Programarea în limbajul Lisp

Viorel NEGRU

October 14, 2003

Cuprins

Lista figurilor

Introducere

Prodomo

”High thoughts must have high language” Aristophanes - Frogs (405 B.C.)

Limbaje de programare

Rezolvarea unei probleme presupune, în general, pornind de la un enunţ, (mai mult sau mai puţin formalizat), proiectarea unui algoritm, a cărui implementare (codificare) va fi un program, scris într-un limbaj de programare, program ce va fi executat pe un calculator în vederea obţinerii soluţiei problemei. Execuţia unui program presupune declanşarea unui proces de calcul. Procesele de calcul sunt entităţi abstracte care, în evoluţia lor, manipulează alte entităţi abstracte, numite date. Evoluţia unui proces este condusă de o mulţime de reguli, ce formeaza programul.

Un limbaj de programare este o notaţie prin care programatorii comunică cu calculatoarele, precum şi cu alţi programatori. Limbajele de programare pot fi mai apropiate de limbajul cod-masină al calculatorului (limbaje de asamblare) sau mai îndepărtate (limbaje de nivel inalt). În funcţie de scop, întâlnim limbaje generale şi limbaje speciale.

Limbajelor de programare se pot clasifica în funcţie de paradigmele de programare utilizate: programare procedurală (C, Pascal); programare logică (Prolog); programare functională (Lisp, Scheme, ML); programare orientată obiect (Smalltalk, C++); programare orientată pattern-uri (OPS5, Clips); programare orientată pe componente etc.

O clasificare mai generală este legată de modul de rezolvare a problemei. Vorbim de programare declarativă (în care furnizăm calculatorului **ce** să rezolve) şi de programare imperativă (în care furnizăm calculatorului **cum** să rezolve problema). În primul caz este vorba de limbaje din clasa PROLOG, iar în al doilea caz de limbaje ca: Pascal, C, Lisp etc.

Majoritatea limbajelor de programare au fost create, în primul rând, pentru a efectua calcul numeric. Limbajele din clasa Lisp au fost create special pentru a efectua calcul simbolic. Limbajele Lisp şi Prolog se mai caracterizeaza şi prin faptul că au fost cele mai utilizate limbaje în Inteligenţa Artificială.

Limbajele de programare pot fi interpretative sau compilative. Limbajele interpretative sunt conversaţionale, prin raport cu limbajele compilative în care există o întârziere între introducerea şi execuţia programului. Limbajele interpretative sunt mai mult utilizate pentru dezvoltarea şi modificarea programelor (permit o interacţiune a programatorului uşoară şi rapidă cu calculatorul). Limbajele compilative sunt utilizate pentru "end users" (cerinţe de viteză şi mai puţin de posibilităţi de modificare).

Un limbaj de programare trebuie definit pentru a obţine programe corecte şi eficiente. Din păcate este foarte greu, daca nu imposibil, de a respecta, în acelaşi timp cele două condiţii. Limbajele din familia Lisp au fost create şi dezvoltate având în vedere mai întâi corectitudinea şi apoi eficienţa programelor. În schimb limbajele din familia C au avut în vedere mai întâi eficienţa şi apoi corectitudinea.

Isoric

Limbajul Lisp (LISt Processing) este unul din cele mai vechi limbaje de programare (al doilea limbaj ca vechime, primul fiind limbajul Fortran). Lisp a fost creat între anii 1956 - 1962 de către John McCarthy la MIT. Scopul a fost crearea unui limbaj algebric bazat pe prelucrarea listelor pentru a fi utilizat în Inteligenţa Artificială. Lisp a fost conceput pentru a face faţă la calcule simbolice (nenumerice) complicate. Crearea sa a fost influenţată de evoluţia limbajului FORTRAN, dar mai ales de limbajul specializat IPL2, folosit de Newell, Shaw si Simon pentru implementarea programului de demonstrare automată de teoreme "Logic Theorist" (program prezentat la conferinţa de Inteligenţă Artificială de la Dartmouth, în vara anului 1956).

Limbajul Lisp se bazează pe modele matematice formale ale calcului (teoria funcţiilor recursive). La baza limbajului stă calculul λ, elaborat de Alonzo Church în 1941. Mai exact este vorba de o apropiere de schema de reprezentare a funcţiilor parţial recursive definite peste anumite clase de expresii simbolice.

Prima versiune stabilă a fost Lisp 1.5 realizată de John McCarthy în 1962. Ulterior au aparut o mulţime de variante (dialecte) de Lisp.

În continuare au fost dezvoltate mai multe dialecte Lisp. MacLisp a fost primul dialect Lisp dezvoltat în laboratorul de IA de la MIT, de la sfârşitul deceniului 60 până la începutul deceniului 80. Un alt dialect care a dominat anii 70 alături de MacLisp a fost Interlisp dezvoltat la Xerox PARC. Alte dialecte Lisp: Portable Standard Lisp, Franz Lisp, Zeta Lisp etc.

În 1975 Sussman şi Steele crează un nou dialect Lisp - Scheme, o versiune care simplifică mult limbajul, fiind uşor de implementat şi de învăţat. Limbajul Scheme îl regăsim foarte mult în cercetare şi în invăţământ [?].

Au fost create maşini Lisp, cu arhitecturi specializate pentru execuţia rapidă a programelor Lisp (CADR - la MIT; DIALISP - la I. P. Bucureşti). De asemenea fost implementate versiuni Lisp pe mini si microcalculatoare (GCLisp, TC-Lisp).

În perioada 1981 - 1983 un comitet tehnic porneste un program de creeare a unui standard Lisp sub denumirea de Common Lisp. Deşi finalizat în 1983, Common Lisp apare efectiv odata cu publicarea de către Digital Press a cărţii: Common Lisp: The Language, cunoscută sub denumirea de CLTL1 [?]. În 1985 sub ANSI este creat comitetul X3J13 - Tehnical Committee for Programming Language Common Lisp. Standardizarea continuă, în 1990 fiind publicat CLTL2 [?].

În Franţa Jérôme Chailloux crează limbajul LeLisp, împreună cu un mediu de dezvoltare şi un nucleu obiect Ceyx [?]. În Europa este creată şi o replică a limbajului Common Lisp - EuLisp.

Între 1986 - 1988 are loc standardizarea limbajului obiect CLOS (Common Lisp Oriented System) [?]. CLOS este un nucleu obiect integrat in Common Lisp, fiind bazat pe limbajele New Flavors (Symbolics) şi Common Loops (Xerox).

Dintre variantele paralele ale limbajului Lisp amintim dialectele: Multilisp, Qlisp şi Butterfly PSL.

În 1997 ISO - grupul WG16 încearcă o standardizare Lisp internaţională (unificare Common Lisp şi EuLisp), primul draft find produs în 1992. În 1991 are loc standardizarea IEEE şi ANSI pentru Scheme, iar în 1992 are loc publicarea de către X3J13 a unui draft pentru Common Lisp [?].

Caracterizare

Lisp ne permite să învăţăm noi şi puternice posibilităţi de a gândi despre programe şi programare.

Lisp este un limbaj creat sa opereze mai mult cu *simboluri* decât cu numere. Operaţiile cu simboluri şi liste sunt cel mai bine reprezentate în limbaj. Reprezentarea informaţiilor în format extern se face cu ajutorul listelor multinivel şi, în general, cu ajutorul expresiilor simbolice. În format intern este utilizată structura de tip listă (cu pointeri). Primele versiuni de Lisp au utilizat ca structura de date agregată doar structura de tip lista. Common Lisp, din considerente de eficienţă, conţine şi alte tipuri de date agregate.

Lisp este însoţit de un mediu de dezvoltare. *Interactivitatea* limbajului, existenţa a peste 700 de funcţii definite în Lisp, cât şi a unor instrumente de dezvoltare /depanare (editoare de texte, editoare de forme Lisp, module de trasare a proceselor de evaluare, module pentru captarea şi tratarea erorilor, depanatoare simbolice, compilare incrementală, interfeţe grafice etc.) permit prototiparea rapidă de aplicaţii. Fiind un limbaj interpretativ cresc posibilităţile de modificare dinamică în timpul execuţiei. Cuplul interpretor - compilator permite o dezvoltare / testare de aplicaţii cu ajutorul interpretorului şi o reducere a timpului de execuţie prin compilarea la cerere a unor parţi (funcţii) din program. Natura interactivă şi incrementală a limbajului permite utilizarea sa pentru prototiparea rapidă de aplicaţii.

O caracteristică foarte importantă a limbajului Lisp este *uniformitatea.* Limbajul Lisp este unul din puţinele limbaje ce permit o flexibilitate deplină în definirea şi manipularea programelor ca si date. Programele Lisp şi datele Lisp au aceeaşi structură, find reprezentate prin expresii simbolice. Un program Lisp poate avea ca date de intrare un alt program Lisp sau, şi mai mult, un program Lisp poate produce (crea) un alt program Lisp. Dispare, astfel, diferenţa dintre date pasive si procese active.

Limbajul Lisp este *extensibil.* Functia Lisp eval este folosită atât pentru definirea formală a limbajului cât şi ca interpretor. Se pot adauga noi funcţii şi, chiar mai mult, se pot produce alterări sintactice cu ajutorul macrourilor. Acest lucru permite dezvoltarea de utilitare speciale care se integrează foarte bine în mediul Lisp. Limbajul Lisp devine astfel un limbaj de programare programabil. Unele funcţii pot fi redefinite, limbajul Lisp devenind şi un limbaj adaptabil (se pot translata uşor programe dintr-un dialect în alt dialect).

Limbajul Lisp este *flexibil.* Aceasta flexibilitate rezultă din utilizarea ca entitate de bază a noţiunii de funcţie şi din facilitatea furnizată de utilizarea macrourilor. Lisp permite crearea de funcţii în timpul execuţiei unui program. Macrourile permit extinderea limbajului de bază, astfel că adaugarea de noi stiluri de programare se poate realiza prin definirea de noi macrouri. Lisp poate suportă astfel programarea procedurală (orientată stări, bazată dominant pe atribuire); programarea funcţională; programarea orientată obiect; programarea bazată pe pattern-uri.

Limbajul Lisp este un mediu excelent pentru experimentarea limbajelor de programare. Lisp permite utilizarea recursivităţii, a închiderilor lexicale şi a tipării în timpul execuţiei. Recursivitatea permite definirea abstractă a algoritmilor. Toate acestea alături de macrouri fac din Lisp un limbaj ce permite cu uşurinţă definirea de noi *abstractizări*, cum ar fi, de exemplu, definirea unui limbaj orientat obiect propriu în Lisp.

Noţiunea centrală în Lisp este cea de funcţie. Un program este format din una sau mai multe funcţii. *Programarea funcţională* presupune existenţa doar a funcţiilor care returnează valori şi nu au efecte laterale. Un efect lateral produce o schimabare de stare a contextului (mediului) ca o consecinţă a evaluării unei expresii. Unul din avantajele programării funcţionale este dat de posibilitatea testării interactive, fiecare funcţie putând fi testată atunci când este scrisă. Lipsa efectelor laterale într-o funcţie are drept consecinţă suficienţă existenţei unei singure expresii în corpul funcţiei, celelalte neavând nici un efect asupra rezulatului. Limbajul Lisp nu este pur funcţional, dar conţine un subset ce îndeplineşte această condiţie. Trebuie eliminate din Lisp primitivele de legare (atribuire), primitivele de I/O şi toate funcţiile care produc efecte laterale.

Aplicaţiile de calcul simbolic (din Inteligenta Artificială, Algebră Computaţională etc.) operează, în general, cu structuri dinamice de date. Limbajul Lisp operează cu liste, care sunt structuri implicite cu pointeri. *Gestiunea automată a memoriei* (alocarea şi eliberarea memoriei) este realizată cu ajutorului unui Garbage Collector. Nu mai este, astfel, nevoie de alocare / eliberare explicită de memorie, deşi Lisp dispune de primitive pentru astfel de operaţii.

Programare bottom-up

Limbajul Lisp este compus din funcţii şi macrouri care sunt la fel ca cele scrise de utilizator. Acest lucru permite ca extinderea limbajului să nu fie mai dificilă decât scrierea de programe. Limbajul Lips permite atât scrierea de programe top-down, cât şi bottom-up, abordarea bottom-up fiind mult mai naturală.

Programarea bottom-up presupune scrierea aplicaţiei ca nivele succesive dezvoltate pornind de la limbajul Lisp, fiecare nivel find ca un limbaj de programare pentru nivelul de deasupra. Astfel, în loc să depunem efortul să coborâm aplicaţia spre limbajul de programare, urcăm limbajul spre aplicaţie. Are loc, astfel, separarea formei generale de cea specifică. În final, aplicaţia este scrisă în limbajul de pe ultimul nivel, limbaj mult mai apropiat de termenii aplicaţiei. Această abordare are câteva avantaje imediate: producerea de software extensibil; obţinerea de software reutilizabil; prototiparea rapidă a aplicaţiei.

Aplicaţii

Aplicatii: Macsyma (in MacLisp).

Emacs, Autocad, Interlief

Continut

Capitolul 1

Noţiuni de bază Lisp

1.1 Noţiuni introductive

1.1.1 Interpretorul Lisp

Bazându-se pe o sintaxă simplă limbajul Lisp este uşor de învăţat. Un program Lisp este format din expresii simbolice şi prelucrează expresii simbolice. Unitatea de bază într-un program este funcţia, un program fiind format din una sau mai multe funcţii[1].

Un sistem Lisp conţine o interfaţă (front-end) interactivă numită *top-level*. La lansarea interpretorului Lisp este afişat un *prompter Lisp* specific versiunii Common Lisp utilizate (`>`, `:`, `*` etc). În multe cazuri este utilizat prompterul `>`, prompter pe care o sa-l folosim şi noi în continuare.

Funcţionarea interpretorul Lisp se bazează pe repetarea unui *ciclu de bază*: `read-eval-print` (top-level loop), format din trei etape sau stări:

1. `read`: citeşte o expresie simbolică;
2. `eval`: evaluează expresia simbolică introdusă;
3. `print`: afişează rezultatul obţinut în urma evaluării expresiei.

La apariţia prompterului, interpretorul Lisp începe un nou ciclu de bază trecând în starea `read` (aşteaptă introducerea unei noi expresii simbolice).

[1] În realitate în Lisp pe lângă funcţii o să mai intâlnim forme speciale şi macrouri. Unii autori ([?]) le încadrează în conceptul mai general de procedură. Funcţia este o procedură în care intrările sunt date de argumente, iar iesirea (rezultatul) de valoarea întoarsă. Un exemplu este funcţia din matematică. Un program este implementarea unei proceduri într-un limbaj de programare. Deoarece în Lisp nu sunt deosebiri majore între program, procedură şi funcţie noi o să folosim în continuare termenul de funcţie.

Spre exemplu, dacă introducem simbolul `pi`, urmat de tasta < *enter* >, `pi` va fi evaluat, iar valoarea va fi afisată:

```
>pi
3.14159
```

1.1.2 Elemente de baza

Atomi şi liste

Cele mai simple entităţi în limbajul Lisp sunt *atomii*. Atomii pot fi atomi numerici sau *numere*, respectiv atomi simbolici sau *simboluri*. Numerele se evaluează la ele însele.

```
>16                     >1.25
16                      1.25
```

Simbolurile[2] se evaluează la valorile la care au fost legate. Să presupunem ca simbolul `suma` este legat la valoarea 16 (`suma` $\longrightarrow$ `16`) şi simbolul `locul-nasterii` $\longrightarrow$ `ARAD`):

```
>suma                   >locul-nasterii
16                      arad
```

Încercarea de evaluare a unui simbol ce nu a fost legat la o valoare produce eroare:

```
>a
error: unbound variable - a
```

O categorie aparte o reprezintă constantele de tip şir de caractere care conţin zero sau mai multe caractere între ghilimele. De exemplu: ”sir de caractere”. Un şir de caractere se evaluează la el însăşi:

```
>"Sir de caractere"
"Sir de caractere"
```

[2]În cadrul fazei `read` din ciclul *read-eval-print*, are loc, implicit, transformarea, în cazul simbolurilor, a literelor mici în litere mari (variabila globala `*print-case*`[3] este legată la valoarea `:upcase`[4]). Daca dorim ca să avem o transformare din litere mari în litere mici atunci `*print-case*` se leagă la valoarea `:downcase`. Efectul acestei transformari se observă în faza `print`. În această prezentare considerăm că `*print-case*` este leagă la valoarea `:downcase`.

O listă constă din zero sau mai multe elemente (atomi sau liste), separate prin spaţii şi cuprinse între paranteze rotunde. Exemple de liste:

```
(), (a b c), (a (b (c))), (+ 1 2 3), (aceasta este o lista)
```

Cu ajutorul listelor putem reprezenta mulţimi, arbori, grafuri, expresii matematice etc.

Ca atomi, în Lisp, avem constantele `t` şi `nil`. Prima are semnificaţia de adevarat (true), iar a doua de fals (false). Constanta `nil` mai reprezintă şi lista vida (`nil` $\equiv$ `()`), `nil` fiind singura entitate Lisp cu dublă semnificaţie.

```
>t                    >nil                  >()
t                     nil                   nil
```

Comentarii şi scrierea unui program în Lisp

Comentariile în Lisp sunt de forma:

```
;<text-oarecare>
```

Efectul întâlnirii macrocaracterului[5] `;` în faza `read` a ciclului `read-eval-print` este ignorarea tuturor caracterelor de după `;` până la sfârşit de linie.

Comentariile pot fi singure pe linie sau combinate cu text sursă Lisp. Sunt preferate comentariile la nivel de linie. Acestea pot fi stratificate (de exemplu cel mai general comentariu va conţine la început de linie patru - cinci caractere `;`, iar cel mai specific un caracter `;`).

Funcţiile în Lisp se scriu indentate, în general ţinând cont de paranteze:

```
;;; Calculul factorialului
;;;
(defun fact (n)
   ;;conditia de oprire
   (if (zerop n)
        1    ; 1 <-- (fact 0)
        ;;apelul recursiv
        (* n (fact (1- n)))
   )
)
```

[5]Un macrocaracer este un caracter căruia i se ataşează o funcţie care este apelată în momentul în care parser-ul `read` întâlneşte caracterul respectiv, funcţie care potenţial poate modifica expresia simbolică prelucrată de `read`.

Expresii simbolice

Listele şi atomii formează expresiile simbolice sau s-expresiile în Lisp. O definiţie (recursivă) a expresiilor simbolice este următoarea:

1. Atomii sunt expresii simbolice;
2. O listă este o construcţie de forma () sau $(e_1, e_2, \ldots, e_n)$, unde $n \geq 1$ şi $e_1, e_2, \ldots, e_n$ sunt expresii simbolice;
3. O pereche cu punct[6] este o construcţie de forma $(e_1\ .\ e_2)$, unde e_1 ,si e_2 sunt expresii simbolice;
4. Listele şi perechile cu punct sunt expresii simbolice.

Exemple de expresii simbolice:

```
1, abc (), (a . b), ((a) (b c (d)) e), "ab1", ("a" 2 b)
```

Funcţii

Funcţiile în Lisp sunt obiecte obişnuite, la fel ca simbolurile şi listele.

Apelul unei funcţii este reprezentat de o listă în care primul element reprezintă funcţia, iar celelalte elemente argumentele funcţiei. De exemplu, forma (+ 1 2 3) este compusă din funcţia (operatorul) de adunare + şi din argumentele (operanzii) 1 2 şi 3.

```
>(+ 1 2 3)                 >(sqrt 4)
6                          2

>(* (+ 2 3) 10)
50
```

Evaluarea unei liste care nu este o formă va produce eroare:

```
>(a 1 2)
error: unbound function - a
```

Funcţiile pot fi funcţii sistem sau funcţii definite de utilizator.

[6]Perechea cu punct are un efect mai mult intern, pentru reprezentarea mai compactă listelor în memorie

1.1.3 Evaluarea

O expresie simbolică ce poate fi evaluată se numeşte *formă*. Daca o formă este reprezentată printr-o listă distingem trei interpretări ale acestei forme: apel de funcţie, apel de formă specială, respectiv apel de macro. Într-un apel de funcţie se aplică funcţia (dată de primul element) asupra argumentelor (restul elementelor) evaluate. O formă specială evaluează / nu evaluează argumentele conform unor reguli proprii.

Evaluarea implicită are loc în cadrul ciclului `read-eval-print`, în faza `eval`. Funcţia de evaluare acţionează astfel:

1. dacă expresia este atom, întoarce valoarea sa;

2. dacă expresia este o listă:

   (a) dacă primul element din listă reprezintă o funcţie, regăseşte această funcţie şi

      i. evaluează restul elementelor din listă aplicând aceleaşi reguli la fiecare din ele;

      ii. aplică funcţia de la a) la argumentele de la i) şi întoarce rezultatul.

   (b) daca primul element reprezintă o formă specială, aplică un tratament specific asupra argumentelor sale şi asupra formei speciale;

   (c) dacă primul element reprezintă un macro, aplică un tratament specific macrourilor.

Stoparea evaluării

În situaţiile în care dorim ca o expresie simbolică să nu fie evaluată folosim forma speciala `quote`. Forma specială `quote` nu îşi evaluează argumentul, valoarea întoarsă fiind data de argument:

```
(quote <expr-simb>)
```

Se poate folosi si o notatie simplificată cu ajutorul macrocaracterului `'`. Astfel, `(quote <arg>)` ≡ `'<arg>`. Exemple de utilizare:

```
>(quote a)          >(quote (a b c))         >'a
a                   (a b c)                  a
```

Funcţia `eval` folosită de interpretor poate fi apelată şi explicit. Funcţia îşi evaluează argumentul, valoarea întoarsă fiind rezultatul obţinut prin evaluarea argumentului evaluat:

Exemple de utilizare:

```
>(eval '(+ 1 2))          >(car '(+ 1 2))
3                         +
```

Dacă presupunem că `y` $\longrightarrow$ `x`, iar `x` $\longrightarrow$ `10`:

```
>(eval y) ;y (ca argument) este evaluat la x, iar x la 10
10
```

1.1.4 Atribuirea şi legarea variabilelor

Până acum am văzut că datele cu care operăm sunt expresii simbolice (atomi, liste). Datele ocupă locaţii de memorie. Forma de reprezentare şi conţinutul locaţiei de memorie depind de tipul datelor. Un simbol ce desemnează valoarea dintr-o locaţie de memorie se numeste *variabilă*. Asocierea unei variabile la o dată se numeşte *legare*, fiind echivalentul atribuirii din alte limbaje.

Spre deosebire de alte limbaje de programare în Lisp locaţia de memorie conţine amprenta tipului datei, tip ce se atribuie variabilei în momentul legării variabilei la valoarea din locaţia de memorie. Această operaţie se mai numeşte şi tipare în timpul execuţiei.

Pentru legarea variabilelor avem următoarele funcţii[7] Lisp: `set, setq` pentru legarea secvenţială a variabilelor, respectiv `pset psetq` pentru legarea paralelă a variabilelor.

`Setq` este formă specială şi are forma generală:

$$(\texttt{setq} <\texttt{var}_1> <\texttt{val}_1> \ \ldots \ <\texttt{var}_n> <\texttt{val}_n>)$$

Argumentele de ordin impar trebuie să fie simboluri şi nu se evaluează, argumentele de ordin par se evaluează, valoarea întoarsă este dată de ultimul argument evaluat, iar ca efect lateral $<\texttt{var}_i> \longrightarrow <\texttt{val}_i>$ evaluate.

Exemple:

```
>(setq x 1 y '(a b c))       >(setq x 1 y (+ x 2))
(a b c)                      3
>x                           >x
```

[7]Prin abuz de limbaj o sa folosim termenul de funcţie şi pentru forme speciale, urmând ca, acolo unde este cazul, să facem distincţia necesară

```
1                              1
>y                             >y
(a b c)                        3
```

`Set` este funcţie şi are forma generală:

$$(\texttt{set } <\texttt{var}_1> <\texttt{val}_1> \ \texttt{...} \ <\texttt{var}_n> <\texttt{val}_n>)$$

În urma evaluării argumentelor, valoarea întoarsă este dată de ultimul argument evaluat, iar ca efect lateral $<\texttt{var}_i>$ evaluate la simboluri $\longrightarrow$ $<\texttt{val}_i>$ evaluate.

Exemple:

```
>(set 'x 1 'y 2)               >(setq y 'x)
2                              x
>x                             >(setq x 'a)
1                              a
>y                             >(set y 2)
2                              2
                               >y
                               x
                               >x
                               2
```

Observaţie: `(set 'x 1)` $\equiv$ `(setq x 1)`.

Funcţiile `pset` şi `psetq` au aceeaşi formă cu `set` şi `setq`, diferenţa constând în modul de legare a valorilor la variabile. Legarea secvenţială presupune, după fiecare evaluare $<\texttt{val}_i>$, legarea la $<\texttt{var}_i>$ a valorii obţinute. În cazul legării paralele, are loc evaluarea tuturor $<\texttt{val}_i>$ şi apoi legarea la $<\texttt{var}_i>$ a valorilor obţinute (are loc o legare multicontext - sau într-un paralelism virtual). O comparaţie între setq şi psetq este prezentată în exemplul următor:

```
>(setq x 10)                   >(setq x 10)
10                             10
>(setq x 1 y (+ x 2))          >(psetq x 1 y (+ x 2))
3                              12
>x                             >x
1                              1
>y                             >y
3                              12
```

1.2 Operaţii cu liste

O listă conţine elemente care pot fi atomi sau liste. Listele pot fi prelucrate pe nivelul superficial sau în adâncime. Nivelul superficial se referă la primul nivel de elemente. Daca toate elementele unei liste sunt atomi avem doar un nivel (nivelul superficial).

Operaţiile principale asupra listelor se referă, în principal, la selectarea unor elemente (utilizare accesori) din listă şi crearea de noi liste (utilizare constructori).

Reprezentarea internă a unei liste este dată de o structură arborestenta. Lista `(a b c)` are structura internă dată de Figura 1.1.

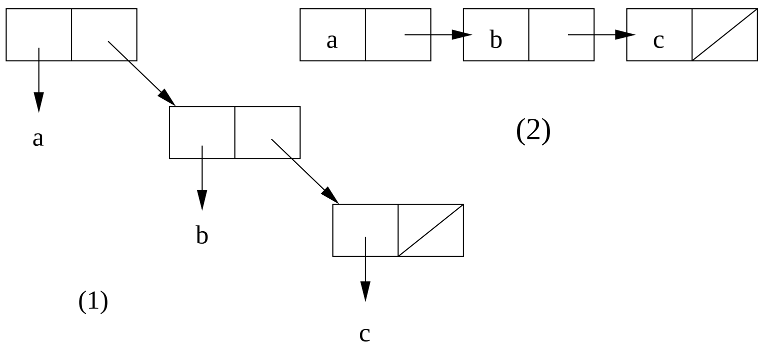


Figura 1.1: Structura internă a listei (a b c): (1) Descrierea arborescenta; (2) Descrierea simplificată.

Un alt exemplu este dat de reprezentarea internă a listei `((a (b c)) d ((e f) g) h)` (vezi figurile 1.2,1.3)

1.2.1 Funţiile car, cdr şi cons

Funcţia `car`[8] primeşte ca argument o lisă şi întoarce primul element al listei sau partea stângă a unei perechi cu punct. Se poate folosi o funcţie echivalentă şi cu un nume mai semnificativ: `first`.

```
(car <lista>)
```

Exemple de utilizare:

[8]Denumirile de car şi cdr vin de la numele unor regiştri de la calculatorul pe care s-a implementat prima versiune Lisp

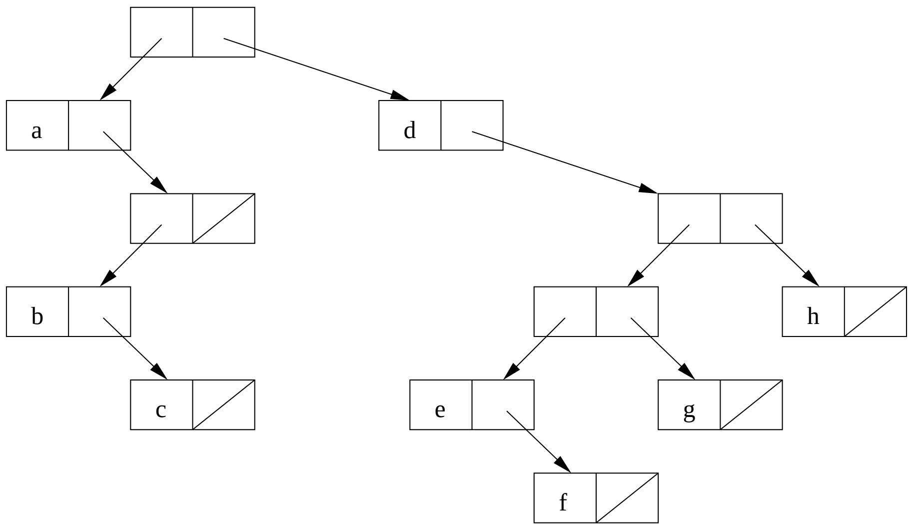


Figura 1.2: Structura internă a listei ((a (b c)) d ((e f) g) h), descrierea arborescenta.

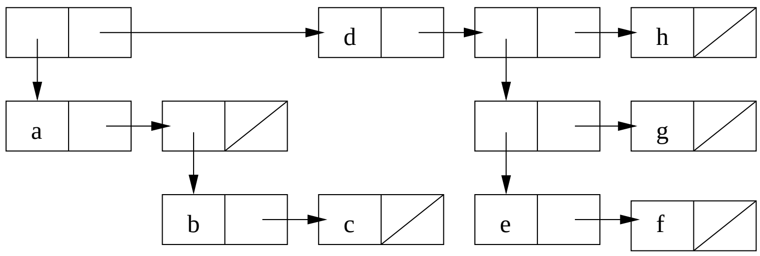


Figura 1.3: Structura internă a listei ((a (b c)) d ((e f) g) h), descrierea pe nivele.

```
>(car '(a b c))
a
>(car '(1 . 2))
1
>(car '((a (b c)) d ((e f) g) h))
(a (b c))
```

```
>(car 'a)
error: bad argument type - a
>(first '(a b c))
a
```

Funcţia `cdr` primeşte ca argument o lisă şi întoarce lista mai puţin primul element sau partea dreaptă a unei perechi cu punct. Se poate folosi o funcţie echivalentă şi cu un nume mai semnificativ: `rest`.

```
(cdr <lista>)
```

Exemple de utilizare:

```
>(cdr '(a b c))
(b c)
>(cdr '(1 (2 . 3)))
((2 . 3))
>(dar '((a (b c)) d ((e f) g) h))
(d ((e f) g) h)
```

```
>(cdr 'a)
error: bad argument type - a
>(rest '(a b c))
(b c)
```

Prin convenţie `car` şi `cdr` din `nil` este `nil`.

Funcţia `cons` (denumirea vine de la CONStructor) crează o listă (celulă cons) care are car-ul dat de primul argument şi cdr-ul de al doilea argument (Figura 1.4). Rezultatul va fi o listă (când al doilea argument este o lista) sau o pereche cu punct (când al doilea argument este un atom).

```
(cond <el> <lista>)
```

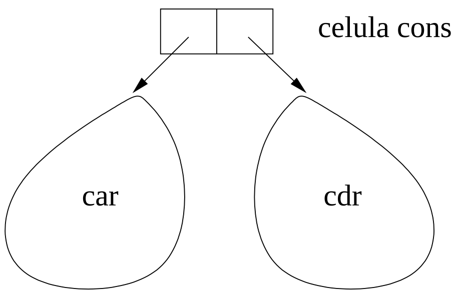


Figura 1.4: Structura unei celule cons.

Exemple de utilizare:

```
>(cons 'a '(b c))
(a b c)
>(cons '(a) '(b c))
((a) b c)
>(cons 1 nil)
(1)
```

```
>(cons nil nil)
nil
>(cons 1 2)
(1 . 2)
>(cons '(a b) 'c)
((a b) . c)
```

Funcţiile `car` şi `cdr` pot fi compuse:

```
>(car (car (cdr '(a ( b c) d))))
b
```

Aplicarea funcţiilor se face de la drepta spre stânga. În funcţie de implementare, există, pe lângă `car` şi `cdr` funcţiile `cxxr, cxxxr, cxxxr` cu `x` egal cu `a` sau `d`. Funcţia echivalenta cu apelul din exemplul anterior este:

```
>(caadr '(a (b c) d))
b
```

O combinaţie de `car, cdr` şi `cons` (vezi Figura 1.5) din care rezultă relaţia dintre acestea este dată în exemplul următor:

```
>(setq l '(a b c))
(a b c)
>(cons (car l) (cdr l))
(a b c)
```

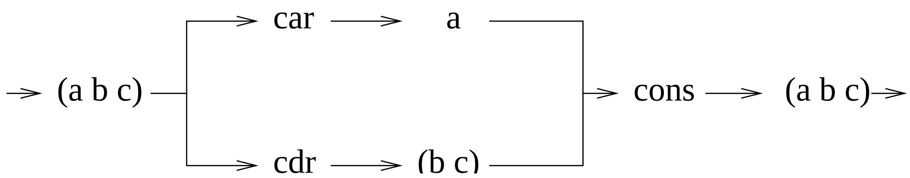


Figura 1.5: Relaţia dintre car, cdr şi cons.

1.2.2 Alte funcţii

În continuare descriem pe scurt următoarele funcţii: `append, list, reverse, last` şi `length`.

Funcţia `append` are un mumăr variabil de argumente (argumente ce trebuie să se evalueze la liste) şi are ca efect concatenarea elementelor listelor date de argumentele funcţiei.

Forma generală a funcţei este:

$$\texttt{(append } < \texttt{lista}_1 > \ < \texttt{lista}_2 > \ \ldots \ < \texttt{lista}_n > \texttt{)}$$

Ultimul argument poate fi atom. În acest caz ultima celulă cons devine pereche cu punct.

Exemple de utilizare:

```
>(append '(a) '(b c))
(a b c)
>(append '((a) b) '(c) '(d (e f)))
((a) b c d (e f))
>(append 'a '(b c))
error: bad argument type - a
```

```
>(append '((a)) '(b c) 'd)
((a) b c . d)
>(append)
nil
```

Funcţia `list` are un număr variabil de argumente, argumentele evaluate putând fi orice expresie simbolică. valoarea întoarsă este dată de concatenare argumentelor evaluate.

Forma generală este:

$$(\texttt{list}\ <\texttt{sexpr}_1> \ <\texttt{sexpr}_2> \ \ldots \ <\texttt{sexpr}_n>)$$

Exemple de utilizare:

```
>(list 1 2 3)
(1 2 3)
>(list '(a b) 'c '((d e) f))
((a b) c ((d e) f))
>(list 1 '(2 . 3))
(1 (2 . 3))
>(list nil nil)
(nil nil)
```

O comparaţie între `cons, append` şi `list` este prezentată în exemplul următor:

```
>(cons '(a) '(b c))
((a) b c)
>(append '(a) '(b c))
(a b c)
>(list '(a) '(b c))
((a) (b c))
```

Funcţia `last` întoarce ultima celulă cons a listei primite ca argument.

Forma generală este:

```
(last <lista>)
```

Exemple de utilizare:

```
>(last '(a b c d))
(d)
>(last '(a b . c))
(b . c)
>(last '(a))
(a)
```

Funcţa `reverse` are ca argument o listă şi întoarce ca şi rezultat o listă cu elementele de pe nivelul superficial inversate.

Forma generală este:

```
(reverse <lista>)
```

Exemple de utilizare:

```
>(reverse '(1 2 3 4 5)
(5 4 3 2 1)
>(reverse '(a (b c d) e))
(e (b c d) a)
```

Funcţia `length` întoarce numărul de elemente de pe nivelul superficial al listei primite ca argument.

Forma generală este:

```
(length <lista>)
```

Exemple de utilizare:

```
>(length '(a b c))
3
>(length '((a b (c)) (d e)))
2
>(length ())
0
```

1.3 Probleme propuse

Capitolul 2

Funcţii utilizator

Funcţiile în Lisp pot fi funcţii sistem sau funcţii definite de utilizator. Fucţiile definite de utilizator permit împreună cu macro-urile extinderea limbajului Lisp. Funcţiile sistem sunt funcţii scrise, pentru eficienţă, în cod. Funcţiile utilizator, tot din considerente de eficienţă, pot fi compilate.

2.1 Definirea funcţiilor

2.1.1 Definire

Definirea funcţiilor utilizator se face cu ajutorul macro-ului[1] `defun`[2].

Forma generală este:

```
(defun <nume-func> <lista-param>
        <expr-1> <expr-2> ... <expr-n>)
```

unde:

- <`nume-func`> este primul argument şi reprezintă numele funcţiei definite de `defun`;
- <`lista-param`> este al doilea argument al lui `defun`, are forma (<`par-1`> <`par-2`> ... <`par-m`>) şi reprezintă lista cu parametri pentru funcţia definită;
- <`expr-i`>, $i = 1, \ldots, n$ sunt forme ce alcătuiesc corpul funcţiei definite.

[1]Pentru a obţine o eficienţă mai bună, în multe implementări ale limbajului Lisp - o parte din macro-urile sistem au fost implementate ca şi forme speciale

[2]În *Scheme* se foloseşte `define`

`Defun`, în cazul în care definiţia este corectă sintactic, întoarce numele funcţiei (valoarea primului argument). `Defun` are un număr variabil de parametri şi în cadrul unui apel nu îşi evaluează argumentele.

Efectul lateral al definirii unei funcţii este crearea în mediul Lisp a unui obiect Lisp de tip funcţie ce are ca şi nume primul argument, parametrii fiind daţi de al doilea argument şi corpul funcţiei este dat de restul argumentelor.

Exemple de utilizare:

```
>(defun patrat (x)        ; patratul unui număr
   (* x x))
patrat

>(defun calcul (x y z)    ; calculează valoarea unei expresii
   (+ x (* y z))
calcul
```

2.1.2 Apel

În definiţia unei funcţii avem *parametri* (echivalentul parametrilor formali din alte limbaje), iar în apel avem *argumente* (echivalentul parametrilor actuali din alte limbaje).

Apelul unei funcţii are forma:

```
(<nume-func> <arg-1> <arg-2> ... <arg-n>)
```

unde: <`nume-func`> reprezintă numele funcţiei, iar <`arg-i`>, $i = 1, \ldots, n$ argumentele funcţiei. Valoarea întoarsă în urma apelului este ultima formă evaluată din corpul funcţiei.

Exemple de apel:

```
>(patrat 2)                  >(calcul 2 3 4)
4                            14
>(setq y 3)                  >(setq x 3 y 2 z 4)
3                            4
>(patrat y)                  >(calcul x y z)
9                            11
```

2.1.3 Evaluare

În urma unui apel de funcţie se parcurg următoarele etape:

1. Se identifică funcţia;
2. Se evaluează argumentele;
3. Parametrii formali sunt legaţi la argumentele evaluate (Figura **??**. Dacă înainte de apel parametrii au fost legaţi, valorile acestora se salvează, urmând a se restaura după revenirea din funcţie; Un parametru nelegat înainte de apelul funcţiei, redevine nelegat după revenirea din funcţie.
4. se evaluează corpul funcţiei;
5. valoarea întoarsă este dată de valoarea ultimei expresii simbolice din corpul funcţiei.

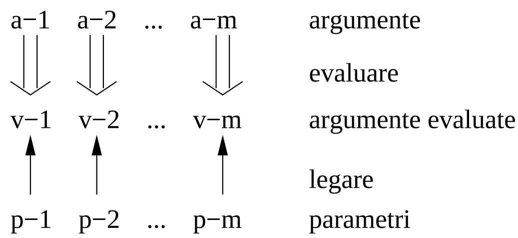


Figura 2.1: Legarea parametrilor.

Redefinirea funcţiilor sistem provoacă, în general, eroare sau avertizare în funcţie de implementarea Lisp. Oricum, nu este indicată redefinirea funcţiilor sistem, decât în cazul unor extensii menite să schimbe comportamentul limbajului Lisp.

În schimb se poate efectua, fără probleme, renumirea unor funcţii siste. De exemplu:

```
>(defun primul (x)   ;funcţia primul va avea aceeaşi
    (car x))         ;comportare ca şi funcţia car
primul
```

2.1.4 Variabile legate şi variabile libere

Variabilele utilizate în Lisp se gasesc, în general, în una din situaţiile: ca argumente în `set, setq, pset, psetq, setf` sau `defvar`; ca variabilă într-o listă de parametri ai unei definiţii de funcţie. În primul caz variabilele se numesc *variabile globale*, iar în al doilea caz *variabile locale*.

O variabilă care apare în lista de parametri a unei funcţii se mai numeşte *variabilă legată* în raport cu acea funcţie, iar o variabilă care apare în corpul funcţiei şi nu apare în lista de parametri se numeşte *variabilă liberă* în raport cu acea funcţie.

```
>(setq x 1 y 2)
>(defun f1 (x)
    (+ x y))
>(f1 3)
5
>x
1
>y
2
```

```
>(setq x 1 y 2)
>(defun f2 (x)
    (setq x 10)
    (+ x y)
>(f2 x)
12
>x
1
```

```
>(setq x 1 y 2)
>(defun f3 (x)
    (setq x 10 y 20)
    (+ x y))
>(f3 x)
30
>y
20
```

```
>(setq x 1 y 2)
>(defun f4 (x)
    (setq x 10)
    (+ (symbol-value ) y)
>(f4 x)
3
>x
1
```

În primul exemplu `x` este variabilă legată, iar `y` este variabilă liberă. În al doilea exemplu se observă că deşi valoarea lui `x` a fost schimbată în funcţia `f2` folosind `setq` , `x` fiind o variabilă locală, la ieşirea din funcţie valoarea lui `x` este cea dinainte de apel. Variabila globală `x` are acelaşi nume cu variabila locală, în acest caz variabila locală este cea vizibilă.

În exemplul al treilea modificarea valorii lui `y` în cadrul funcţiei `f3`, `y` fiind variabilă liberă, are efect şi după părăsirea funcţiei. În ultimul exemplu funcţia `symbol-value` întoarce valoarea globală a lui `x` şi nu valoarea sa locală.

După cum se observă din exemplele anterioare nu este indicată utilizarea variabilelor globale în cadrul unei funcţii. Funcţia depinde de modificarea variabilelor globale, comportarea funcţiei modificându-se atunci când se schimbă valoarea variabilei globale.

Un context (mediu) în Lisp este dat de o mulţime de legături (variabilele cu valorile la care sunt legate, definiţii de funcţii etc.). Contextul din momentul definirii unei funcţii se numeşte *context de definire*, iar contextul în care se evaluează funcţia de numeşte *context de evaluare*.

Funcţiile pot fi imbricate, o funcţie fiind apelată din altă funcţie. Fiecare funcţie va fi

2.1.5 Modul de transmitere a parametrilor

2.2 Expresii condiţionale

2.2.1 Cond

2.3 Predicate Lisp

Capitolul 3

Recursivitatea

3.1 Definirea recursivităţii

Un obiect este recursiv daca este definit funcţie de el însăşi. Întâlnim, astfel, termeni ca: functii recursive, proceduri recursive, definiţii recursive de date, calcul recursiv.

Recursivitatea ne oferă posibilitatea de a defini un număr infinit de obiecte printr-o declaraţie finită, respectiv de a descrie un număr infinit de operaţii printr-un program recursiv finit.

Exemple de definiţii recursive:

1. `GNU = Gnu is Not Unix`

2. numerele naturale:

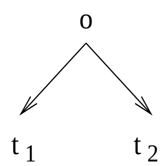
   (a) 0 este număr natural;

   (b) succesorul unui număr natural este un număr natural.

3. arborii binari:

   (a) *o* este un arbore binar (arborele vid);

   (b) dacă t_1 şi t_2 sunt arbori binari atunci şi

   este un arbore binar.

4. factorialul unui număr:

$$n! = \begin{cases} n * (n-1)! & \text{dacă} \quad n > 0 \\ 1 & \text{dacă} \quad n = 0 \end{cases}$$

Definiţiile recursive de date pot fi reprezentate şi cu ajutorul notaţiei BNF. Spre exemplu dacă dorim să definim tipul de date `listă-de-numere` ca cea mai mică mulţime ce satisface următoarele proprietăţi:

1. lista vidă este o `listă-de-numere`;

2. dacă l este o `listă-de-numere` şi n este un număr, atunci perechea `(n . l)` este o `listă-de-numere`.

În notatţia BNF avem următoarele reguli:

```
<listă-de-numere>::=()
<listă-de-numere>::=(<număr> . <listă-de-numere>)
```

sau utilizând simbolul bară verticală din BNF:

```
<listă-de-numere>::=() | (<număr> . <listă-de-numere>)
```

sau utilizând asteriscul (Kleen star):

```
<listă-de-numere>::=({<număr>}*)
```

O astfel de definiţie se mai numeşte şi definiţie bazată pe inducţie structurală. Când definim un program bazat pe inducţie structurală - structura programului trebuie să reflecte structura datelor, iar apelurile recursive sunt efectuate în punctele în care recursivitatea este utilizată în definirea inductivă a tipului de date.

3.2 Recursivitatea în Lisp

O funcţie ce se apelează pe ea însăşi se numeşte funcţie recursivă.

În limbajul Lisp recursivitatea joacă un rol mai important decât în alte limbaje, motivele principale fiind [?]:

- Recursivitatea ne permite să vedem algoritmii într-un mod mai *abstract*. Putem, astfel, verifica dacă o funcţie recursivă este corectă fără a considera toate apelurile ce rezultă în cazul în care funcţia este apelată.

- Utilizarea implicită a pointerilor în Lisp permite utilizarea cu uşurinţă a *structurilor de date recursive*.

- Algoritmii recursivi reduc posibilitatea de a avea efecte laterale, fiind în acord cu cerinţele *programării funcţionale*.

- Algoritmii recursivi sunt, în general, mai *eleganţi* decât cei iterativi.

Dacă comparăm versiunea recursivă cu versiunea iterativă a unui algoritm observăm că, în general, versiunea iterativă este mai clară (algoritmul este mai simplu de scris si mai lizibil), dar mai puţin eficientă (consum mai mare de memorie şi timp, la fiecare apel fiind creat un nou set de variabile locale).

Funcţiile recursive pot fi *direct recursive* (f apelează f) sau *indirect recursive* (f apelează g_1, g_1 apelează g_2, ..., g_k apelează f). Al doilea caz, pentru $k = 0$, se reduce la primul caz, iar pentru $k = 1$ devine apelul mutual recursiv a două funcţii.

Funcţia Lisp corespunzătoare definiţiei de mai sus a factorialului este următoarea:

```
(defun fact (n)
  (cond ((zerop n) 1)                ; condiţia de terminare
        (t (* n (fact (1- n)))) ))   ; apelul recursiv
```

sau

```
(defun fact (n)
  (if (zerop n)
       1                ; condiţia de terminare
       (* n (fact (1- n))) ))  ; apelul recursiv
```

Arborele inversat pentru apelul `>(fact 3)` este dat de figura 3.1.

3.3 Corectitudinea unui algoritm recursiv

În cazul unui algoritm recursiv, demonstrarea corectitudinii algoritmului este mai simplă. Pentru a ne asigura că o funcţie recursivă funcţionează corect este suficient de a verifica dacă acoperă toate cazurile. Astfel, în cazul calculului factorialului trebuie verificate următoarele:

1. lucrează corect pentru factorial de 0;

2. fiind dat că lucrează corect pentru n, va lucra corect şi pentru $n + 1$.

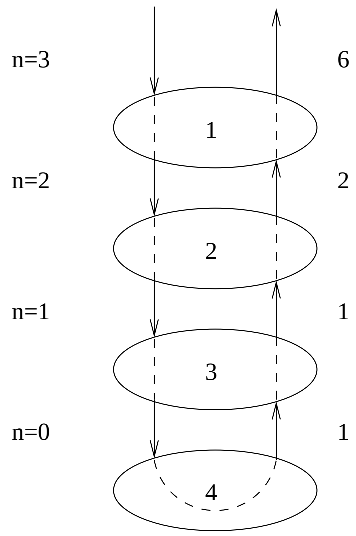


Figura 3.1: Arborele inversat pentru funcţia `fact`

Dacă sunt verificate cele de mai sus atunci rezultă că funcţia este corectă pentru orice n natural. Demostraţia se face prin inducţie după n.

1. Etapa de bază: Primul punct este satisfacut (pentru $n = 0$ factorialul este $1 = 0!$).

2. Etapa inductivă: Presupunem ca funcţia lucrează corect pentru un număr natural oarecare k (`(fact` k`) =` $k!$). Pentru $k + 1$ funcţia va întoarce (conform definiţiei factorialului) `(*` k `(fact` k`))`. Dar (fact k) este $k!$ (conform ipotezei inducţiei), rezultând astfel `(*` k $k!$`)` care este egal cu $(k + 1)!$. QED.

Pentru funcţii recursive mai complicate sunt mai multe cazuri, dar procedeul de demonstrare rămâne acelaşi.

3.4 Reguli pentru conceperea de algoritmi recursivi

Pentru a rezolva o problemă cu ajutorul recursivităţii trebuie avute în vedere două lucruri:

- trebuie găsit un mod de rezolvare a problemei în cazul general prin descompunerea ei într-un număr finit de probleme tot mai mici, similare;

- trebuie găsit modul de rezolvare a celei mai mici versiuni a problemei (numită şi cazul de bază), printr-un număr finit de operaţii.

De exemplu, în algoritmul recursiv pentru determinarea factorialului, la fiecare apel recursiv se determină factorialul dintr-un număr mai mic:

- În cazul general, factorialul unui număr este egal cu produsul dintre număr şi factorialul din număr mai puţin unu;

- În cazul de bază, factorialul din 0 este 1.

De aici rezultă o mulţime de reguli de programare în Lisp a funcţiilor recursive:

- se vor utiliza funcţii de control `cond`, `if`;

- clauzele recursive din `cond`, `if` vor fi precedate de clauze de ieşire;

  Dacă în definiţia matematică a unei funcţii recursive condiţiile de terminare se pun după apelul recursiv, în cazul unui program Lisp acestea trebuie puse înaintea apelului recursiv. Aşejarea greşită, definirea incorectă sau lipsa acestora conduce la ciclări infinite. De exemplu, funcţia următoare ce determină dacă un obiect aparţine la o listă, nu ia în calcul cazul în care obiectul nu aparţine listei şi conduce, în acest caz, la o ciclare infinită.

  ```
  (defun our-member (el l)
    (cond ((equal (carl l)) el)
          (t (our-member (cdr l))) ))
  ```

  Pentru a funcţiona corect trebuie inserat la început în `cond` clauza `((endp l) nil)`.

- apelurile recursive se vor face cu argumente mai simple (mai apropiate de satisfacerea condiţiilor de ieşire);

- clauzele în care se utilizează `car`, `cdr` trebuie să fie precedate de clauze ce sunt satisfăcute când argumentele sunt suficient de simple (de exemplu: lista vidă, atom etc).

3.4.1 Recursivitate simplă şi recursivitate dublă

Funcţiile recursive în Lisp pot fi *simplu recursive* (la fiecare apel crează o copie), *dublu recursive* (la fiecare apel crează două copii) sau combinaţii de cele două. Pentru a putea reprezenta grafic cum funcţionează apelul unei funcţii recursive vom folosi *tehnica arborilor inversaţi*. *Adâncimea de recursivitate* a unui algoritm recursiv este dată de numărul de nivele din cadrul arborelui inversat corespunzător.

În cazul în care este vorba de prelucrarea unei liste, recursivitatea simplă permite parcurgerea listei pe nivelul superficial (adâncimea de recursivitate fiind dată de lungimea listei), iar recursivitatea dublă permite parcurgerea listei în profunzime (adâncimea de recursivitate fiind dată de adâncimea listei si de numarul elementelor din subliste). Daca avem in vedere că listele în Lisp reprezintă arbori atunci recursivitatea simplă înseamnă parcurgerea recursivă numai a subarborelui drept, iar recursivitatea dublă înseamnă parcurgerea atât a subarborelui stâng, cât şi a subarborelui drept.

Numărul de elemente (pe nivelul superficial) ale unei liste se poate defini astfel:

$$\text{our\_length}(l) = \begin{cases} 1 + \text{our\_length}((cdrl)) & \text{altfel} \\ 0 & \text{dacă} \quad l = nil \end{cases}$$

Funcţia Lisp corespunzătoare este următoarea:

```
(defun our_length (l)
  (cond ((endp l) 0)
        (t (+ 1 (our_lenght (cdr l)))) ))
```

Numerele lui Fibonacci se calculează astfel:

$$fib(n) = \begin{cases} fib(n-1) * fib(n-2) & \text{dacă} \quad n > 1 \\ 1 & \text{dacă} \quad n = 0 \\ 1 & \text{dacă} \quad n = 1 \end{cases}$$

Funcţia Lisp corespunzătoare este o funcţie dublu recursivă:

```
(defun fib (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1)) (fib (- n 2)))) ))
```

În urma apelului `>(fib 4)` rezultatul este `5`, arborele inversat asociat fiind dat de figura 3.2:

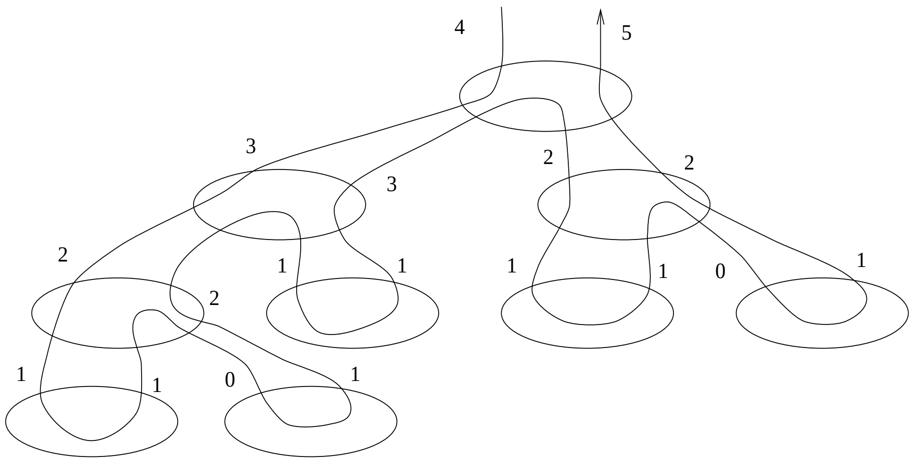


Figura 3.2: Arborele inversat pentru funcţia dublu-recursivă `fib`

Funcţia recursivă pentru atomizarea unei liste (obţinerea listei de atomi corespunzătoare unei liste oarecare) este un exemplu de funcţie ce combină recursivitatea dublă (pentru parcurgerea în adâncime a listei) cu recursivitatea simpla (pentru parcurgerea unei subliste pe nivelul superficial).

```
(defun atomizare (l)
  (cond ((endp l) nil)
        ; recursivitate simplă
        ((atom (car l)) (cons (car l)
                              (atomizare (cdr l))))
        ; recursivitate dublă
        (t (append (atomizare (car l))
                   (atomizare (cdr l)))) ))
```

În urma apelului `>(atomizare ((a b) c) d))` rezultatul întors este `(a b c d)`. Arborele inversat corespunzător este în figura 3.3.

3.5 Tipuri de funcţii recursive

3.5.1 Funcţii final recursive

O categorie intereresantă de funcţii recursive este dată de funcţiile final-recursive. O funcţie recursivă este *final-recursiva* dacă apelurile recursive nu sunt argumente pentru alte funcţii şi nu sunt utilizate ca şi teste. Altfel spus, o funcţie este final-recursivă dacă valoarea obţinută pe ultimul nivel de recursivitate

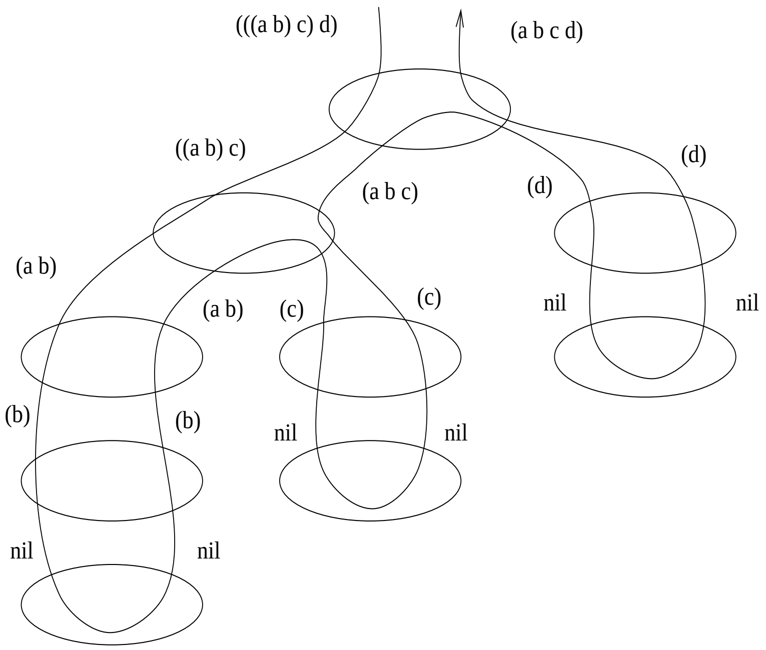


Figura 3.3: Arborele inversat pentru funcţia recursivă `atomizare`

rămâne neschimbată până la revenirea pe nivelul de sus. Mai exact, la ultima copie creată se obţine rezulatul, rezultat ce rămâne neschimbat la revenire.

Exemplele de mai sus reprezintă funcţii ce nu sunt final-recursive. La o funcţie ce nu este final recursivă se poate observa că apelul recursiv este conţinut într-un apel de funcţie (+, −, cons, append etc). Calculele respective rămân agăţate (neefectuate) pe fiecare nivel în coborâre, urmând a fi efectuate la revenire.

Pentru transformarea unei funcţii recursive într-o funcţie final-recursivă se foloseşte *tehnica variabilelor colectoare*.

Funcţia `fact` poate fi rescrisă astfel:

```
(defun fact (n)
  (fact-aux n 1))                    ; rez = 1

(defun fact-aux (n rez)              ; funcţie auxiliară
  (cond ((zerop n) rez)              ; rezultatul_final = rez
        (t (fact-aux (1- n)
                     (* n rez))) ))  ; rez = n*rez
```

Se observă că la coborâre se colectează rezultatele parţiale în `rez` urmând ca pe ultimul nivel să avem rezultatul final (arborele inversat corespunzător se găseşte în figura 3.4).

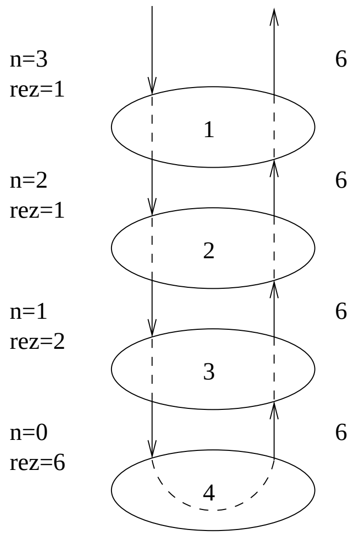


Figura 3.4: Arborele inversat pentru funcţia `fact` - varianta final recursivă

Varianta final-recursivă pentru determinarea celui de-al n-lea număr din şirul lui Fibonacci este următoarea:

```
(defun fib1 (n)
  (cond ((< n 2) 1)
        (t (fib1-aux
                    1     ; f1 - penultimul număr calculat
                    1     ; f2 - ultimul număr calculat
                    2     ; i  - indexul pentru numărul
                          ;    curent de calculat
                    n)) ))

(defun fib1-aux (f1 f2 i n)
  (cond ((> i n) f2)
        (t (fib1-aux f2       ; f2 -> f1
                    (+ f1 f2) ; (+ f1 f2) -> f2
                    (1+ i)    ; (1+ i) -> i
                    n)) ))
```

Dintr-o funcţie dublu recursivă (în varianta nefinal-recursivă) funcţia `fib` este transormată într-o funcţie simplu recursivă (în varianta final recursivă). Dacă în primul caz complexitatea este exponenţială (apelul `>(fib 100)` necesită mai mult de 10^{20} apeluri de funcţie, fiind practic imposibilă efectuarea acestui calcul pe un calculator), în al doilea caz complexitatea este polinomială (`>(fib1 100)` întoarce valoarea $\approx 5.73 * 10^{20}$, valoare ce se obţine după circa 100 de apeluri de funcţie).

O funcţie final-recursivă se bucură de proprietatea că poate fi tradusă automat într-o funcţie iterativă. Mediile Lisp realizează acest lucru prin existenţa unor opţiuni de optimizare la nivelul interpretorului/compilatorului Lisp.

3.5.2 Recursivitate compusă

În cazul în care în cadrul apelurilor recursive există apeluri recursive spunem că avem *recursivitate compusă.* Numărul de operaţii pentru aceste funcţii creşte icredibil de repede odată cu creşterea valorilor argumentelor. Un exemplu este funcţia lui Ackermann.

```
A<I, 0>=I+1
A<0,J>=A<1,J-1>
A<i,j>=A<A<I-1,J>,J-1>
```

Valorile si numărul operaţiilor cresc foarte repede: `A<0,1>=2, A<1,2>=5, A<2,3>=29, A<3,4>=`2^{65536}, ...

3.5.3 Recursivitate monotonă şi nemonotonă

Recursivitatea monotonă este recursivitatea în care modificarările asupra argumentelor din apelurile recursive se fac tot timpul în aceeaşi direcţie. Ca exemplu de funcţii recursive monotone avem: calculul factorialului, calculul lungimii unei liste, funcţia de atomizare a unei liste (argumentul fiind înlocuit cu `car`, respectiv cu `cdr` până la reducerea sa la un atom). Recursivitatea monotonă se mai numeşte şi *recursivitate structurală.*

Recursivitatea nemonotonă este cea în care modificările asupra argumentelor sunt nemonotone (nu tot timpul în aceeaşi direcţie). Spre exemplu funcţia recursivă ce implementează metoda lui Newton pentru găsirea zerourilor unei funcţii este nenonotonă.

Metoda lui Newton pentru găsirea unei soluţii pentru $f(x) = 0$ se bazează pe formula iterativă urmaţoare:

$$x_{k+1} = x_k - \frac{f(x_k)}{Df(x_k)}$$

În continuare este prezentat programul Lisp pentru $f(x) = x^3 - 1$:

```
(defun f (x)
     (- (* x x x) 1) )
(defun df (x)
     (* 3 x x) )
(defun newx (x)
     (- x (/ (f x) (df x))) )
(defun newton (x)
     (cond ((< (abs (f x)) 0.00001) x)
           (t (newton (newx x))) ))
```

3.6 Trasarea funcţiilor recursive

Pentru a putea observa mai bine evaluarea funcţiilor recursive, respectiv funcţionarea corectă a acestora limbajul Lisp permite utilizarea unor funcţii de depanare (trasare, evaluare pas cu pas, utilizarea unor puncte de intrerupere etc). Mai multe informaţii se găsesc în Anexa **??**. În continuare o sa ne referim pe scurt la funcţiile de trasare. Acestea sunt `trace` pentru activarea trasării şi `untrace` pentru dezactivarea trasării. Trasarea presupune afişarea argumentelor la fiecare apel (intrare în funcţie) şi a rezultatului la fiecare ieşire din funcţie. Argumentele din cele două funcţii de trasare sunt funcţiile pentru care dorim sa activăm / dezactivăm trasarea.

Rezultatul trasării funcţiei `fact` - varianta nefinal-recursiva este:

```
>(trace fact)         ; activare trasare
(FACT)
>(fact 3)
 0: (FACT 3)          ; -->
   1: (FACT 2)        ; -->
     2: (FACT 1)      ; -->
       3: (FACT 0)    ; -->
       3: returned 1  ; <--
     2: returned 1    ; <--
   1: returned 2      ; <--
 0: returned 6        ; <--
6
>(untrace fact)       ; dezactivare trasare
(FACT)
```

Trasarea funcţiei `fib` (varianta nefinal-recurivă şi varianta final recursivă):

```
>(trace fib fin1 fib1-aux)
```

```
(FIB FIB1 FIB1-AUX)
>; trasare variantă nefinal-recursivă
>; a se observa similitudinea cu arborele inversat
>; corespunzător
>(fib 4)
 0: (FIB 4)
   1: (FIB 3)
     2: (FIB 2)
       3: (FIB 1)
       3: returned 1
       3: (FIB 0)
       3: returned 1
     2: returned 2
     2: (FIB 1)
     2: returned 1
   1: returned 3
   1: (FIB 2)
     2: (FIB 1)
     2: returned 1
     2: (FIB 0)
     2: returned 1
   1: returned 2
 0: returned 5
5
>; trasare variantă final-recursivă
>(fib1 4)
 0: (FIB1 4)
   1: (FIB1-AUX 1 1 2 4)
     2: (FIB1-AUX 1 2 3 4)
       3: (FIB1-AUX 2 3 4 4)
         4: (FIB1-AUX 3 5 5 4)
         4: returned 5
       3: returned 5
     2: returned 5
   1: returned 5
 0: returned 5
5
>(untrace fib fib1 fib1-aux)
(FIB FIB1 FIB1-AUX)
```

Trasarea funcţiei `newton` (cu argumentul funcţiei ce nu se modifică în acelaşi sens):

```
>(trace newton)
(NEWTON)
>(newton -1.0)
 0: (NEWTON -1.0)
   1: (NEWTON -0.3333333)
     2: (NEWTON 2.7777781)
       3: (NEWTON 1.895052)
         4: (NEWTON 1.3561869)
           5: (NEWTON 1.0853586)
             6: (NEWTON 1.0065371)
               7: (NEWTON 1.0000424)
                 8: (NEWTON 1.0)
                 8: returned 1.0
               7: returned 1.0
             6: returned 1.0
           5: returned 1.0
         4: returned 1.0
       3: returned 1.0
     2: returned 1.0
   1: returned 1.0
 0: returned 1.0
1.0
>
```

3.7 Probleme

3.7.1

Problema 3.1 *Fiind date m şi n, două numere naturale, să se calculeze m^n. Să se scrie atât varinta nefinal-recursivă, cât şi cea final-recursivă.*

Problema 3.2 *Să se scrie o funcţie recursivă,* `aduna`*, pentru adunarea a două numere naturale fără a utiliza $+$. Se vor folosi funcţiile Lisp* `1+` *şi* `1-` *de incrementare, respectiv decrementare a unui număr cu* 1*.*

Problema 3.3 *Fiind date doua numere naturale a şi b, să se calculeze cel mai mare divizor comun.*

Problema 3.4 *Fiind date: a un numar real şi n un număr natural, să se calculeze a^n utilizând un număr minim de înmulţiri.*

3.7.2 Operaţii asupra listelor

Operaţiile asupra listelor pot avea loc pe nivelul superficial sau în adâncime. Spre exemplu, funcţia `member` acţionează doar pe nivelul superficial al unei liste, pe când funcţia `subst` pe orice nivel.

O lista poate fi privită ca un arbore binar, în care subarborele stâng este reprezentat de `car`-ul listei, iar subarborele drept de `cdr`-ul listei.

Exerciţiul 3.1 *Diferenţa între parcurgerea pe nivel superficial şi parcurgerea în adâncime rezultă din scrierea următoarelor funcţii:* `our-copy-list`*, ce crează o copie a unei liste (copierea doar a nivelului superficial) şi* `our-copy-tree`*, ce crează o copie a arborelui binar (copiere a listei pe toate nivelurile).*

```
(defun our-copy-list (l)
  (cond ((atom l) l)
        (t (cons (car l) (our-copy-list (cdr l))))))
(defun our-copy-tree (t)
  (cond ((atom l) l)
        (t (cons (our-copy-tree (car l))
                 (our-copy-tree (cdr l))))))
```

Problema 3.5 *Să se scrie funcţia* `our-reverse` *care inversează elementele unei liste pe nivelul superficial.*

Problema 3.6 *Să se scrie o funcţie ce determină primele n elemente dintr-o listă.*

Problema 3.7 *Să se scrie o funcţie ce determină ultimele n elemente dintr-o listă.*

Problema 3.8 *Fiind dată o listă l, să se scrie o funcţie recursivă care întoarce numărul de atomi din listă, indiferent de nivelul pe care se găsesc atomii.*

Problema 3.9 *Fiind dată o listă l, să se scrie o funcţie recursivă care determină adâncimea listei.*

Problema 3.10 *Să se scrie funcţia* `our-member` *care determină prezenţa unei expresii simbolice (în particular, a unui atom) într-o listă (indiferent de nivel).*

3.7.3 Operaţii cu mulţimi

O mulţime este o colecţie de elemente distincte, fiecare element fiind considerat membru al mulţimii. O mulţime poate fi reprezentată ca o listă: $(el_1, el_2, \ldots, el_n)$, unde $el_i, i = \overline{1, n}$, sunt atomi, ordinea acestora nefiind importantă.

Exerciţiul 3.2 *Pentru a determina daca o lista reprezină o mulţime vom scrie un predicat* `multimep`*:*

```
(defun multimep (l)
  (if (or (not (listp l)) (not (listp (cdr l))))
    nil ; sunt eliminaţi atomii şi perechile cu punct
    (cond
      ((endp l) t)
      ((and (atom (car l))
            (not (member (car l) (cdr l))))
                        (multimep (cdr l)))
      (t nil))
  ))
```

Problema 3.11 *Fiind date două mulţimi A şi B, să se scrie o funcţie recursivă ce determină reuniunea celor două mulţimi ($A \cup B$). Similar, să se scrie câte o funcţie pentru calculul intersecţiei ($A \cap B$), diferenţei ($A \setminus B$) şi diferenţei simetrice ($A \triangle B$).*

Problema 3.12 *Fiind date două mulţimi A şi B, să se scrie un predicat ce verifică dacă $A \subseteq B$ şi un predicat ce verifică egalitatea celor două mulţimi ($A = B$).*

3.7.4 Operaţii cu vectori şi matrice rare

Stocarea tablourilor (vectori, matrice etc) ridică probleme atunci când dimensiunile acestora sunt foarte mari. Un caz aparte de tablouri este cel în care un procent ridicat de elemente este 0. În acest caz vorbim de tablouri rare (sparse arrays). Pentru efectuarea de operaţii cu astfel de tablouri este suficient de a reţine doar elementele diferite de zero.

Pentru reprezentarea vectorilor rari şi a matricelor rare vom folosi listele.

Operaţii cu vectori rari

Un vector rar va fi reprezentat printr-o lista de subliste, fiecare sublistă (componentă) fiind formată din două elemente: un index şi valoarea corespunzătoare: $((< index_1, < val_1 >), \ldots, (< index_n >, < val_n >))$. De exemplu, vectorul #(1.0, 0, 0, 0, 0, 0, -2.0) va fi reprezentat prin ((1 1.0) (7 -2.0)).

Pentru accesul la indexul şi valoare am folosit două funcţii `index` şi `val`.

```
(defun comp (vector)
  (car vector)) ; extrage componenta
(defun rest-comp (vector)
  (cdr vector)) ; rest componente
(defun index (comp)
  (car comp))  ; extrage indexul din perechea
               ; (<index>, <valoare>)
(defun val (comp)
  (cadr comp)) ; extrage valoarea din perechea
               ; (<index>, <valoare>)
```

Exerciţiul 3.3 *Scriem în continuare o funcţie* `prod-vect-const` *ce calculează produsul dintre un vector rar şi un scalar.*

```
(defun prod-vect-const (V s)
  (cond ((zerop s) nil)  ; caz special
        ((endp V) nil)   ; cond. terminare
        (t (cons (list (index (comp V))
                       (* s (val (comp V))))
                 (prod-vect-const (rest-comp V) s)))))
```

Exerciţiul 3.4 *Scriem în continuare o funcţie* `suma-vect` *ce calculează suma a doi vectori rari U şi V.*

```
(defun suma-vect (U V)
  (cond ((endp U) V)
        ((endp V) U)
        ((< (index (comp U)) (index (comp V)))
          (cons (comp U) (suma-vect (rest-comp U) V)))
        ((> (index (comp U)) (index (comp V)))
          (cons (comp V) (suma-vect U (rest-comp V))))
        (t (cons (list (index (comp U))
                (+ (val (comp U)) (val (comp V))))
                   (suma-vect (rest-comp U) (rest-comp V))))))
```

Problema 3.13 *Să se scrie o funcţie* `prod-vect-scalar` *ce calculează produsul scalar a doi vectori rari U şi V.*

Operaţii cu matrice rare

O matrice rară poate fi reprezentată ca o lista de subliste în care fiecare sublista are forma: $(< nr - linie > ((< index_1, < val_1 >), \ldots, (< index_n >, < val_n >)))$ (de exemplu: ((1 ((1 1.0) (5 -2.0))) (3 ((3 2.0) (11 -1.0))))).

Pentru a avea acces la elementele matricei rare vom folosi următoarele funcţii:

```
(defun linie (matrice)
    (car matrice)) ; extrage linie
(defun rest-linii (matrice)
    (cdr matrice)) ; rest linii
(defun nr-lin (linie)
    (car linie)) ; extrage index linie
(defun comp-lin (linie)
    (cadr linie)) ; extrage lista componente linie
```

Exerciţiul 3.5 *În continuare scriem o funcţie (*`suma-m`*) care calculează suma a două matrice rare.*

```
(defun suma-m (A B)
 (cond ((endp A) B)
       ((endp B) A)
       ((< (nr-lin (linie A)) (nr-lin (linie B)))
         (cons (linie A) (suma-m (rest-linii A) B)))
       ((> (nr-lin (linie A)) (nr-lin (linie B)))
         (cons (linie B) (suma-m A (rest-linii B))))
       (t (cons (list (nr-lin (linie A))
       (suma-vect (comp-lin (linie A)) (comp-lin (linie B))))
          (suma-m (rest-linii A) (rest-linii B))))))
```

Problema 3.14 *Să se scrie o funcţie (*`transp-m`*) care calculează transpusa unei matrice rare.*

Problema 3.15 *Să se scrie o funcţie (*`prod-v-m`*) ce calculează produsul dintre un vector rar V si o matrice rară A.*

1. *Produsul dintre un vector rar şi o matrice rară poate fi obţinut ca o listă a produselor scalare ale vectorului cu fiecare coloana a matricei. Aceasta necesită transpunerea matricei înainte de efectuarea calculelor.*

2. *O altă varianta este de a parcurge fiecare linie a matricei A la care îi corespunde câte un element din vectorul V şi de a depune în vectorul rezultat sumele parţiale corespunzătoare produsului scalar.*

Problema 3.16 *Să se scrie o fucţie (*`prod-m`*) ce calculează produsul a două matrice rare A şi B. Un element (i, j) al matricei rezultat este dat de produsul scalar dintre linia i a matricei A şi coloana j a matricei B. Se vor aborda mai multe variante bazându-ne pe observaţiile de la 3.15 şi pe funcţiile anterioare.*

Problema 3.17 *Modificaţi programele de mai sus astfel încât să fie eliminate componentele cu valori zero din rezultat. Adăugaţi fucţii ce permit citirea, respectiv afişarea unor tablouri rare.*

Proiectul 3.1 *Extindeţi mulţimea de funcţii de mai sus pentru a opera cu tablouri cu mai multe dimensiuni.*