n(1,[_|Cd],Cd).
eliminare_n(Cnt,[C|Cd1],[C|Cd2]):-
        Cnt1=Cnt-1,
        eliminare_n(Cnt1,Cd1,Cd2).
```

Notăm, că în clauza recursivă capul primei liste s-a unificat cu capul listei doi. Să formulăm, acum, niște întrebări.

```
?-eliminare_n(0,[1,2,3],L).
no

?-eliminare_n(2,[1,2,3],L).
L=[1,3]

?-eliminare_n(4,[1,2,3],L).
no
```

3.6. Acumulatoarele

În această secțiune se redefinesc unele predicate din secțiunile de mai sus cu scopul de ale face mai eficiente. Pentru aceasta se folosesc acumulatoarele. Acumulatorul este un argument adițional pentru fixarea datelor intermediare.

3.6.1. Calcularea lungimii unei liste

Iată unele răspunsuri la întrebări referite la acest predicat.

```
?-lungime_ac([1,2,3],0,Total)
Total=3

?-lungime_ac([1,2,3],0,4)
no

?-lungime_ac([],0,Total)
Total=0

?-lungime_ac(Lista,0,3)
Lista=[_,_,_]
```

Este incomod de a atribui *0* argumentului doi ori de câte ori se formulează întrebarea. De aceea se mai scrie o regulă adițională care apelează la regula ce conține *0* în calitate de argumentul doi.

```
lungime_ac(Lista,Total):-
        lungime_ac(Lista,0,Total).
```

3.6.2. Indexarea listei

Deja s-a menționat că predicatul *pozitie/3* nu poate afișa poziția în listă a unui element dat. Vom rescrie acest predicat pentru ca el să soluționeze ambele probleme: găsirea poziției elementului în listă dacă el e specificat și găsirea elementului în listă dacă este dată poziția lui.

```
pozitie_ac(Poz,Poz,E,[E|_]).
pozitie_ac(Cnt,Poz,E,[_|Cd]):-
        Cnt1=Cnt+1,
        pozitie_ac(Cnt1,Poz,E,Cd).
```

Asemenea predicatului *lungime_ac/3,* predicatul *pozitie_ac/4* utilizează un contor (numit *Cnt1* şi *Cnt*) şi un total (numit *Poz*) pentru a păstra calea spre poziția elementului. Condiția recursivă este adevărată, când contorul se măreşte cu o unitate şi apelul recursiv este adevărat. Condiția de terminare este adevărată, dacă capul listei se unifică cu *E* şi *Cnt* poate fi unificat cu *Poz*.

Din nou se poate formula un apel mai simplu al regulii:

```
pozitie_ac(Poz,E,Lista):-
        pozitie_ac(1,Poz,E,Lista).
```

Întrebările de mai jos ilustrează unele utilizări ale predicatului.

```
?-pozitie_ac(2,E,[1,2,3])
E=2

?-pozitie_ac(Poz,2,[1,2,3])
Poz=2
```

3.6.3. Utilizarea eficientă a recursiei

Ar fi incorect să se creadă că numai utilizarea acumulatoarelor face ca predicatele să fie “bidirecționale” ca cel de mai sus. Recursia, eventual, este un mare consumator de memorie operativă, deoarece ea poate necesita memorarea tuturor calculelor pentru toate adâncimile recursiei. În programele Prolog efectiv mari, recursia este frecvent utilizată şi deci, şi consumul de memorie poate fi imens. Dar există tehnici de implementare a recursiei cu un consum de memorie foarte economicos. Să examinăm două versiuni ale unei reguli recursive.

```
lungime([_|Cd],Total):-
        lungime(Cd,Cont),
        Total=Cont+1.

lungime_ac([_|Cd],Suma,Total):-
        Cont=Suma+1,
        lungime_ac(Cd,Cont,Total).
```

În prima versiune, primul subscop (*lungime(Cd,Cont)*) este nondeterminist, fiindcă pot fi mai multe alternative de potrivire (unificare) a clauzei în programul Prolog. Al doilea subscop este determinist, fiindcă există o singură soluție pentru *Total=Cont+1*.

În a doua versiune, primul subscop (*Cont=Suma+1*) este determinist, iar al doilea (*lungime_ac(Cd,Cont,Total)*) - nondeterminist.

În Prolog utilizarea eficientă a memoriei se bazează pe determinism şi nondeterminism. Să mai considerăm odată versiunea doi. Întrucât primul subscop (*Cont=Suma+1*) este determinist, atunci când al doilea subscop (*lungime_ac(Cd,Cont,Total)*) este atins, Prologul nu trebuie să memoreze nimic despre subscopul precedent. Aceasta înseamnă, că chemarea recursivă poate utiliza memoria ce a fost ocupată de chemarea curentă. Deci, nu este nici o creştere a memoriei cu adâncirea recursiei.

În cazul primei versiuni, primul subscop este nondeterminist şi atunci Prologul nu poate suprascrie în memoria utilizată de chemarea curentă a recursiei. Deci, are loc un consum suplimentar de memorie.

Această tehnică de economisire a memoriei în procesul recursiei este o parte a tehnicii, numită *optimizarea prin recursia terminală* (Tail Recursion Optimisation). Numele se trage de la poziția în corpul regulii a subscopului recursiv (este ultimul sau nu). În opoziție cu recursia terminală se găseşte *recursia în stânga* (vezi prima versiune). Programatorii experimentați se identifică prin aplicarea largă a cozii recursive.

3.6.4. Inversarea eficientă a listei

Vom examina un predicat de inversare a ordinii elementelor în listă, cunoscut sub numele de “inversare naivă”. Acest predicat este deseori folosit ca etalon de măsurare comparativă a vitezei interpretatoarelor şi compilatoarelor Prolog. “Inversarea naivă” utilizează predicatul *concatenare/3*. O definiție poate fi

```
inversare_naiva([],[]).
inversare_naiva([C|Cd],Inv):-
        inversare_naiva(Cd,CdInv),
        concatenare(CdInv,[C],Inv).
```

La prima vedere predicatul pare inofensiv. Dar să examinăm clauza recursivă. Ea recursiv inversează *Cd* pentru a obține *CdInv* şi apoi alipeşte lista cu un singur element *C* la *CdInv*. Ineficiența acestui predicat foarte bine se poate observa, dacă este trasată întrebarea:

```
?-inversare_naiva([1,2,3],Inv)
```

Atunci s-ar putea vedea că se utilizează *28* de chemări, dintre care *7* chemări ale predicatului *inversare_naiva/2* şi *21* chemări ale predicatului *concatinare/3*. Numărul de chemări poate fi redus, dacă se utilizează un acumulator:

```
inversare_ac(L,Inv):-
        inversare_ac(L,[],Inv).

inversare_ac([],Inv,Inv).
inversare_ac([C|Cd],Rest,Inv):-
        inversare_ac(Cd,[C|Rest],Inv).
```

Această versiune porneşte cu argumentul doi al predicatului *inversare_ac/3* egal cu *[]* şi apoi utilizează constructorul de listă, adăugând *C* (la argumentul doi) la fiecare pas al recursiei. Condiția de terminare specifică că argumentele doi şi trei se unifică.

Trasând această versiune, se poate observa că ea conține doar *8* chemări. Este evident, că versiunea ce utilizează acumulatorul este eficientă ca şi variantele cu coada recursivă, în timp ce “versiunea naivă” nu este eficientă.

Începătorii în Prolog consideră că e mai uşor de elaborat predicate definite prin reguli ce nu au cozi recursive şi nu utilizează acumulatoarele. Dar acumulând experiență, cu timpul această obişnuință prevalează mai puțin.

© Vitalie Cotelea