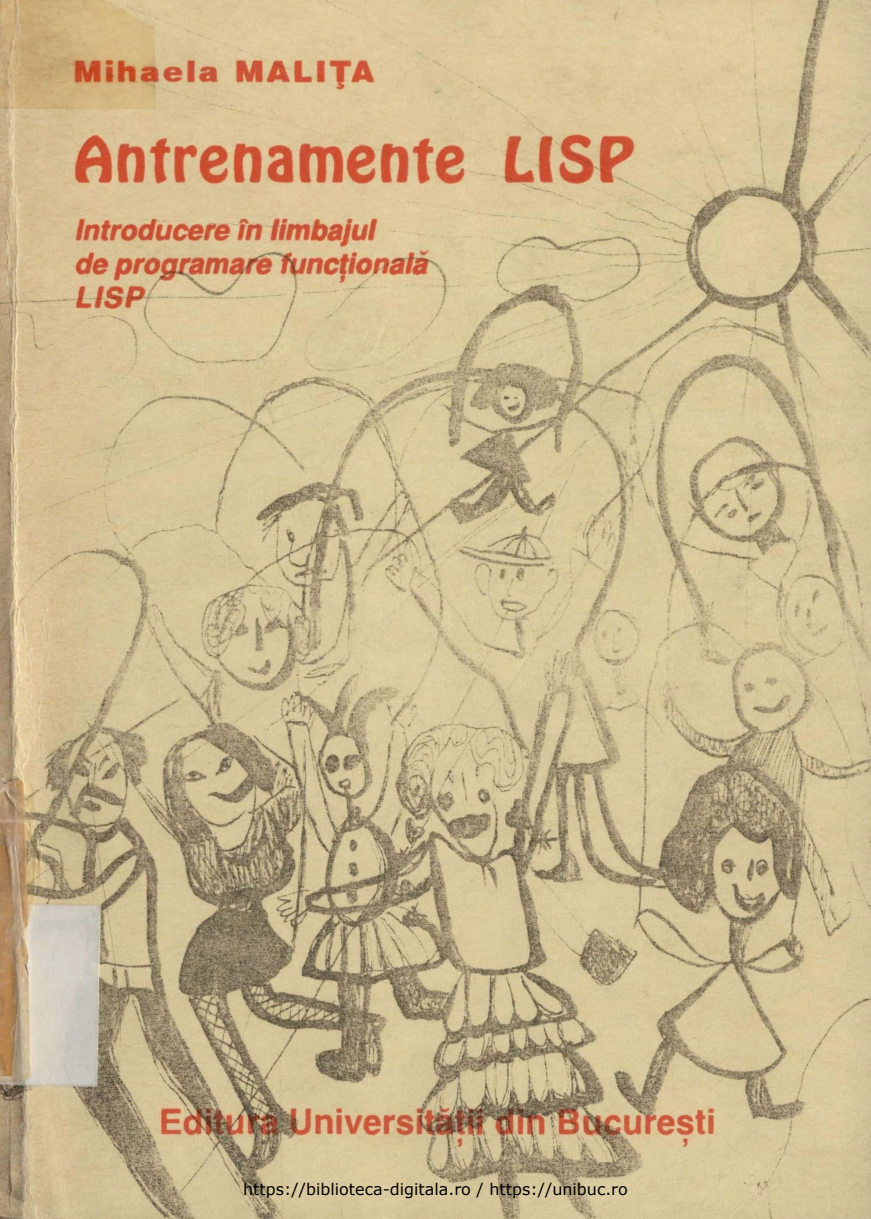
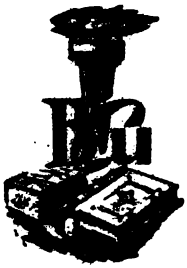
Mihaela MALIŢA

Antrenamente LISP

Introducere în limbajul de programare funcțională LISP



Editura Universității din București



Mihaela Maliţa

*

Antrenamente LISP

Mihaela Malița

Antrenamente LISP

Introducere în limbajul de programare funcțională: LISP

EDITURA UNIVERSITĂȚII DIN BUCUREȘTI
- 1998 -

Coperta: Iuliana Mihuț

Tehnoredactare: Mihaela Malița
Florian Mihalcea

Șos. Panduri, 90-92, București - 76235; Telefon/Fax 410.23.84

ISBN - 973 - 575 - 204 - 2

Cuprins

Partea I. Introducere în LISP

1. Prima lecţie 15
• Exerciţii
2. Despre Lisp 21
Povestea LISP-ului • Ce scriem în LISP • LISP caracteristici • Dialecte LISP
3. Lumea listelor 25
Atomul • Perechi cu punct • Lista • S-expresii • Arbori binari • Celula LISP • Scrierea prefixată
4. LISP-ul pur 31
Adevăr şi fals • Predicatul • Atribuirea în LISP • Funcţiile Lisp-ului pur • Exerciţii
5. Aritmetica 47
Principalele funcţii aritmetice • Funcţii trigonometrice • Numere aleatoare • Cel mai mic divizor comun • Exerciţii
6. Recursivitatea 53
Ultimul element • Factorialul • Şirul lui Fibonacci • Funcţia lui Ackermann • Tabelul unei funcţii • Exerciţii
7. Principalele funcţii pe liste 61
Backquote • Funcţii logice • Lipirea listelor • Apartenenţa • Lista ca vector • Lungimea listei • Listele de asociere • Substituţii • Ştergerea • Inversarea listei • Exerciţii

8. Totul despre funcţii 69
Notaţia LAMBDA • LET şi LABELS •Tipuri de funcţii • MACRO-uri • Funcţii chirurgicale • Exerciţii
9. Iteraţia 79
Structura DO • Structura PROG • MAP-ări • Problema FUNARG • Exerciţii
10. Liste de proprietăţi 91
Lucru cu proprietăţi pe atom • Intersecţia prin liste de proprietăţi • Bridge • Exerciţii
11. Stringuri (şiruri de caractere) 95
Stringul şi alte tipuri de date • Comparări de stringuri • Căutare în string • Exerciţii
12. Mulţimi şi combinatorică 101
Operaţii frecvente pe liste •Mulţimile ca liste • Combinatorică • Şabloane pe listă • Exerciţii
13. Intrări şi ieşiri 115
Tipuri de date • Lucru cu fişiere • Scriere • Citire • Compar fişiere • Introducerea datelor • Cum organizăm un menu • Exerciţii

Partea a II-a. Antrenamente în LISP

14. Melania învaţă cu calculatorul 131
Melania învaţă tabla înmulţirii • Test *Tabla Inmulţirii* pentru Melania • Exerciţii
15. Derivarea 135
Derivarea funcţiilor elementare • Exerciţii
16. Mulţimi fuzzy 137
Noţiuni generale • Operaţii cu mulţimi fuzzy • Exerciţii
17. Grafuri 145
Noţiuni generale •Nodurile grafului • Drum într-un graf • Exerciţii

18. Arbori 151
Generalităţi • Parcurgerea arborilor • Arbori binari • Exerciţii
19. Sortare 159
Sortare prin inserare • Sortare prin interclasare • Sortarea rapidă • Sortare prin inserare în arbore • Metoda Bubble Sort • Exerciţii
20. Logică 167
Calculul cu propoziţii (Cp) • Formulă bine formată *fbf* • Valoarea unei formule *fbf* • Trecerea unei *fbf* în *fnc* • Principiul rezoluţiei • Exerciţii
21. Automate celulare. Jocul vieţii 185
Exerciţii
23. Algoritmi genetici 191
Simularea ruletei (metoda Goldberg) • Exerciţii
Bibliografie 203
Anexă. *Funcţiile din carte* 207

Stimaţi cititori,

Am numit cartea **Antrenamente LISP** pentru că ideea mea despre învăţarea unui limbaj de programare seamănă cu învăţarea unui sport. Degeaba citeşti sau îţi explică cineva dacă nu exersezi **singur.** De aceea aş vrea să priviţi cartea ca un îndrumar pentru antrenamentele dumneavoastră de LISP.

Deci înafară de această carte aţi avea nevoie de un calculator şi unul dintre dialectele LISP pus pe el. Vă sfătuiesc să aveţi în faţă şi documentaţia de la firmă a dialectului cu care lucraţi. În această carte sunt prezentate în general lucrurile de principiu şi pot exista diferenţe între dialecte mai ales în ceea ce privesc intrările şi ieşirile sistemului.

Am conceput redactarea cărţii nu numai pentru studenţii mei de la cursurile Inteligenţă Artificială de la Facultatea de Matematică, Universitatea Bucureşti ci şi pentru oricine doreşte să înveţe LISP.

A învăţa LISP **nu** înseamnă: *încă un limbaj de programare şi la urmei urmei ce ne mai trebuie*? Limbajul LISP este altfel decît BASIC, PASCAL, C etc şi total diferit de PROLOG.

LISP-ul face parte din clasa limbajelor **funcţionale** din care nu fac parte limbajele amintite mai înainte. În LISP un program este o compunere de funcţii. Funcţia este unitatea de bază şi singura.

Deşi este un limbaj mai puţin "industrial", limbajul LISP se învaţă în toate centrele universitare din lume.

În prima parte a cărţii am prezentat limbajul, iar în partea a doua am dat exemple de programe în LISP din domeniile:

învăţarea cu calculatorul, mulţimi vagi (fuzzy), grafuri, calculul cu propoziţii, algoritmi genetici, automate celulare. Am încercat să prezint în aşa fel aplicaţiile încît cititorul să nu trebuiască să cunoască dinainte noţiunile.

Nimic nu se face fără ajutor. Cel mai mult m-au ajutat studenţii mei care prin întrebările lor repetate m-au făcut să înţeleg că uneori profesorii nu înţeleg ce nu înţeleg studenţii. Mulţumiri speciale aş dori să-i aduc lui Viorica Sofronie pentru colaborarea la capitolele 17-19.

Aş aprecia foarte mult dacă mi-aţi sesiza greşelile şi mi-aţi trimite observaţiile dumneavoastră ca să scutim de suferinţă pe cititorii unei posibile următoare ediţii. Mult succes,

Mihaela Maliţa *E-mail:* malita@sunu.rnc.ro

Melaniei,

Partea I

Introducere în LISP

1. Prima lecţie

Ştiu că sunteţi nerăbdători să scrieţi un program în LISP. Aveţi în faţă un calculator care are undeva un director LISP şi un executabil LISP (lisp.exe). Vă situaţi în directorul respectiv şi tastaţi (cu litere îngroşate ceea ce tastăm noi):

>lisp

- se lansează interpretorul LISP -

- ne punem prima problemă: *cît face 1 + 1 ?*

:(+ 1 1) <ENTER>

2

- vom apăsa tasta <ENTER> de fiecare dată, *noi nu o vom mai scrie!*

- este anunţul nostru că dăm controlul sistemului -

- am prins curaj, putem să încercăm lucruri mai grele -

:(+ 1 2 3)

6

:(+ 1 (* 2 3))

7

:(+ (* 2 2) (* 3 3) (* 4 4))

29

:13

13

Ce am observat pînă acum:

- îndată ce scriem, primim şi răspunsul (efect al interpretorului)
- punem totul între paranteze

- scriem altfel ca în matematică, semnul de operație este la început !
- dacă tastăm un număr, sistemul răspunde ca un *ecou*

:(DEFUN DUBLU (N) (+ N N))
DUBLU
:(DUBLU 5)
10
:(DEFUN SALUT ( ) (PRINT 'Buna!))
SALUT
:(SALUT)
BUNA!
: (EXIT)
- IESIRE DIN LISP -

Ce am observat:
- am putut defini chiar noi o funcție: DUBLU(N)
- cu DEFUN am definit și funcția SALUT, fără argumente
- indiferent cum scriem noi sistemul răspunde cu litere mari
- a apărut un semn ciudat în fața lui 'Buna!, se numește *QUOTE*
- am ieșit din sistem cu (EXIT)

Să continuăm sesiunea de LISP.
:A
eroare cine este A?
:'A
A
:'(+ 1 2 3)
(1 2 3)
:(1 2 3)
eroare: 1 nu este o funcție
:'(ASTA ESTE O LISTA)
(ASTA ESTE O LISTA)
:'(SI (ASTA) ESTE O LISTA)
(SI (ASTA) ESTE O LISTA)
:(o lista cu 4 elemente)

eroare: pe prima poziție nu este un semn de operatie, nu stiu ce să fac
:(EQUAL 3 4)
NIL
:(> 8 3)
T
:'(EQUAL 3 4)
(EQUAL 3 4)

Ce am observat:

- virgula de sus, ', (QUOTE) inhibă calculul. Ca și apostroful din limba curentă, parcă ar spune: ***nu cerceta, lasă-l așa cum e***!
- unele liste pot fi calculate, altele nu. Dacă au un semn de operație în față interpretorul LISP întoarce ceva: spunem că *evaluează lista*
- dacă întrebăm *3 este egal cu 4?* Răspunsul este **T,** care vine de la **TRUE** (*adevărat* în limba engleză)
- dacă întrebăm *8 este mai mare ca 3* răspunsul este **NIL** (*fals*)

Să intrăm înapoi în LISP, lansăm deci din nou interpretorul. Acum să scriem un mic program care să ne ceară un număr de la terminal și să întoarcă numărul nostru adunat cu 1.

>lisp
- am apelat interpretorul LISP -
:(DEFUN ADUN-CU-1 () (PRINT (+ 1 (READ)))
ADUN-CU-1
:(adun-cu-1)
5
6
:(DUBLU 3)
eroare: nu știu cine e DUBLU

Ce am observat:

- nu are importanță dacă scriem cu litere mari sau mici
- programul este o listă
- totul este între paranteze, adică este o listă
- scriu puțin pe dos, în loc să scriu: *citește, adună cu 1, tipărește.*

Am scris: ***tipăreşte, după ce ai adunat cu 1, ceea ce ai citit.***

- seamănă cu compunere de funcţii: f (g (h (x))). Intîi calculez pe h(x) apoi pe g (h (x)) apoi f (g (h (x))).
- dacă am ieşit din sistem, funcţiile definite cu o sesiune mai înainte nu se reţin, trebuie să le scriem într-un fişier

Să punem în fişierul *LECTIE* din directorul unde se află LISP-ul cele trei funcţii pe care le-am încercat mai înainte:

- Fisierul LECTIE -

```
; Acesta este un comentariu în LISP
; tot ce este după punct şi virgulă, nu se calculează
(DEFUN DUBLU ( N)     (+ N N))
;
(DEFUN        SALUT   ( )
        (PRINT     'Buna! ) )
;
(DEFUN ADUN-CU-1 () ; aduna 1 la numarul citit
        (PRINT (+ 1 (READ))
)
```

Acum avem salvat tot ce am lucrat, adică toate funcţiile scrise de noi. Să lansăm interpretorul LISP şi să încărcăm fişierul LECTIE.
În limba engleză a încărca se spune *LOAD*. Avem secvenţa

```
>lisp
:(LOAD "LECTIE" )
T
:(DUBLU 5)
10
:(SALUT)
BUNA!
:(ADUN-CU-1)
4
5
:(exit)
>Inapoi în sistemul de operare
```

Ceea ce am observat:

- funcţiile definite de noi se scriu într-un fişier
- nu are importanţă cum aşezăm liniile în fişier
- nu au importanţă numărul de spaţiile goale (blanck-uri) între paranteze şi cuvinte sau între cuvinte
- fişierul se încarcă în mediul LISP cu *LOAD*
- liniile de comentariu încep cu ;
- indiferent ce scriem primim un răspuns: este tipic pentru un interpretor (LISP) de a răspunde întotdeauna

Să modificăm funcţia (ADUN-CU-1) astfel încît să-mi tot ceară un număr de la consolă pînă cînd îi spun eu să se oprească.
Intru în fişierul LECTIE îi modific cu orice editor am la îndemînă, numai funcţia ADUN-CU-1.

```
(DEFUN ADUN-CU-1 ()
(PRINT (+ 1 (READ))
(PRINT 'CONTINUAM? )
(IF (EQUAL 'DA (READ)) (ADUN-CU-1))
)
```

Să încărcăm funcţia în mediul nostru de LISP şi s-o apelăm.

```
>lisp
:(LOAD "LECTIE")
T
:(ADUN-CU-1)
5
6
CONTINUAM?
DA
8
9
CONTINUAM?
NU
:(EXIT)
- IESIRE din LISP -
```

Exerciţii. 1. Prima lecţie

1. Exploraţi toate fişierele aflate în directorul unde se află LISP-ul.
2. Faceţi exerciţiul de a edita un fişier, de a-l încărca, de a lucra cu funcţiile definite şi apoi de a ieşi din sistem.
3. Faceţi o listă cu toate cărţile de LISP existente în bibliotecile accesibile dumneavoastră. Răsfoiţi-le.
4. Întrebaţi şi aflaţi care sunt dialectele de LISP/SCHEME care există pe calculatoarele din anturajul dumneavoastră.

2. Despre LISP

2.1. Povestea LISP - ului

LISP vine de la *LISt Processing language* (limbaj pentru prelucrarea listelor).
În 1956 în urma unei întîlniri de lucru de la Dartmouth (S.U.A.) între specialiştii din calculatoare s-a propus crearea unui limbaj de manipulare a listelor. Dominată de spiritul marelui savant John McCarthy, tot atunci s-au pus şi bazele unei noi ramuri care a fost botezată pe loc *Artificial Intelligence* (Inteligenţa Artificială).
În 1958, John McCarthy a propus prima versiune a limbajului LISP, inspirat din lambda-calcul, inventat în 1941 de către matematicianul englez Alonzo Church.
Multă vreme limbajul de programare LISP a fost folosit doar în centre universitare şi cercetare, piaţa fiind invadată de limbajele orientate spre prelucrări numerice. Odată cu impunerea inteligenţei artificiale s-a impus şi limbajul LISP.

2.2. Ce scriem în LISP

Gîndit ca limbaj de prelucrare simbolică limbajul de programare LISP a fost folosit în multe aplicaţii de inteligenţă artificială. Enumerăm cîteva domenii:

- matematică simbolică

- demonstrare automată
- sisteme expert
- învăţare automată
- limbaj natural
- robotică
- educaţie (CAI = Computer Aided Instruction)
- design (CAD = Computer Aided Design)
- rezolvarea problemelor (problem solving)
- lingvistică computaţională

Dăm mai jos numele unor pachete de programe celebre scrise în limbajul LISP / 40 /:

LUNAR (Wood)	SHRDLU (Winograd)	MACSYMA
DENDRAL	MYCIN	AM (D. Lenat)
DEDU	INTERNIST	CONNIVER
FUZZY	PLANNER	POP-2

2.3. LISP - caracteristici

Dăm pe scurt cîteva caracteristici principale ale limbajului LISP:

- specializat în prelucrări simbolice
- gestiune automată a alocării memoriei
- exemplu de limbaj de programare funcţională
- utilizarea lambda-expresiilor pentru funcţii
- aceeaşi reprezentare a datelor şi programelor
- structura de date de bază: *lista*
- funcţia EVAL (evaluarea) serveşte deopotrivă ca definiţie formală a limbajului şi ca interpretor
- admite un nucleu LISP-pur
- recursivitatea tehnică principală

- extensibilitate
- universalitate
- interactivitate
- consum mare de memorie

2.4. Dialecte LISP

Încă de la început limbajul LISP nu a avut un standard. Fiecare implementare de 50 de ani încoace, îşi impune amprenta autorului, aduce îmbunătăţiri şi soluţii mai bune de implementare. Fiecare versiune îşi are propriile funcţii şi uneori aceeaşi funcţie este denumită în mod diferit în implementări diferite.

Abia în 1984 a apărut prima variantă acceptată de standardizare sub forma limbajului COMMON LISP, ceea ce nu a dus deloc la dispariţia celorlalte dialecte. Dintre dialectele mai vechi şi mai răspîndite amintim:

MACLISP	INTERLISP	ZETALISP	FranzLISP
LISP1.5	UCILISP	StanfordLISP	MuLISP
LeLISP	BYSO-LISP	XLISP	SCHEME
IQ LISP	LISP-80	LISP-86	CLISP
Allegro LISP	MuLISP	CommonLISP	TRANSLISP
Golden Common LISP		KYOTO-CLISP	

De curînd s-a impus un nou dialect de LISP, numit **SCHEME** /12/, /17/.

Dacă COMMON LISP concurează la cel mai mare număr de pagini de specificaţie, SCHEME îşi găseşte locul printre limbajele cu cele mai scurte specificaţii. Dialectul de LISP ales pentru a ilustra exerciţiile din cartea de faţă este CLISP. Implementarea pe calculatoarele personale cea mai apropiată de standardul LISP-ului: COMMON LISP este dialectul GOLDEN COMMON LISP care are deja o foarte largă răspîndire. Multe din programele cărţii de faţă sunt scrise în TRANSLISP. Avem convingerea că pot fi uşor transportate în orice dialect din familia Common LISP.

3. Lumea LISTELOR

3.1. Atomul

Atomii sunt cărămizile principale care formează listele. Atomii se împart în două categorii:

- atomi **numerici** (numere: 5, 6.7 , 3.14, 7/8, -100.4 etc)
- atomi **simbolici** (cucu, variabila1, $X, a23, a-b, -+, da?, nu! etc)

Numerele sunt considerate ca fiind atomi numerici, şi pot fi, la rîndul lor, de mai multe tipuri: ***întregi, reale***, etc., depinzînd de tipurile de numere implementate în dialectul LISP respectiv.

Atomii simbolici (sau denumiţi uneori atomi literali) pot fi formaţi din orice combinaţie de caractere, oricît de lungă, dar în care să nu apară caracterele nepermisibile:

( ) \ ` , " ; blanc

Exemple. Următoarele combinaţii de caractere formează **atomi simbolici**:

$functie-utilizator$	#eticheta#	A	???
I.L.CARAGIALE	*VAR-GLOBALA*	este-PREDICAT:	=multimi?
an1991	+dec31	punct.	fals
intre-bare/	A1b2c3!	**	+2 120.0

Exerciţiu. De ce următoarele expresii nu sunt atomi ?

"a 12	bcd)	a"9	a . bine	a ,tom	)b12
1 mai	(martie)	an='89	;comentariu	`citare	(a23)

3.2. Perechi cu punct

O pereche cu punct constă dintr-o paranteză deschisă urmată de o s-expresie, blanc, punct, blanc şi o altă s-expresie şi paranteză dreaptă. S-expresia este fie un atom fie o pereche cu punct. Cu alte cuvinte avem o pereche cu punct (p1 . p2) unde p1, p2 sunt fie atomi fie perechi cu punct. Se observă că orice pereche cu punct are o parte stîngă şi una dreaptă.
În acest mod se pot identifica perechile cu punct ca arbori binari. (vezi ARBORI BINARI)

Exemple. Dăm mai jos cîteva perechi cu punct:

(a . (b . c)) (a . (b .(3 . nil)) ((a . nil) . nil)
((a . 1) . (c . 2)) (((a . b) . z) . (s . b)) (a . b)

Exerciţii. Scrieţi sub formă de arbori binari următoarele perechi cu punct:

((a . nil) . (b . c)) ((a . b) . (c .(d . nil))) (a . (b . nil))

3.3. Lista

O listă este un obiect de forma (p1 ... pn), unde *pi* sunt atomi sau liste. Se defineşte în plus lista vidă () echivalentă cu atomul rezervat NIL. O listă în particular este o pereche cu punct care are întotdeauna în partea dreaptă o pereche cu punct sau atomul NIL.

Exemple. Acestea sunt liste:

(( )) (acum (ori) niciodata) ((ieri azi)(x (5))) ((a)(b)(c))
(aveti ()) (+ 3 2 4) (+ 2 (+ 3 3)) (* (+ (- 2) 3)(+ 4 5)) (A ((B)))

Exerciţii.

1) De ce următoarele obiecte **nu** sunt liste ?:

(a (b c) (+ 1 . 2) (2 . 3) (abracadabra11" "(l m)
(nil . a) (i) o)) 57

2) Completaţi scrierea următoarele liste ca perechi cu punct:

(a b) → (a . (b)) → (a . (b . nil))
(a b c) → (a . (b c)) →
((a) b) → ((a . nil) . (b .nil))
((a b) c) →
((a) (b)) → ((a . nil) .((b . nil) . nil))

(()) → (nil) →
(a (b)) → (a . ((b . nil) . nil))
((a b)) →

LISP-ul transformă toate listele în perechi cu punct cu care apoi lucrează. Deci structura fundamentală a LISP-ului este ***perechea cu punct*** (vom vedea echivalența pereche cu punct ↔ arbore binar).
Avantajele oferite de structura de listă:

- listele pot fi oricît de lungi (teoretic)
- nu sunt necesare declarații de lungime de listă
- listele pot conține obiecte de natură variată, care nu trebuie declarată, exemplu: (= 3 (+ a 1))
- listele sunt o reprezentare liniară a ideii de nivel, exemplu:
  (BUNIC (TATA (FIU)) (MAMA (FIU FIICA)))
- listele pot reprezenta imediat ideea de asociere, exemplu:
  ((unu 1) (doi 2) (trei 3))
- listele pot fi în același timp și programe și date, exemplu: (+ 2 4 5)

Dezavantajele oferite de liste:

- programatorul trebuie să închidă corect parantezele!

3.4. S-expresii

S-expresiile sunt obiectele de bază ale limbajului LISP, ele incluzînd atomii, perechile cu punct și listele. Reamintim că ***orice listă se poate scrie ca o pereche cu punct***, reciproca nefiind adevărată.
Iată gramatică Chomsky pentru definirea perechilor cu punct:
S→ (P . P) P → ATOM P → (P . P)

Exemple de ***s-expresii***: atom-ul A12 (p x y +) ((a . b) . c)
Exerciții. Arătați că următoarele obiecte **nu** sunt s-expresii:
((a . b . b) ((a . b) . n .m)) (a .) (a . b . b) (a ..) .a
Comentarii

- In unele dialecte LISP **melania.** este considerat ***atom***, sau **ACUM.** este ***atom***. Se observă că: **(A .)** este diferit de **(A.)** Deci neapărat

punctul trebuie să aibă blanc în față, altfel este considerat ca parte componentă a atomului.

- Deși în programarea începătorilor se pot evita perchile cu punct lucrîndu-se numai cu liste, pentru programe serioase se observă o importantă economie de memorie cînd se folosesc perechile cu punct (acolo unde se poate, de exemplu la liste de asociere).

3.5. Arbori binari

Arborii binari sunt acei arbori care au întotdeauna 2 ramuri din fiecare nod.

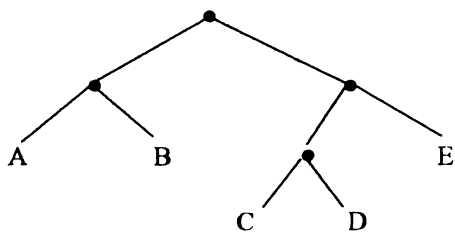


Fig. 3.1. Arborele corespunde listei ((A . B) . ((C . D) . E))

Arborele de mai sus se scrie astfel ca o pereche cu punct. Se marchează numai frunzele.

Se amintește echivalența ***arbore binar = pereche cu punct.***

Exerciții. Scrieți sub forma de arbori binari următoarele S-expresii (pentru ușurință se transformă mai înainte în perechi cu punct):

((A . B) (C . D)) → ((A . B). ((C . D) . NIL))

((A B) (C)) → ((A. (B . NIL)) . (C . (D . NIL)))

((A B) . (C D)) → ((A . (B . NIL)) . (C . (D . NIL)))

((A B) (C . D)) → ((A .(B . NIL)). ((C . D) . NIL))

((A B) . (C . D)) → ((A . (B . NIL)) . (C . D))

3.6. Celula LISP

Celula LISP este un alt mod de a reprezenta perechile cu punct. Scrierea cu celule LISP provine din reprezentarea perechilor cu punct în memoria calculatorului. Astfel partea stîngă şi partea dreaptă sunt înscrise în cîte un byte (de exemplu). Iată reprezentarea internă a listei ((a . b) . ((c . d) . e))

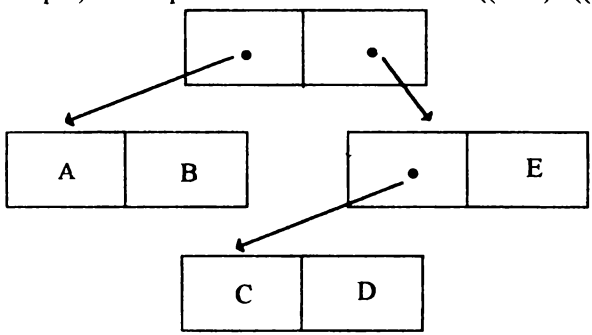


Fig. 3.2. Arborele binar din Fig.3.1 scris cu celule LISP

Uneori se mai foloseşte notaţia

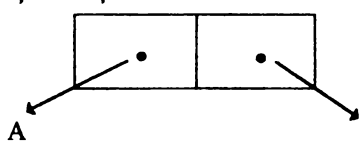


Fig. 3.3. Notaţie pentru (A . B)

care desemnează faptul că atît în stînga cît şi în dreapta sunt adrese, care în final pot fi adresele unor atomi A şi B, vezi /4/.

3.7. Scrierea prefixată

Forma de scriere a expresiilor aritmetice cu care suntem obişnuiţi este cea **infixată (inordine).** Aici operatorul este înăuntru, între operanzi, pe cînd la scrierea **prefixată (preordine)** va fi mereu pe primul loc. Un fapt esenţial al limbajului LISP îl constituie faptul că expresiile se scriu în forma prefixată sau preordine.

(* (+ x y) (- z 5)) este în *prefix*
(x + y) * (z - 5) este în *infix*
Scrierea folosită în matematică a compunerilor de funcţii va deveni:

Scrierea uzuală	**Scrierea în LISP**
f (x, y)	(f x y)
f (x, g(y, z))	(f x (g y z))
f (g(a,b), g(x,y))	(f (g a b) (g x y))
2sin x cox y	(* 2 (sin x) (cos y))

Esenţial în LISP este că dacă avem o listă program (supusă evaluării), atunci obligatoriu va avea pe primul loc al fiecărei subliste numele unei funcţii.

Exerciţii. Transformaţi în liste LISP următoarele expresii:

Scrierea uzuală	**In LISP**
f(x+1,y)	
f(x) + g(y)	(+ (f x)(g y))
a2 + 2ab + b2	
g(f1(x), f2(y),f3(z))	
(-b + (radical (b2 - 4 ac))) / 2a	
x+ y2/5	(+ x (/ (* y 2) 5))
f(x, g(y), z)	
sin x + cos y	
g(f(x), 5)	(g (f x) 5)

4. LISP-ul Pur

LISP-ul pur este nucleul de la care pornesc toate dialectele LISP. LISP-ul pur este un model de calculabilitate în sine. LISP-ul pur se referă doar la tipurile de date: atomi simbolici şi perechi cu punct (implicit şi liste). Se demonstrează că întreg modelul de calculabilitate al LISP-ului pur se construieşte cu următoarele 8 funcţii:

QUOTE EVAL ATOM CAR CDR CONS EQ COND

Astfel toate funcţiile referitor la liste pot fi scrise prin compuneri ale acestor funcţii de bază. Vom prezenta în această carte, pe cît este cu putinţă, funcţiile sistem cu ajutorul funcţiilor de bază, Urmărim astfel să construim în mod natural întregul limbaj LISP şi să convingem cititorul de puterea LISP- ului pur. Ne integrăm astfel în spiritul cărţii lui Ileana Streinu, LISP /35/ o foarte elegantă prezentare a limbajului.

Demonstrarea riguroasă a faptului că LISP-ul pur este echivalent cu modelele de calculabilitate bine cunoscute (maşina Turing, lambda-calcul, funcţiile recursive, etc.) depăşeşte cadrul propus al acestei cărţi (vezi /33/ şi /37/).

4.1. Adevăr şi fals. T şi NIL

Prin convenţie în limbajul LISP valoarea de FALS este notată cu NIL.

NIL este un atom rezervat de sistem. NIL mai desemnează şi ideea de listă vidă, lista fără nici un element, adică NIL este echivalent cu (). Ambiguitatea atomului NIL de a desemna deopotrivă falsul şi lista vidă este puternic speculată în LISP. Iată o sesiune de lucru, (semnul : este *prompt*-ul sistemului

nostru şi este urmat de ceea ce introduce programatorul, apoi pe rîndul următor apare răspunsul sistemului):

:nil
NIL
:0
NIL

Obiectul listei vide are un statut special: este singurul obiect în acelaşi timp şi atom şi listă. De reţinut că în LISP ***tot ce nu este fals este considerat ca fiind adevărat.*** Deci testarea adevărului unei condiţii se face prin verificarea dacă este diferită de NIL. Totuşi pentru **adevăr** există şi un atom special, rezervat, şi anume T (True = adevărat în limba engleză). Avem:

:t
T

4.2. Predicatul

Un **predicat** este o funcţie care ia valorile de adevăr ***adevărat*** sau ***fals***. În LISP predicatele iau valorile de T sau NIL. Amintim că orice obiect care nu este NIL este considerat ca fiind adevărat.
Astfel unele predicate pot lua valoarea NIL pentru fals şi orice altă valoare diferită de NIL pentru adevărat.
Vom exemplifica noţiunea de predicat pe nişte predicate aritmetice folosite pentru numere (atomi numerici).

Exemple.
:(= 2 3)
NIL
:(< 5 8)
T

Foarte frecvent se pune litera **P** la sfîrşitul unui nume de predicat.

NUMBERP **(NUMBERP atom)**
Predicat care întoarce T dacă atom este număr, altfel NIL.
Exemple. (NUMBERP 8.9) → T (NUMBERP 'melania) → NIL

INTEGERP **(INTEGERP număr)**

Predicat care întoarce T dacă număr este întreg, altfel NIL.

Exemple. (INTEGERP 8.9) → NIL (INTERGERP 'A) → NIL

<, >, >= , <= **(< număr1 număr2)**

Predicatele < şi <= (mai mic sau egal) sunt funcţiile clasice de comparare a numerelor din aritmetică.

Exemplu. (< 5 8) → T (<) → mesaj: prea putine argumente

ZEROP **(ZEROP număr)**

Deseori apare necesitatea testării dacă un număr este zero sau nu. Pentru această situaţie avem funcţia ZEROP.

Exemple. (SETF A 7) (ZEROP A) → NIL

Iată funcţia cum putem s-o scriem în LISP:

(DEFUN ZEROP (N) (= N 0))

MINUSP, PLUSP **(MINUSP număr)** **(PLUSP număr)**

Pentru a testa dacă un număr este negativ putem să folosim predicatul MINUSP, iar pentru a vedea dacă este pozitiv predicatul PLUSP.

(DEFUN MINUSP (N) (< N 0)) (DEFUN PLUSP (N) (> N 0))

Exerciţii. Testaţi la calculator predicatele existente la capitolul aritmetică. Propunem:

(= 5 8)→	(= (+ 7 3) (+ 5 5)) →	(<= 7 7) →
(ZEROP 8) →	(MINUSP 7) →	(EVENP 7) →
(= 1 (/ 5 5)) →	(=/ 7 7) →	(NUMBERP 2.1) →

4.3. Atribuirea în LISP

(SETF var1 valoare2 var2 valoare2 ...)

Orice atom simbolic poate fi considerat o variabilă. O variabilă poate lua valori, numere, nume de simboli, liste etc. De exemplu, vrem să-i dăm variabile L valoarea 5 scriem:

:(SETF L 5)

5

Din acel moment pînă la o nouă schimbare variabila L va avea valoarea 5. Dacă nu suntem sub incidenţa vreunei declaraţii de variabilă locală pentru L, valoarea ei se menţine în contextul în care lucrăm. Variabila L este *globală*. Scriind (SETF L 100) → 100, variabila L a luat valoarea 100, ştergînd vechea sa valoare. Avem

:(SETF A 2 B 3 C 4)
4
;A devine 2, B devine 3 , C devine 4

4.4. Funcţiile LISP-ului pur

QUOTE sau ' **(QUOTE obiect) sau 'obiect**

Funcţia QUOTE este o funcţie de bază a LISP-ului pur. QUOTE se poate scrie în două moduri:

Datorită frecvenţei ei de utilizare este preferată scrierea prescurtată prin ă. Ea are rolul de a stabili distincţia între listele ***dată*** şi listele ***program***, mai precis între listele care se evaluează şi cele care se iau ca atare. Orice obiect LISP care este prezentat sistemului fără a avea pe ă în faţă este supus evaluării, şi invers: o listă care-l are pe ' în faţă este luată ca atare. În lumea compunerilor de funcţii pe care este bazat limbajul de programare funcţională LISP, QUOTE are de fapt rolul funcţiei identitate.

Exerciţii.

1) Transformaţi în scrierea cu ' , următoarele expresii LISP.

(QUOTE (A)) → '(A)) (QUOTE A) →

(QUOTE (+ 2 3)) → (+ (QUOTE N)(QUOTE M)) →

(QUOTE (+ (QUOTE A) (QUOTE B))) → ' (+ ('A 'B))

2) Scrieţi cu QUOTE următoarele expresii:

'(A 'B) →

(F '(X Y)) →(F (QUOTE (X Y)))

(F '(X) '(Y)) →

3) Completaţi următoarele evaluări:

'(A B) → (A B) '(+ 2 3) →

(+ 2 3) → 'A →

2 → 2 '2 →

'(A 'B) → '(* (+ 2 3) 5) →

(* (+ 2 3) 5) → (* (- 1) 0) →0

EVAL **(EVAL forma)**

EVAL este însăşi inima LISP-ului, evaluatorul la îndemîna noastră. ***Evaluare se numeşte procesul de reducere a listelor din LISP.*** O listă prezentată sistemului LISP este practic oferită spre prelucrare evaluatorului (funcţiei EVAL) procedeu numit **evaluare .**

Astfel tastînd la intrare lista:

:(+ 2 3 4)

9

se va obţine calculul operaţiei +. În schimb dacă se introduce lista:

:(2 3 4)

Funcţia EVAL va eşua evaluarea, căci va observa imediat că pe primul loc nu se află o funcţie, ci numărul 2. În acest caz eroarea va fi semnalată: ***2 nu este nume de funcţie.*** Avem în schimb

:'(2 3 4)

(2 3 4)

Acest lucru este echivalent cu (QUOTE (2 3 4)) şi va întoarce chiar lista dată, funcţia QUOTE avînd rolul funcţiei identitate.

Vom nota cu → procesul de evaluare a unei liste.

Exemple. (+ 2 3) → 5 (+ 2 (* 3 4)) → 14 5 → 5

(2 (+ 3 4)) → *eroare: 2 nu este functie*

Prezentăm două reguli simple de bază pentru înţelegerea funcţiei EVAL:

1) EVAL aplicat unui atom caută valoarea acelui atom

2) EVAL aplicat unei liste caută funcţia pe primul loc al listei şi în caz afirmativ evaluează (aplică regulile 1 şi 2) pe rînd argumentele funcţiei şi apoi aplică funcţia.

Exemplu. Considerăm că variabila L are valoarea (+ 1 2 3). Acest lucru s-a obţinut prin

:(SETF L '(+ 1 2 3))

Deci:

L → (+ 1 2 3)	(EVAL L) → 6
(EVAL 'L) → (+ 1 2 3)	(EVAL ' ' L) → L
(EVAL 'L → (QUOTE L)	(+ L L) → eroare
(+ 'L 'L) → eroare	(+ (EVAL L) (EVAL L)) → 12

Interpretorul LISP ***citeşte*** forma LISP, apoi o ***evaluează*** (o preia funcţia EVAL) şi apoi ***tipăreşte*** rezultatul:

READ = citeşte forma (READ forma)
EVAL = evaluează forma (EVAL forma)
PRINT = tipăreşte forma (PRINT forma)

Acest proces se poate scrie chiar în limbajul LISP astfel:

(PRINT (EVAL (READ)))

Exerciţiu 1. Evaluaţi listele şi semnalaţi erorile:

(+ (* 5 * 6)) → ***eroare: * nu este număr***
(+ 8 (= 8 7)) →
(+ (* 5 6) 3 (/ 4 2)) →
(+ (* 4 6) 3) →
(5 + 6 * 3) → ***eroare: 5 nu este funcţie***
(+ * 4 6) →

Exerciţiu 2. Evaluaţi listele, unde L are valoarea (+ 5 3).

(SETF L '(+ 5 3)) → (+ 5 3)
L →
(EVAL '(+ 5 3)) → (+ 5 3)
'EVAL → EVAL
'(EVAL (+ 5 3)) → (EVAL (+ 5 3))
'(EVAL EVAL) →
(EVAL NIL) → NIL
(EVAL T) →
(EVAL L) → 8
(+ (EVAL L) 6) →
(+ L L) → EROARE
(+ (EVAL L)(EVAL L)) → 16

ATOM **(ATOM obiect)**

Funcţia ATOM este o funcţie a LISP-ului pur. ATOM este un predicat de un argument pe care îl evaluează. Valorile întoarse sunt T (pentru adevărat) sau NIL (pentru fals), testînd dacă obiectul este sau nu atom. De exemplu:

:(ATOM 'MELANIA)
T

Considerăm că atomul L a căpătat valoarea (A B C), acest lucru s-a înfăptuit prin (SETF L ă(A B C)) deci L → (A B C). Avem:

(ATOM 'L) → T
(ATOM L) → NIL
(ATOM '(L L)) → NIL
(ATOM (ATOM L)) → T
(ATOM '(ATOM L)) → NIL
(ATOM 7) → T

CAR **(CAR lista)**

Funcţia CAR este o funcţie de bază a LISP-ului pur. CAR întoarce primul element dintr-o listă sau partea din stînga a unei perechi cu punct.

Exemple. (CAR '(A B)) → A
(CAR '((A B) C)) → (A B)
(CAR 'atom) → eroare
(CAR '(A (B C))) → A
(CAR '(((A B) C) D) → ((A B) C)
(CAR '((nil))) → (NIL)

Observaţi că (CAR 'ATOM) este nedefinită şi este semnalată ca eroare. Avem în schimb:
(CAR NIL) → NIL
(CAR '(nil)) → NIL
(CAR '(A)) → A
(CAR '()) → NIL
(CAR '(())) → NIL
(CAR (CAR '((2 3 4) (5 6))) →2

CDR **(CDR lista)**

CDR este o funcţie de bază a LISP-ului pur. CDR întoarce partea dreaptă a unei perechi cu punct. Aplicat unei liste CDR va întoarce restul listei după ce se extrage CAR-ul (primul element).

Exemple.
(CDR '(A)) → NIL
(CDR '(A . B)) → B
(CDR (CDR '(A B C))) → C
(CDR '(A B C)) → (B C)
(CDR '(1 (B))) → ((B))
(CDR (CDR '((A 1) B 2))) → (2)

C...R CAAR, CADR, CDDR, CDAR, CAAAR, CADDR,...

Compunerile funcţiilor CDR şi CAR sunt extrem de frecvente, de aceea variante de compuneri pînă la un anumit număr se vor găsi în sistemul pe care lucraţi sub numele unor funcţii C...R. Compunerile de CAR şi CDR se redenumesc cu litera C urmată de ordinea respectivă de A şi D şi apoi R. Astfel

(CAR (CDR (CDR (CDR L)))) devine (CADDDR L)

Majoritatea sistemelor au implementate compuneri de 2 poziţii CxxR cu *x* sau *A* sau *D*. Combinaţiile mai mari le putem defini şi noi.

Exemple. Observaţi modul prescurtat de notare a funcţiilor compuse.
(CAR (CDR L))
(CDR (CDR (CAR '((A B) C))))
(CAR (CAR (CDR L)))
(CDR (CAR (CDR L)))
(CADR L)
(CDDAR '((A B) C))
(CAADR L)
(CDADR L)

CONS **(CONS obiect1 obiect2)**

Funcţia CONS (CONS de la CONStructor) are două argumente pe care le evaluează, şi întoarce o pereche cu punct care are drept CAR primul obiect şi drept CDR cel de-al doilea. CONS aplicat asupra a două liste va întoarce lista a doua căreia i se va adăuga pe prima poziţie prima listă.

Exemple. (CONS 'A ()) → (A)
(CONS 'A '(B C)) → (A B C)
(CONS '(A B) '(C)) → ((A B) C)
(CONS '(1 2) '((A B)(C D))) → ((1 2)(A B)(C D))
(CONS T NIL)→ (T)
(CONS '(A) '(1 2 3)) → ((A) 1 2 3)

EQ, EQUAL **(EQ atom1 atom2) (EQUAL ob1 ob2)**

Funcţia EQ este o funcţie de bază a LISP-ului pur. Este un predicat care întoarce T sau NIL, după cum atomii comparaţi sunt egali sau nu. Se observă că este suficient să se definească egalitatea pentru atomi (funcţia EQ) ca apoi să se extindă pentru liste EQUAL (în particular EQUAL compară orice tip dc date).

Exemple. (EQ 'A 'A) → T (EQUAL 5 (+ 2 3))→ T
(EQUAL (CAR '(A (B C))) (CAR '(A (B) C))) →T
(EQUAL (EQUAL 'L 'L) NIL) → NIL

Prezentăm funcţia **EQUAL** pentru liste, având funcţia EQ pentru testarea egalităţii a doi atomi. Este un exemplu mai complex care necesită cunoaşterea mecanismului de recusivitate.

```
(DEFUN EQUAL (X Y) ; X si Y  s-expresii
(COND  ((AND (ATOM X) (ATOM Y)) (EQ X Y))
        ((OR (ATOM X) (ATOM Y)) NIL)
       ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR  X) (CDR Y)))
)) ; daca primul din X= primul din Y caut mai departe
```

În ce situaţie avem (EQUAL L 'L) → T ?

= **(= număr1 număr2 ...)**

Egalitatea a două numere poate fi testată şi cu predicatul EQUAL, dar este mai rapid de testat cu predicatul specific pentru aritmetică =, care nu va mai face teste de tip.

Exemplu. (= -5 5) → NIL (= 3 3 5 3) → NIL (= 2) → T (=) → T
(= 'A 'A) → *eroare de tip, a nu este numar*

CONDiţionala **(COND ((predicat1 ...)(predicat2 ...) ...))**

Obligatoriu în orice limbaj de programare trebuie să existe o funcţie de test. În majoritatea limbajelor se numeşte **IF** şi este de forma

(dacă **predicat** atunci-ceva altfel-ceva2) sau
(IF test then else)

Exemplu.

:(IF (> 8 0) 'pozitiv 'negativ)

pozitiv

În LISP această funcţie, condiţionalul, este mai generală. COND admite oricîte ramuri.

Exemplu. Modulul unui număr N este funcţia ABS (de la *ABSolute* în engleză). Avem (ABS -5.3) → 5.3

```
(DEFUN ABS (N) ; modulul unui numar N
(COND ((> N 0) N)       ; daca N>0 atunci rezulatul este  N
      ((= N 0) 0)       ; dacă N = 0 atunci  modul este 0
      ((< N 0) (- N))   ; daca N< 0 atunci  modulul este -N
      (T 'eroare)       ; N nu este in nici unul  din cazuri: eroare
) )
```

În ultima ramură a COND-ului am pus pe prima poziţie atomul T. Deci am forţat intrarea pe aceea ramură în cazul în care nu a fost selectată nici o altă ramură mai înainte. Atomul T are valoarea adevărat şi este atom rezervat (nu putem să-i dăm alte valori).

Definiţia lui COND:

```
(COND   (ramura-1)
        (ramura-2)

        (ramura-n)
)
```

Fiecare ramură are oricîte obiecte LISP (liste sau atomi):

ramura = (predicat1 obiect2 ... obiectn)

Se ia pe rînd fiecare ramură şi se evaluează primul obiect, dacă este adevărat se selectează acea ramură celelalte fiind ignorate complet şi se evaluează pe rînd obiectele, fiind întors ca valoare a COND-ului rezultatul evaluării ultimului obiect.

Deci, primul obiect este interpretat ca un predicat, testînd dacă are valoarea T sau NIL. Reamintim că orice nu este NIL, este considerat ca adevărat.
Putem scrie mai restrîns modulul unui număr.

```
(DEFUN  ABS (N) ;modulul lui N
(COND  ((>= N  0) N)
          (T (- N))
))
```

Iată cîteva exemple de aplicare a COND-ului:

```
(COND    (T (+ 7 8))
          ( 'NU-AJUNGE-NICIODATA)
) → 15
(COND   ((= 7 6) 'RAU)
          (( 7 6) (+ 3 4) (* 5 4))
) → 20
((COND ((= 3 4) 'NU)
        ((EQ () NIL) (SETF L `DA) 'BINE)
) → BINE
(COND  ((= 1 2) 'RAU)
         ((+ 4 5) (SETF  L (* 3 4)) (+ L 1))
         (T 'NU-AJUNGE-AICI)
) → 13
```

Dacă COND nu poate selecta nici o ramură, atunci întoarce NIL.

/= **(/= N1 N2)**

Să scriem predicatul ***diferit*** pentru numere.

Exemple. (/= 5 6) → NIL (/= 5 5) → T

Soluţie. Prezentăm două moduri de scriere a funcţiei diferit /=

```
(DEFUN /= (N1 N2)
(COND  ((= N1 N2) NIL)
          (T)
 ))
(DEFUN /=  (N1 N2)  (IF (= N1 N2) T NIL ))
```

MIN, MAX **(MIN [numar ...]) (MAX [numar ...])**

Să scriem o funcţie MIN de două argumente care să întoarcă cel mai mic număr între două numere date. Similar MAX, pentru maximul între două numere. Observaţi că funcţia sistem are oricîte argumente.

Exemple.(MIN 5 4) → 4 (MAX 1 3 5 2 6 9 8) → 9

Soluţie.

```
(DEFUN MIN (X Y)
(COND (( > X Y) Y)
      (X)
))
```

```
(DEFUN MAX (X Y)
(COND (( > X Y) X)
      ( Y )
))
```

Definirea funcţiilor **(DEFUN nume-func lista-arg corp)**

Cum definim o funcţie? Intîi trebuie să-i dăm un ***nume***. Apoi îi explicităm ***argumentele***, apoi ceea ce face funcţia, prin ***corpul*** funcţiei.

Să definim funcţia f(x) = 10 x + 2

```
(DEFUN  F  (X)   (+  (*  10  X) 2)  )
```

nume-func - F

lista-argumentelor - (X)

corp - (+ (* 10 X) 2)

Se apelează:

:(F 5)

52

Funcţiile definite de noi se numesc ***funcţii utilizator*** spre deosebire de ***funcţiile sistem***. Este bine pentru a evita confuziile să dăm nume diferite funcţiilor noastre de cele date de sistem. Iată cîteva exemple:

Să definim funcţia $f(x,y) = x^3 + y^3$

```
(DEFUN SUMA-CUBURI (X Y)
(+ (* X X X) (* X X X))
)
```

Putem s-o construim şi scriind în prealabil $f(x) = x^3$

```
(DEFUN CUB  (X)  (*  X  X  X)  )
```

şi avem
(DEFUN SUMA-CUBURI (X Y) (+ (CUB X) (CUB Y)))
Apelul funcţiei va fi:
:(SUMA-CUBURI 3 2)
35

1+, 1- **(1+ număr) (1- număr)**
Iată funcţiile 1+ şi 1- care adaugă 1, respectiv scad 1, dintr-un număr.
Exemple. (1+ 6) → 7 (1- 8) → 7 (1+ (1- 70)) → 70
(DEFUN 1+ (N) (+ N 1)) (DEFUN 1- (N) (- N 1))

(APPLY functie lista)
Avem o listă de numere L = (1 2 3). Vrem să le adunăm. Deci să formăm lista (+ 1 2 3) pe care s-o trimitem evaluatorului pentru calcul. Putem scrie acest lucru astfel:
:(SETF L '(1 2 3))
(1 2 3)
:(SETF X (CONS '+ L))
(+ 1 2 3)
:(EVAL X)
6
sau
:(APPLY '+ L) ; aplica functia + listei !
Funcţia APPLY o putem scrie şi noi:
(DEFUN APPLY (F L) ; F = functie , L=lista cu argumentele
(EVAL (CONS F L)) ; evaluez lista cu functia L in fata lui L
)

Exerciţii. 4. LISP-ul pur

1. Completaţi:
(CAR '(STING . DREPT)) → STING (CAR '(A . B)) →
(CAR '(A .(B . C))) → A (CAR (CAR '((A B) C))) →

(CAR '(CAR '(A B))) →
(CAR '((()) (()))) →
(CAR '((A . B) . (C . D))) →
(CAR 'CAR) → EROARE
(CAR (CAR '((A 1)(B 2))) →
(CAR (CAR '((A)))) →
'(CAR CAR) → '(CAR CAR)
(CAR '((5))) → (5)

2. Verificaţi următoarele evaluări:

(ATOM 'a) →
(ATOM T) → T
(ATOM 12) →
(ATOM '(atom atom)) → NIL
(ATOM (ATOM T))→
(ATOM NIL) → T
(ATOM ()) →
(ATOM '(!) →
(ATOM 'cotcodac*) →
(ATOM (ATOM '(!))) → T

3. Dîndu-se listele L1-L5, completaţi aplicarea funcţiilor C...R.

L1 = (P * Q - (R + Q))
L2 = (- (* P Q) (+ R Q))
L3 = (P SAU (NON Q) SI R)
L4 = ((a11 a12) (a21 a22))
L5 = ((MAMIFER (LUP OAIE)) (PASARE (CUC GAINA)))

	L1	L2	L3	L4	L5
CAR	P				
CDR					
CADR			SAU		
CDDR					
CAAR	eroare				mamifer
CDAR				(A12)	
CAAAR					
CAADR		*			
CADAR				A12	
CDAAR					
CADDR			(NON Q)		
CDDAR					NIL
CDADR					
CDDDR					

4.Se dau listele:

L1 = (A B C D E) L2 = ((A B)(C D)(E)) L3 = ((A B C) (D E))

L4 = ((A (B C)) (D (E))) L5 = (((A B) C) ((D E)))

Scoateţi atomii *A, B , C , D, E* din fiecare listă folosind scrierea prescurtată a compunerilor CAR şi CDR, completînd tabelul:

	L1	L2	L3	L4	L5
A		caar			
B	cadr		cadr		
C				cadadar	
D		cadadr			
E					cadaadr

5. Verificaţi următoarele evaluări:

(CONS 'CAR 'CDR) →

(CONS 'A '(B)) →

(CONS 'A '(B C)) →

(CONS '(A) '(B C)) →

(CONS '(1 2) '(B C)) →

(CONS '(A B) '(B . C)) →

(CONS '(A) '(B . C)) →

(CONS NIL NIL) →

(CONS 'A NIL) →

(CONS 'A '(NIL)) →

(CONS 'NIL NIL) →

(CONS '(NIL) NIL) →

(CDR '(STING . DREPT)) →

(CDR '()) →

(CDR '(A B)) →

(CDR '((A B) C)) →

(CDR '((A B) (C D))) →

(CDR '(A . (B . C))) →

(CONS 'A 'B) →

(CONS 'STINGA 'DREAPTA) →

(CONS 'A '((B) C)) →

(CONS '((A)) '((B) (C))) →

(CONS 'A '(B . C)) →

(CONS '((A . B) . C) 'D) →

(CONS 'A '(B (C D))) →

(CONS '(A . 1) '((B . 2)(C . 3))) →

(CONS NIL '(B C)) →

(CONS NIL '(A B)) →

(CONS '(A B C) NIL) →

(CONS NIL '(A B)) →

(CDR ()) →

(CDR '())) →

(CDR '(A . B)) →

(CDR '((A . B) . (C . D))) →

(CDR '((A) B)) →

(CDR (CDR '(A B C))) →

6. Completaţi ce întorc următoarele COND-uri:

```
(COND  (T)) →
(COND  (T 'REGE)
       (T 'AS)) →
(COND  (( (SETF DELTA -2) 0) 'ARE-SOLUTII)
       ((= DELTA 0) 'RADACINI-EGALE)
       (T 'NU-ARE-SOLUTII)  ) →
(COND  ((CAR '(NIL NIL)) (* 7 8))
       ((CDR '(NIL NIL)) (+ 2 3) (* 5 3))
       (T 'NIMIC) ) →
(COND ((CONS 2 '( 3 4)) )
       (T (* 6 5 )) ) → (2 3 4)
(COND  ((EQ 'A 'B) 'BINE)
       ((EQ 'A (CAR '(A D))) (* 5 4)) →
       (T 'GHICI? )) → 20
(COND  (NIL)
      (NIL T)
       (T NIL)
       (T T) ) →
(COND (1)
       (2)
       (T)  )  → 1
(COND  () ) →
(COND ((SETF A NIL) 'BINE?)(A 'RAU)('A 'CEL-MAI-BINE))  ) →
```

7. Verificaţi dacă o listă are fix 4 elemente.

8. Pentru uşurinţă multe sisteme LISP au dat nume unor combinaţii de CAR şi CDR care scot dintr-o listă elemetul al N-lea.

Exemplu: (THIRD '(A B C D)) → C

Vă prezentăm mai jos numele lor cum apar în CLISP. Rugăm completaţi definirea lor cu C...R:

Primul	FIRST	CAR
Al doilea	SECOND	CADR
Al treilea	THIRD	CADDR
Al patrulea	FOURTH	CADDDR
Al cincilea	FIFTH	
Al şaselea	SIXTH	
Al şaptelea	SEVENTH	
Al optulea	EIGHTH	
Al nouălea	NINTH	
Al zecilea	TENTH	

9. Formaţi o listă din primele elemente a două liste date.

10. Scrieţi o funcţie care interclasează două liste de cîte două elemente.
(INTERCLAS '(A B) '(C D)) → (A C B D)

11. Formaţi un fişier bibliotecă pe care să-l încărcaţi automat de fiecare dată cînd apelaţi sistemul vostru LISP. Puneţi înăuntru funcţiile CAD...R pe cît mai multe nivele. Cîte funcţii sunt cu 4 combinaţii de CAR şi CDR ? Dar pe N poziţii?

5. Aritmetica

5.1. Principalele funcţii aritmetice

Iată un tabel pe scurt ale celor mai folosite funcţii aritmetice:

Operaţie	Funcţie LISP
Adunare	+
Scădere	-
Înmulţire	*
Impărţire	/
Exponenţiere	EXPT
Radical	SQRT
Restul împărţirii	REM
Logaritm	LOG
Rotunjire	TRUNCATE

+ Adunarea **(+ numar1 [numar2 ...])**

Adunarea este concepută ca o funcţie de oricîte argumente, care se evaluează pe rînd. Este necesar să aibă cel puţin un argument.

Exemple. (+ 4 3 2 1 0.0) → 10.000000 (+ 8) → 8

- Scăderea **(- numar1 [numar2 ...])**

Este o funcţie cu oricîte argumente care se evaluează pe rînd. Dacă are un singur argument îi va schimba semnul. În cazul mai multor argumente se scad toate numerele din primul număr.

Exemple. (- 8) → -8 (- 9 1 2 3) → 3 (- 1.2 1) → 0.2

*** Inmulţirea** **(* numar1 [numar2 ...])**

Înmulţirea este concepută ca o funcţie de oricîte argumente numerice care sunt evaluate pe rînd. Ca şi în cazul celorlalte operaţii există convenţia de a întoarce numărul rezultat avînd tipul cel mai mare dintre toate argumentele. Astfel, dacă între argumente este un singur număr real, rezultatul va fi real. Înmulţirea * este deja definită în sistemul cu care lucraţi, încercaţi totuşi să scrieţi singuri o funcţie care să efectueze înmulţirea a două numere naturale. Înmulţirea va fi privită ca o adunare repetată.

```
(DEFUN * (N M) ; N si M numere naturale
(COND ((= M 1) N) ; N * 1 = N
      ((= M 0) 0)  ; N *  0 = 0
      (T (+ N (* N (1- M))))  ; N * M = N + N * (M − 1)
))
```

/ Împărţirea **(/ numar1 [numar2 ...])**

Împărţirea /, este o funcţie cu oricîte argumente. Se împarte primul număr număr1, pe rînd la toate celelalte. Rezultatul este întreg (INTEGER) sau real (FLOAT), depinzînd de tipul numerelor în cauză.

Exemple. (/ 4 2 2) → 1 (/ 8 3) → 2 (/ 1.0 2 2) → 0.250000

EXPonenţiala **(EXPT bază exponent)**

EXPT este funcţia exponenţială, deci ***baza*** la puterea ***exponent***. Iată funcţia exponent definită ca o înmulţire repetată, într-o manieră recursivă, pentru o bază număr real, iar exponent un număr natural.

Exemple. (EXPT 7.0 0) → 1 (EXPT 2.0 3.0) → 8.000000

Soluţie.

```
(DEFUN EXPT (A N) ; A la puterea N
(COND  ((= N 0) 1) ; A 0  = 1
       (T (* A (EXPT A (1- N))))  ; A^n = A *  A^(n-1)
))
```

De obicei găsiţi în LISP-ul cu care lucraţi şi funcţia expenenţiere cu baza *e*, (EXP 2)→ 7.389056

Radical **(SQRT real)**

SQRT vine din limba engleză(SQuare RooT) însemnînd *rădăcină pătrată*. Deci această funcţie scoate radicalul dintr-un număr real pozitiv. Rezultatul este întotdeauna real.

Exemple. (SQRT 4) → 2.000000 (SQRT 2) → 1.414214

Restul împărţirii **(REM nr-întreg nr-întreg)**

Pentru a afla restul a două numere întregi există funcţia sistem REM, de la REMainder (în engleză *rest*). Avem: (REM 10 3) → 1

Am fi putut s-o obţinem şi noi, astfel:

```
(DEFUN REM (N M) ; N si M sunt intregi
(- N (* (/ N M) M))
) ; (/ N M) intoarce un numar intreg pt M si N intregi
```

LOGaritm **(LOG numar)**

Logaritmul unui număr în baza **e.** Scrieţi o funcţie care să calculeze logaritmul pentru o bază dată. Avem: (LOG 2) → 0.693147

Trunchierea **(TRUNCATE număr)**

TRUNCATE înseamnă a trunchia. Numerelor reale funcţia TRUNCATE le va trunca partea zecimală. Rezultatul este întotdeauna întreg. Încercaţi să construiţi singuri această funcţie.

Exemple. (TRUNCATE (SQRT 4)) → 2 (TRUNCATE -7.13) → -7

Par- EVENP, impar- ODDP **(EVENP întreg) (ODDP întreg)**

Pentru a vedea dacă un număr este par sau nu, se poate folosi funcţia EVENP (în engleză înseamnă *par*) şi se practică punerea lui **P** la sfîrşitul cuvîntului pentru a evidenţia că este vorba de un predicat.

Simetrica ei, funcţia ODDP (*impar*) testează dacă numărul este impar.

Exemple. (EVENP 6) → T (ODDP 6) → NIL

Iată cum se scriu uşor aceste funcţii folosind funcţia REM (Rest).

```
(DEFUN EVENP (N) ; restul impartirii la 2 este 0
```

```
(ZEROP (REM N 2)))
(DEFUN ODDP (N) ; restul impartirii diferit de 0
(/= 0 (REM N 2))
)
```

5.2. Funcţii trigonometrice

În multe sisteme funcţiile trigonometrice sunt concepute pentru unghiuri date în radiani. Valoarea întoarsă de ASIN (arcsin), ATAN (arctan) şi ACOS este tot în radiani. Iată câteva funcţii:

SIN COS TAN ASIN ACOS ATAN

Exemple. (ACOS -1) → 3.141593 (valoarea lui π)

5.3. Numere aleatoare

RANDOM **(RANDOM număr)**

Random înseamnă în limba engleză ALEATOR. Foarte des avem nevoie ca sistemul să ne furnizeze un număr aleator mai mic decît un prag dat. Iată (RANDOM 10) va genera un număr între 0 şi 9 (inclusiv). Să scriem o funcţie ZAR.

Exemple. (ZAR) →1 (ZAR) →6 (ZAR) →1 (ZAR) → 5

```
(DEFUN ZAR () (RANDOM 7))
```

5.4. Cel mai mic divizor comun

(GCD intreg1 intreg2)

Algoritmul lui Euclid pentru aflarea celui mai mare divizor comun a doua numere, GCD (*Greatest Common Divisor* în limba engleză).

Exemplu. (GCD 15 21) → 3

```
(DEFUN GCD (N M) ; N si M numere naturale
(COND ((ZEROP (REM N M)) M)
      (T (GCD  M (REM N M)))
))
```

1. Calculaţi aria cercului (ARIA-CERC raza).
2. Calculaţi lungimea cercului (LUNG-CERC raza).
3. Calculaţi media aritmetică, geometrică şi armonică a trei numere.
4. Dîndu-se un unghi să se afişeze toate valorile funcţiilor trigonometrice asociate.
5. Pentru calculul timpului sistemul oferă funcţia (GET-DECODED-TIME) → (sec minut ora zi luna an zi-din-saptamina). Construiţi separat următoarele funcţii: ora, minut, zi, luna, an, ziua-săptămînii. De exemplu: (ZIUA-SAPTAMINII) → MARTI
6. Scrieţi o funcţie PAUZA? care să spună dacă suntem în pauză sau nu (pauzele sunt 10 minute începînd de la fără 10 până la fix).
7. Scrieţi o funcţie care să spună în ce zi a săptămînii cade ziua dumneavoastră de naştere.
8. Scrieţi o funcţie (DREPTUNGHIC? x y z) care să verifice dăcă trei laturi formează un triunghi dreptunghic.
9. Verificaţi dacă trei laturi date pot forma un triunghi.
10. Calculaţi ARIA unui triunghi. (Comentarii)
11. Obţineţi numărul π cu cît mai multe zecimale exacte.
12. Transformaţi un unghi dat în grade în radiani.
13. Scrieţi un program care să scoată rădăcinile ecuaţiei de gradul doi.
14. Tipăriţi toţi divizorii unui număr natural dat.
15. Scrieţi cel mai mic multiplu comun a două numere naturale numit **LCM** (least common multiple).
16. Scrieţi o funcţie PRIM? care să verifice dacă un număr este prim.
17. Verificaţi dacă un număr este pătrat perfect, (PATRAT? n).
18. Scrieţi o funcţie care să convertească suma de N lei în dolari.
19. Scrieţi o funcţie care să calculeze DOBINDA pe un număr de zile a unei sume ţinute în cont la o bancă.
20. Generaţi numere aleatoare pînă în 1000. Calculaţi frecvenţa numerelor prime.

21. Se modică frecvența apariției numerelor prime dacă le generăm aleator pînă la 10000 ? (vezi exercițiul precedent)
22. Generați la întîmplare 3 numere mai mici ca N. Care este probabilitatea ca ele să poată reprezenta laturile unui triunghi?
23. Generați la întîmplare 3 numere mai mici ca N. Care este probabilitatea ca un număr să fie suma celorlalte?
24. Închipuiți-vă că jucați table. Aveți o funcție (ZARURI) care întoarce o listă cu 2 numere între 1 și 6. Faceți experimente cu ea. Observați de cîte ori iese (6 6). Se apropie de 1/36 ?

6. Recursivitatea

Cum să definim o funcţie? O tehnică posibilă este *recursivitatea.*
Cînd definim o funcţie recursiv o definim cu ajutorul ei înşişi. E posibil aşa ceva, să defineşti un lucru prin el însuşi? Da, dacă îl defineşti cu grijă. Să luăm de exemplu:

6.1. Ultimul element

Să scriem o funcţie care întoarce lista cu ultimul element al lui *lista. LAST* înseamnă în limba engleză *ultimul.*
Exemple. (LAST '(A B C)) → (C) (LAST '(A (B C))) → ((B C))
Soluţie.

```
(DEFUN LAST (L) ;L lista
(COND  ((NULL (CDR L)) L) ; daca L este formata dintr-un singur obiect
       (T (LAST (CDR L))) ; reduc problema la restul listei
))
```

Ca să calculez (LAST '(A B C)) reduc problema la (LAST '(B C))
Ca să calculez (LAST '(B C)) reduc problema la (LAST '(C))
(LAST '(C)) o ia pe prima ramură a COND-ului, căci
(CDR '(C)) → NIL şi (NULL (CDR '(C))) → T
deci întoarce L care este (C).
Iată schematic REDUCEREA calculului.

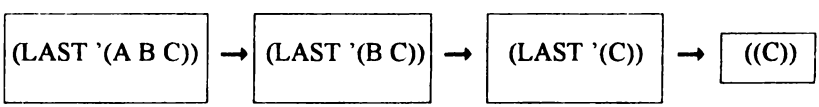


Fig. 6.1.

Funcţia apelată recursiv nu are nevoie să ţină minte (să colecteze) valorile apelurilor. Acest tip de recursie se numeşte ***recursie pe coadă.***

6.2. Factorialul

Binecunoscuta funcţie FACTORIAL. Reamintim că definiţia lui n! este: 0! = 1 şi n! = n * (n-1)!

Exemplu. (FACT 4) → 24 (FACT 5) → 120

Soluţie. Nu facem decît să transcriem în LISP definiţia de mai sus.

```
(DEFUN FACT (N) ; N=numar natural
(COND  ((= N 0) 1) ; daca N=0, FACTORIAL=1
       (T (* N (FACT (1- N)) ; (FACT N) = N * ( FACT (N-1) )
))
```

Iată schematic cum funcţionează mecanismul recursiei:

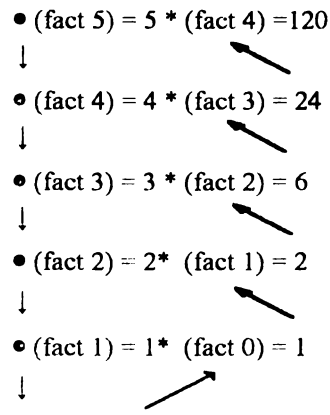


Fig.6.2. Calculul lui *fact (5)*

De data aceasta problema nu se reduce pur şi simplu. Trebuie memorate rezultatele pe parcurs. Săgeţile în jos indică apelurile pe rînd: ca să calculăm *(fact 5)* trebuie să calculăm *(fact 4)* care apelează *(fact 3)* etc. Săgeţile în sus indică transmiterea rezultatului de la un apel la altul. Pe scurt spus, în recursia lui LAST avem săgeţi numai într-un sens, iar aici avem în două sensuri. Observăm că în cazul recursiei lui FACT de mai sus rezultatele acestea parţiale trebuie memorate într-o stivă, care ocupă memorie.

Ne punem întrebarea: putem să scriem FACTORIAL altfel, tot recursiv, dar cu recursie pe coadă? Adică la fiecare apel să REDUCEM problema (ca în cazul lui LAST). Cu alte cuvinte: să scăpăm de transmiterea valorilor de jos în sus.

Răspunsul este DA. Pentru aceasta să introducem o ***variabilă colectoare***, care ţine minte rezultatul. Apelul va fi: (FACT 5 1) → 24

```
(DEFUN FACT (N R)  ; N-numar, R-variabila colectoare
(COND  ((= N 0) R)     ; R =1 la apelul programului
          (T  (FACT (- N 1) (* N R))) ;înmulţim la R pe rînd nr: N, N-1,..
))
```

Următoarea schemă redă fluxul apelurilor:

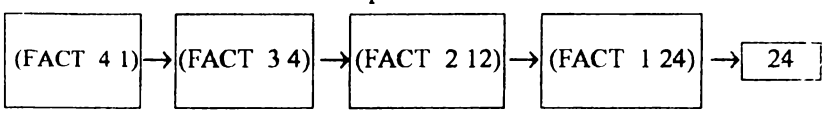


Fig. 6.3. Calculul lui *fact(4)*

6.3. Şirul lui FIBONACCI

Să explorăm un alt tip de recursivitate.Să ne oprim puţin asupra celebrei funcţii a matematicianului italian Leonardo da Pisa sau FIBONACCI (1170-1240). Şirul lui Fibonacci are următoarea regulă:

fib(0) = fib(1) = 1

fib(n+1) = fib(n) + fib(n-1)

Avem: 1, 1, 2, 3, 5, 8,13, 21, ... Iar: (fib 5) → 8

Dacă o scriem după definiţie obţinem:

```
(DEFUN FIB (N); definitie recursiva
(COND  ((= N 0) 1)
          ((= N 1) 1)
          (T (+ (FIB (1- N)) (FIB (- N 2))))
))
```

Iată ce obţinem, dacă urmărim apelurile efectuate pentru a calcula pe (fib 5).

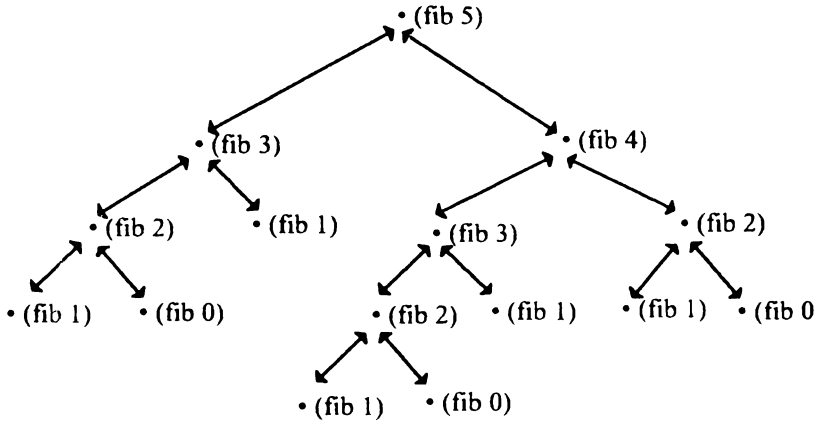


Fig. 6.2. Calculul lui (fib 5)

Observăm că anumite bucăţi din arbore se repetă.Calculăm pe fib(3) de 2 ori, iar pe (fib 2) de 3 ori. Realizăm că dacă scriem arborele pentru (fib 6) nici nu mai avem loc pe hîrtie. O serie de calcule sunt evident redondante şi calculatorul oricît ar fi de rapid şi oricîtă memorie ar avea se poate poticni foarte repede. Se pare că recurisivitatea NU este o tehnică prea bună în acest caz. Să încercăm să o reducem cu tehnica variabilei colectoare la cazul cel mai simplu de recursie pe coadă.

Iată FIB cu ajutorul variabilelor colectoare în recursie pe coadă.

```
(DEFUN FIB (N X Y) ; recursie cu variabilele  colectoare X si Y
(COND  ((=   N 0) X) ; initial X=1 si Y=1, vezi inceput sir FIBONACCI
          (T (F (1-  N) X ( + X Y) ) )
))
```

Iată ce se întîmplă cînd apelăm (fib 5 1 1) → 8

•(fib 5 1 1) → •(fib 4 1 2) → •(fib 3 2 3) → •(fib 2 3 5) → •(fib 1 5 8)

•(fib 0 8 13) → 8

Fig. 6.5.

Observaţi cîte apeluri facem faţă de arborele din Fig.6.4.

Tehnica constă în a transporta rezultatele parţiale în variabilele colectoare X şi Y. De data aceasta a fost nevoie de 2 variabile.
Ne întrebăm dacă nu putem formula funcţia lui Fibonacci fără recursie deloc, folosind compunerile obişnuite de funcţii aritmetice?
Răspunsul este DA Iată un mod nerecursiv de a scrie pe Fibonacci:

$$fib(n)= [(1+ \sqrt{5})^{n+1} - (1 - \sqrt{5})^{n+1}] / \sqrt{5} * 2^{n+1}$$

Cred că este destul de convingător că această abordare este total lipsită de transparenţă: fiindu-ne destul de greu să ghicim repede cum arată şirul dacă avem această formulă în faţă.

Ca să simţiţi cît de complexă poate fi o funcţie recursivă vă prezentăm un exemplu de ***cea mai recursivă*** funcţie. Pentru ea nu este nici o speranţă să-i găsim o formulă ***miraculoasă*** care s-o calculeze direct (fără recursivitate) precum am găsit pentru Fibonacci. Este vorba de funcţia lui Ackermann.

6.4. Funcţia lui ACKERMANN

Funcţia lui Ackermann este un exemplu clasic de funcţie recursivă care nu este primitiv recursivă /35/. Avem:

ack(0,m)= m + 1
ack(n,0) = ack (n-1 , 1)
ack(n,m) = ack (n-1, ack (n, m-1))

Soluţie.

```
(DEFUN ACK (N M)
(COND  ((ZEROP N)(1+  M))
       ((ZEROP M)(ACK (1-  N) 1))
       (T (ACK (1- N) (ACK N (1-  M))))
))
```

Observaţi funcţia lui ACKERMANN. Încercaţi s-o calculaţi singur. La care N şi M vă opriţi? Dar calculatorul?

În continuare un mic exercițiu pe care-l datorăm capitolului de *Aritmetica*. După ce am trecut prin atîtea încercări grele cu recursivitatea să vedem cum scriem ceva simplu:

6.5. Tabelul unei funcții

Să începem cu *tabelul logaritmic*. Iată o funcție care afișează toate valorile funcției logaritm pentru un *pas* dat, pînă la o valoare dată *val*.

```
:(TABEL-LOG 1 3)
1.098612    ; val=3
0.693147    ; val=2
0.000000    ; val=1
STOP
```

Avem:

```
(DEFUN TABEL-LOG (PAS VAL); tabelul lui LOG
(COND ((<= VAL 0)  'STOP) ; incepind cu VAL pina la 0
      (T  (PRINT (LOG VAL))  (TABEL-LOG PAS (- VAL PAS) ))
))
```

Să scriem o funcție mai generală TABEL care să preia ca argument funcția (o funcție care admite drept agument o altă funcție!) Avem:

```
:(TABEL 'LOG 1 3)
1.098612    ; val=3 adica log(3)
0.693147    ; val=2 adica log(2)
0.000000    ; val=1 adica log (1)
STOP

:(TABEL 'SIN 1 3)
1.098612    ; val=3 adica  sin(3)
```

```
0.693147      ; val=2 adica  sin(2)
0.000000      ; val=1 adica  sin(1)
STOP
```

Avem:

```
(DEFUN TABEL (FUN PAS VAL); tabelul lui FUN=functia
(COND ((<= VAL 0) 'STOP) ; incepind cu VAL pina la 0 cu PAS
      (T (PRINT (FUNCALL FUN VAL))
                        (TABEL FUN PAS (- VAL PAS) ))
))
```

Funcţia sistem (FUNCALL F arg1 arg2 ...) are rolul de a diferenţia funcţia F de o variabilă obişnuită şi de a o aplica corespunzător, formează (F arg1 arg2...) şi apoi evaluează.

Concluzii. 6. Recursivitate

- O definiţie recursivă presupune cel puţin:
  1. o condiţie de oprire
  2. o relaţie între F(N) şi F(N-1) sau F(lista) şi F(cdr lista)
- Sunt mai multe feluri de recursivitate.
- Nu orice recursivitate este costisitoare (consumatoare de memorie şi timp).
- Recursia pe coadă este la fel de eficientă ca orice iteraţie.
- Unele recursii pot fi transformate uşor în recursii pe coadă introducînd variabile colectoare.
- Recursivitatea presupune un alt mod de a pune problema.
- Recursivitatea este structura de control a LISP-ului pur.

Exerciţii. 6. Recursivitatea

1. Observaţi funcţia lui FIBONACCI. Care este cel mai mare număr pe care-l puteţi calcula după prima definiţie? Dar cu recursie pe coadă?

2. Scrieţi un program care să afişeze timpii de evaluare pentru funcţiile de test clasice (FIB, ACK).
3. Observaţi funcţiile de la capitolul 5. Aritmetică (EXPT, * , GCD). Care sunt cu recursie pe coadă? şi care nu sunt rescrieţi-le.
4. Modificaţi funcţia TABEL astfel încît să afişeze toate valorile în ordine crescătoare.
5. Modificaţi TABEL astfel încît să scrie mai frumos acest tabel.
6. Scoateţi tabelul cu valorile funcţiilor trigonometrice SIN şi COS.

7. Principalele funcţii pe liste

7.1. BACKQUOTE sau ‘

Cînd o expresie este precedată de ***backquote*** (‘) este ca şi cum ar fi precedată de QUOTE (’). Dacă în faţă se află virgula (,) ***obiect***-ul respectiv este evaluat, iar dacă avem ,‘**X** atunci simbolul atomic X va fi prelucrat cu desfiinţarea parantezelor.

Exemple.

:(SETF X '(A B C))
(A B C)
:‘(X Y Z) ; acelasi efect ca si '(X Y Z)
(X Y Z)
: ‘(,x y z)
((A B C) Y Z)
: (PRINT '(X ESTE ,X))
(X ESTE (A B C))
: (PRINT '(X este , @X)
(X ESTE A B C)

7.2. Funcţii logice

NOT, NULL **(NOT obiect)** **(NULL obiect)**

Amîndouă funcţiile desemnează acelaşi lucru. Pe de o parte avem funcţia logică NOT care este ***adevărată*** numai dacă argumentul ei este NIL, pe de o parte avem predicatul NULL care este adevărat dacă argumentul lui este lista vidă NIL. Cum falsul este identificat în LISP cu lista vidă, aceste două funcţii sunt identice.

Exemple. (NULL 'A) → NIL (NOT ()) →T
Soluţie. Este adevărat ceea ce nu este NIL.
(DEFUN NOT (L) (EQUAL L NIL))

AND **(AND [forma ...])**

Funcţia AND corespunde funcţiei logice SI. Este adevărată numai şi numai dacă toate argumentele ei sunt adevărate, altfel este NIL. Funcţia sistem AND are oricîte argumente.

Exemple. (AND T T) → T (AND 'A NIL) → NIL
(AND 5 NIL T) → NIL (AND 'A 1 5 6) → 6
Soluţie.
(DEFUN AND (X Y); AND cu 2 argumente
(COND (X Y)
))
sau
(DEFUN AND (X Y) (IF X Y NIL))

OR **(OR [forma ...])**

Funcţia OR este corespunzătoare funcţiei logice OR. Are valoarea de NIL dacă toate argumentele ei sunt NIL. Are oricîte argumente.

Exemple. (OR 5 'a NIL) → 5 (OR NIL 6) → 6 (OR '(A B) T) →'(A B)
(OR 'a T T NIL) → A (OR (= 2 3) 'Bine NIL 'rau) → BINE
Soluţie. Deocamdată vom scrie OR numai pentru 2 argumente.
(DEFUN OR (X Y) ; X si Y orice obiecte LISP
(COND (X)
(Y)))
sau
(DEFUN OR (X Y) (IF X X Y))

7.3. Lipirea listelor

LIST **(LIST [obiect ...])**

Să scriem o funcţie care să întoarcă o listă cu toate argumentele sale. LIST are oricîte argumente şi este de bază în manipularea listelor.

Exemple. (LIST 'B 'C) → (B C) (LIST '(N) '(F M)) → ((N) (F M))
(LIST NIL 1 2 3) → (NIL 1 2 3) (LIST 'A '(B C) D) → (A (B C) D)

Soluţie. Deocamdată scriem LIST pentru 2 argumente.
(DEFUN LIST (X Y) ; X si Y orice obiecte LISP
(CONS X (CONS Y NIL)))

APPEND **(APPEND [lista ...])**

APPEND întoarce o listă formată din toate argumentele sale, evaluate în prealabil, ca şi cum ar şterge parantezele dintre ele.
Exemple. (APPEND '(A B) '(C D)) → (A B C D)
(APPEND '(A (B)) '(C (D)) '(E)) → (A (B) C (D) E)
(APPEND '(A (B)) NIL '(E)) → (A (B) C (D) E)
(APPEND 'A '(B C)) → *eroare A nu este lista*
Soluţie. Scriem APPEND pentru 2 argumente
(DEFUN APPEND (X Y) ;X si Y sunt **liste** neaparat!
(COND ((NULL X) Y)
(T (CONS (CAR X) (APPEND (CDR X) Y)))
))

7.4. Apartenenţa

MEMBER **(MEMBER element lista [key test])**

MEMBER înseamnă în limba engleză *aparţine*. MEMBER testează dacă un *element* aparţine sau nu unei *liste* (pe nivelul superficial).
Exemple. (MEMBER 'A '(B A C D)) → (A C D)
Soluţie.
(DEFUN MEMBER (E L) ; E= atom, L=lista
(COND ((ATOM L) NIL)
((EQ E (CAR L)) L) ; face comparatia cu EQ
(T (MEMBER E (CDR L)))
))
Funcţia oferită de sistem este de obicei mai generală, înlocuind predicatul EQ cu predicatul conţinut în *test*. Apelul în acest caz se face:
(MEMBER '(A B) '(8 (C D) (A B)) ':KEY 'EQUAL) →T
sau
(MEMBER 3 '(2 4 5 7) :KEY '>) → (2) care caută în listă un element mai mic decît 3.

7.5. Lista ca vector

NEXT **(NEXT obiect lista)**

NEXT (în limba engleză înseamnă *următorul*) scoate elementul următor din *lista* după l *obiect.*

Exemple. (NEXT 'A '(B A C D)) → C (NEXT 'A '(B A A D)) → A

Soluţie.

```
(DEFUN NEXT (E L) ; ce urmeaza dupa E in lista L
(COND ((NULL L) NIL)
      ((EQUAL E (CAR L)) (CADR L)) ; egalitate pt orice tip de obiecte
      (T (NEXT E (CDR L)))
))
```

NTH **(NTH n lista)**

Funcţia NTH scoate a-l ***n***-lea element al unei liste la nivel superficial.

Este des folosită, mai ales în ocaziile în care se tratează lista ca un vector.

Exemple. (NTH 3 '(A B C (D) E F)) → (D)

(NTH 0 '(A B C)) →A

(NTH 4 '(A B C)) → NIL

Soluţie. Scris în manieră recursivă:

```
(DEFUN NTH (N L) ;N=numar, L=lista
(COND ((= N 0) (CAR L))
      (T ( NTH (1- N) (CDR L)))
))
```

7.6. Lungimea listei

LENGTH **(LENGTH lista)**

Lungimea unei liste LENGTH întoarce numărul de elemente de pe nivelul superficial al listei.

Exemple. (LENGTH 'A) → 0 (LENGTH '(A (B D) C)) → 3

Soluţie.

```
(DEFUN LENGTH (L) ; scrisa recursiv
(COND ((ATOM L) 0)
      (T (1+ (LENGTH (CDR L))))
))
```

```
; apel: (LENGTH '(A B) 0) → 2
(DEFUN LENGTH ( L N);recursie pe coada
(COND  ((NULL L) N) ; initial contor N=0
        (T (LENGTH (CDR L) (1+  N )) );parcurg L si-l incrementez pe N
))
```

7.7. Listele de asociere

O listă de asociere (pe scurt A-listă) este o listă formată din subliste conţinînd perechi. De exemplu, ***liste de asociere*** sunt:

((1 A) (2 B) (3 C) (4 D) (5 E))
((+ PLUS) (- MINUS) (* ORI) (/ IMPARTIRE))
((UNU . ONE) (TWO . DOI) (THREE . TREI) (FOUR . PATRU))
((ROMANIA . 1) (ANGLIA . 2)(FRANTA . 3)(ARGENTINA .4))
((curs-lisp (sala-2 amfiteatru1)) (curs-structuri (sala3 sala5 sala7)))
((DAN (FOTBAL SKI BRIDGE)) (IOANA (DESEN PATINAJ))

Listele de asociere sunt structuri de date foarte des întîlinite. Merită să construim nişte funcţii speciale pentru ele: ASSOC, RASSOC.

ASSOC **(ASSOC obiect A-lista)**

(ASSOC obiect A-lista) scoate dintr-o listă de asociere perechea care are drept CAR elementul obiect.

Exemple. (ASSOC 'C '((A 1)(B 2)(C 3)(D 4))) → (C 3)
(ASSOC 3 '((1 2)(2 4)(3 8)(4 16)(3 9))) → (3 8)

Soluţie. Rescriem această funcţie sistem într-o manieră recursivă.

```
(DEFUN ASSOC (E L)
(COND ((ATOM L) NIL)
        ((EQUAL (CAAR L) E) (CAR L))
        (T (ASSOC E (CDR L)))
))
```

RASSOC **(RASSOC obiect a-lista [key test])**

Funcţia simetrică a lui ASSOC este RASSOC care scoate dintr-o A-listă acea pereche care are drept CDR pe obiect. Dacă sunt mai multe perechi cu această proprietate va scoate doar pe prima întîlnită.

Exemple.(RASSOC 'A '((3 . C)(1 . A)(2 . B))) → (1 . A)
Soluţie.Prezentăm o variantă recursivă a acestei funcţii.

```
(DEFUN ASSCDR (E L)
(COND ((ATOM L) NIL)
        ((EQUAL (CDAR L) E) (CAR L))
        (T (ASSCDR E (CDR L)))
))
```

7.8. Substituţii

SUBST **(SUBST nou vechi lista [key test])**

Deseori apare necesitatea substituirii tuturor aparițiilor unui element dintr-o listă cu alt element. Acest lucru se poate face la nivel superficial, SUBST (funcţie a sistemului CLISP), sau la orice nivel, SUBST-tot (scrisă de noi).

Exemple. (SUBST 'B 'A '(2 3 A 4 (A) 5)) → (2 3 B 4 (A) 5)
(SUBST-TOT 'N 'V '(A (B (A C) A) D)) → (1 (B (1 C) 1) D)

Soluţie.

```
(DEFUN SUBST (NOU VECHI L); substituire la nivel superficial
(COND ((NULL L) NIL)
        ((EQUAL VECHI (CAR L))
                (CONS NOU (SUBST NOU VECHI (CDR L))))
        (T (CONS (CAR L)(SUBST NOU VECHI (CDR L))))
))

; substituie pe nou in loc de vechi la orice nivel
(DEFUN SUBST-tot (nou vechi L); substituie pe nou cu vechi in L
(COND ((ATOM L) L)
        ((EQUAL VECHI (CAR L))
                (CONS NOU (SUBST-tot nou vechi (CDR L))))
        (T (CONS (SUBST-tot nou vechi (CAR L))
                 (SUBS-tot nou vechi (CDR L))))
))
```

SUBLIS **(SUBLIS A-lista lista [key test])**

Funcţia SUBLIS substituie în lista toate aparițiile elementelor din CAR-ul perechilor din A-lista cu CDR-ul corespunzător. Este o formă mai generalizată a funcţiei SUBST-tot, astfel că înfăptuieşte o serie de SUBST-tot-uri.

Exemple. (SUBLIS '((UNU . 1) (DOI . 2)) '(UNU SI DOI))) → (1 SI 2)
(SUBLIS '((a . x)(b . y)(c . z)) '(a (a)(b c) (c))) → (x (x)(y z) (z))
Soluție.
```
(DEFUN SUBLIS (A L ) ; A= ((V1 . N1)(V2 . N2)...) L=lista
(COND  ((NULL A) L)
       (T (SUBLIS (CDR A) (SUBST-tot  (CAAR  A) (CDAR  A) L )))
 ))
```

7.9. Ștergerea

REMOVE **(REMOVE obiect lista [key test])**

REMOVE șterge toate aparițiile lui ***obiect*** de pe primul nivel al unei liste. Nu distruge lista inițială, ci doar întoarce o copie modificată.

Exemple. (REMOVE 'A '(A (B A) C A)) → ((B A) C)
(REMOVE NIL '(NIL B NIL)) → (B)

Soluție.
```
(DEFUN  REMOVE (A L) ; echivalenta cu (REMOVE A L :KEY 'EQ)
(COND  ((NULL L) NIL)
       ((EQ  A (CAR L)) (REMOVE A (CDR L)))
       (T (CONS (CAR L)(REMOVE A (CDR L))))
 ))
```

Scrieți o funcție care șterge din lista L elementul al ***n***-lea.

```
(DEFUN TAKE (N L) ; N=numar L=lista
(COND  ((NULL L) NIL) ; a ajuns la capat
       ((ZEROP N)(CDR L))
       (T (CONS (CAR L) (TAKE (1- N) (CDR L)))
))
```

7.10. Inversarea listei

REVERSE **(REVERSE lista)**

REVERSE inversează elemente unei liste la nivel superficial.

Exemple. (REVERSE '(A B (C D) E)) → (E (C D) B A)
(REVERSE '(1 2 3 4)) → (4 3 2 1)

Soluție. Vom da o versiune cu recursie simplă:

```
(DEFUN REVERSE (L) ; inverseaza lista L pe nivel superficial
```

```
(COND  ((NULL L) NIL)
       (T (APPEND (REVERSE (CDR L)) (CONS (CAR L) NIL)))
 ))
; Iată REVERSE cu recursie pe coadă.
```
Exemplu. (REV '(A (B C) D) NIL)→ (D (B C) A)
```
(DEFUN  REV (L R) : L=lista R=rezultat
(COND  ((NULL L) R)
       (T (REV (CDR L) (CONS (CAR L) R)))
))
```
Să scriem o funcţie REVERSE-TOT, care inversează toate elementele unei liste indiferent de paranteze.

Exemplu. (REVERSE-TOT '(A (B C) 3 (D E))) → ((E D) 3 (C B) A)
```
(DEFUN  REVTOT (L)
(COND  ((ATOM L) L)
       (T (APPEND (REVTOT (CDR L)) (LIST (REVTOT (CAR L)))))
 ))
```

Exerciţii. 7. Principalele funcţii pe liste

1. Completaţi evaluările:

X	Y	CONS	LIST	APPEND
A	(B C)			eroare
(A B)	(C)		((A B) (C))	
(A)	(B)	((A) B)		
NIL	(A B)	(NIL A B)		
(A B)	NIL		((A B) NIL)	
(A B)	(C D)	((A B) C D)		(A B C D)

2. Verificaţi dacă o listă este A-listă.
3. Imaginaţi în LISP lucru cu o stivă FIFO (First In First Out) , FILO (First In Last Out) cu funcţiile PUSH şi POP.
4. Scrieţi (PUT K N L) care inserează în lista L pe locul N pe K.
5. Scrieţi funcţia (POSITION element lista) care întoarce poziţia pe care se află prima apariţie a lui ***element*** în ***lista***.
   Avem: (POSITION 'A '(B A (C A) A)) → 1
6. Scrieţi funcţia NTHCDR aplică un număr de N ori funcţia CDR pe o listă dată L. Avem: (NTHCDR 4 '(A B C D E F)) → (E F)
7. Rescrieţi funcţiile din acest capitol folosind condiţionalul IF.
8. Comentaţi tipurile de recursie la toate funcţiile întîlnite.

8. Totul despre funcţii

8.1. Notaţia LAMBDA

Limbajul LISP s-a inspirat din modelul de calcul propus de matematicainul englez Alonzo Church (n.1903). Modelul lui de calcul numit **lambda calcul** nu este altceva decît un alt model de calculabilitate echivalent cu funcţiile recursive, maşina Turing (A.M. Turing, 1912-1954), algoritmii Marcov, etc /33/.

Cum am putea reprezenta dintr-o dată şi corpul funcţiei şi modul cum se aplică ?

Astfel dacă scriem **f(x) = x + 5** şi **g(y) = 2 y** avem două funcţii. Dacă vrem să aplicăm pe **f** lui **x = 4.** Scriem **f(4).**

Dacă vrem să compunem pe **g** cu **f** în punctul **x = 3** scriem **f (g (3))** .

Notaţia lambda calculului ne oferă o reprezentare mai bună. Nu trebuie să născocim nici o denumire pentru prima funcţie şi nici pentru a doua. Avem în notaţia lambda calculului:

f(x) = x+5	(lambda(x)(x + 5))
f(4)	(lambda(x)(x + 5)) 4)
g(y) = 2*y	(lambda(y)(2 * y))
g(3)	((lambda(z)(2 * z)) 3)
f(g(3))	((lambda(x)(x + 5))((lambda(z)(2 * z)) 3))
h(x,y) = x + y + xy	((lambda(x y)(x + y + x *y))

h(3,8) ((lambda(x y)(x + y + x*y)) 3 8)

Observăm că **x** se va lega de **3** şi **y** de **8**. Deci asocierea se face în ordinea în care apar argumentele, respectiv valorile.

În limbajul LISP putem lucra cu funcţii fără nume. Astfel pe primul loc al unei liste dacă se află o ***lambda expresie*** interpretorul o înţelege şi după ce leagă variabilele încearcă s-o execute. Deci un mod de a lega o variabilă cu o valoare este cu ajutorul notaţiei LAMBDA.

Reluăm în LISP exemplele dinainte:

f(x)=x+5	(lambda(x)(+ x 5)) → *Eroare lambda nelegată!*
f(4)	((lambda(x)(+ x 5)) 4) → 9
g(y)=2*y	(lambda(y)(* 2 y))
g(3)	((lambda(z)(* 2 z)) 3) → 6
f(g(3))	((lambda(x)(+ x 5))((lambda(z)(* 2 z))3)) → 11
h(x,y)=x+y+xy	(lambda(x y)(+ x y (* x y)))
h(3,8)	((lambda(x y)(+ x y (* x y))) 3 8) → 35

Exerciţii. Completaţi evaluările:

((LAMBDA(Z)(* Z Z Z)) 3) →

((LAMBDA(X)(CONS X X)) '(A B)) → ((A B) A B)

((LAMBDA(Z) 1) 'ORICE) → 1

((LAMBDA () 8)) → 8

((LAMBDA(Z) (+ Z 3)) 7) →

((LAMBDA(X Y) (+ X Y (* X Y))) 1 2)) →

((LAMBDA(X)(+ X 3 5)) ((LAMBDA(Z)(* 2 Z)) 1)) →

8.2. LET şi LABELS

LET **(LET ([var | (var val)] ...) [form ...])**

Se evaluează în mod secvenţial expresiile: ***form*** ... Este întoarsă valoarea ultimei expresii. Evaluarea se face cu legarea variabilelor din lista de declaraţie a lui LET. În cazul în care lipseşte *val* se ia drept valoare implicită pe NIL. Variabilele sunt legate în paralel şi sunt LOCALE ! Să adunăm X cu Y, unde X=5 şi Y=3. Avem:

(LET ((X 5) (Y 3)) *; X devine 5 si Y devine 3*
(+ X Y) *; corpul lui LET*
) → 8 *; întoarce ultimul lucru evaluat*

Exemple.
(LET () (PRINT 'CUCU)) → CUCU
(LET (X) (CONS 2 NIL)) → (2)
(LET ((X 'A)) (CONS X NIL)) → (A)
(LET ((X 'A)(Y 'B)) (CONS X Y)) → (A . B)
(LET ((X 'A)(Y X)) (CONS X Y)) → *eroare X nu este legata*

LET* **(LET* ([var | (var val)] ...) [form ...])**
Exact ca şi LET, cu diferenţa că legarea variabilelor se face secvenţial.
Exemple. (LET* ((X 'A)(Y X)) (CONS X Y)) → (A . A)
(LET* ((X 'A)(Y X)(Z Y))
(LIST X Y Z)) → (A A A)

LABELS **(LABELS (nume-functie lista-arg [corp..][form ...])**
În cazul în care avem nevoie de o funcţie locală care va apare numai într-un singur loc, nu este nevoie să o definim prin *DEFUN*. Am putea desigur să o scriem direct ca o lambda expresie, dar să presupunem că funcţia trebuie definită recursiv şi atunci trebuie să aibe un nume. LABELS va fi deci un DEFUN local.
Exemplu. (LABELS ((ADUN5 (Z) ; definesc functia locala ADUN5
(+ 5 Z))) ; corpul lui ADUN5
(ADUN5 10)) → 15 ; apelez ADUN5
După ce părăsesc (LABELS...) nu se mai ştie cine este ADUN5.
Alt exemplu: REVERSE scrisă cu ajutorul unei alte funcţii REV
(DEFUN REVERSE (L) ; L =lista de inversat R = lista rezultat
(LABELS ((REV (L R) ; recursie pe coada
(COND ((NULL L) R)
(T (REV (CDR L)(CONS (CAR L) R)))
)))
(REV L NIL)
))

S-a definit o funcţie locală REV, prin LABELS chiar în corpul funcţiei REVERSE. REV are două argumente al doilea fiind variabila colectoare R, care reţine valoarea finală. Este un caz de recursie pe coadă. Labels întoarce ultima valoare din formele evaluate în corpul său.
Iată **lungimea unei liste** scrisă cu recursie pe coadă.

Exemplu. (LENGTH '(A B (C D))) →3

```
(DEFUN LENGTH(L) ; lungimea unei liste scrisa recursiv
(LABELS ((L-AUX (L N) ; N=contor, initial= 0
(COND  ((NULL L) N)
        (T (L-AUX (CDR L) (1+ N)))
 )))
 (L-AUX L 0) ; N=contor se initializeaza la 0
))
```

8.3. Tipuri de funcţii

Funcţiile şi compunerile de funcţii stau la baza limbajului de programare funcţională LISP. Sunt trei mari criterii după care se împart funcţiile în categorii:

- număr de argumente: poate fi *fix* sau *variabil*
- evaluarea argumentelor: se pot evalua sau nu
- mod-de apel: pot fi compilate sau nu. În cazul interpretorului de faţă, acest criteriu împarte funcţiile în funcţii utilizator şi funcţii sistem.

Exemple.

- Funcţii sistem cu număr fix de argumente cu evaluarea argumentelor: ATOM, CAR, CDR, CONS, LENGTH...
- Funcţii sistem cu număr variabil de argumente cu evaluarea lor. LIST, APPEND, OR, AND, + , * , - , ..
- Funcţii cu număr fix de argumente fără evaluare.
- Funcţii cu număr variabil de argumente fără evaluare.

FUNCTIONP **(FUNCTIONP atom)**

Este un predicat cu valoarea T dacă atom este numele unei funcţii.

Exemplu. (FUNCTIONP 'DUBLU) → T (FUNCTIONP 'LIST)→ T

8.4. Definirea funcţiilor

(DEFUN nume-funcţie lista-arg corp)

Pînă acum am prezentat construirea cu DEFUN simplu a funcţiilor cu număr fix de argumente care sunt evaluate pe rînd. Prezentăm toate alternativele oferite de funcţia sistem DEFUN.

nume-functie - este numele functiei, trebuie să fie un atom simbolic

corp - corpul funcţiei

lista-arg - poate fi de forma

- cu număr fix de argumente: este cazul cel mai simplu
- cu argumente opţionale: se declară cu &OPTIONAL arg1 arg2...
- cu oricîte argumente: se declară cu &REST arg1 arg2...
- cu variabile auxiliare: se declară cu &AUX arg1 arg2...

Argumente opţionale **&OPTIONAL arg**

Putem să declarăm variabile care la apelul funcţiei pot sau nu să fie enunţate. În cazul în care nu sunt, se iau în considerare ca şi cum ar fi NIL.

Exemplu. **YES-OR-NO-P**

Foarte des apare situaţia în care trebuie să întrebăm utilizatorul ceva care aşteaptă răspunsul DA sau NU. Să scriem un predicat YES-OR-NO-P care afişează o întrebare şi pentru răspunsul **DA** al utilizatorului este T.

:(YES-OR-NO-P "Vreti sa continuati")
Vreti sa continuati (Y/N)?= > **NU**
NIL
:(YES-OR-NO-P)
Y/N?= **DA**
T

```
(DEFUN YES-OR-NO-P (&OPTIONAL TEXT)
(IF TEXT (PRINC TEXT)) (PRINC "(Y/N)?= ")
(MEMBER (READ) '(Y YES DA D OK T TRUE))    )
```

Argumente auxiliare **&AUX**

Astfel putem avea variabile auxiliare. Acestea sunt de fapt *variabile locale* care ar putea fi definite şi printr-un LET. Avem

```
(DEFUN LENGTH1  (L)
(LET (( LUNG 0))
(DOLIST (I L)
        (SETF LUNG (1+ LUNG))
) LUNG))
```

Echivalent cu

```
(DEFUN LENGTH2  (LISTA &AUX (LUNG 0))
(DOLIST (I L)
        (SETF LUNG (1+ LUNG))
)  LUNG))
```

La apelul funcţiilor variabilele după &AUX nu se iau în considerare:

```
:(LENGTH2   '(A B C))
 3
```

Oricîte argumente **&REST arg**

În acest mod putem defini funcţii cu oricîte argumente.
Ele se evaluează pe rînd. Să dăm cîteva exemple.

FUNCALL **(FUNCALL fun arg1 arg 2 ...)**

Aplică pe funcţia ***fun*** lui ***arg1 arg2***... Argumentele sunt evaluate înainte de aplicarea lor.

Exemplu. (FUNCALL '+ 1 2 3 4) → 10 (FUNCALL 'LIST 'A) → (A)
(SETF X 1 Y 2 Z 3) (FUNCALL '* X Y Z) → 6

Soluţie. Iată cam cum ar funcţiona FUNCALL:

```
(DEFUN FUN3 (F &REST L) ; aplica functia F argumentelor
(EVAL (CONS  F L))          ; L- lista argumentelor
)
```

Să extindem funcţiile **MIN şi MAX la oricîte argumente**:

(MIN 2 3 4 5 6 1 9) → 1 (MAX 5 4 3 2 1 8) → 8

```
(DEFUN MIN (&REST L)
(COND  ((NULL L) 'EROARE)
       ((NULL (CDR L))  (CAR L))
       (T (MIN2 (CAR L)(APPLY 'MIN (CDR L))))
 ))
```

Comentariu. Variabila L este o listă. Dacă am fi scris:
(MIN (CDR N)) ar fi fost greşit, însemnînd (MIN '(4 3 2 5)), dar funcţia MIN vrem să o definim cu oricîte argumente şi nu cu un singur argument listă.

Funcţia **LIST cu oricîte argumente**:

```
(DEFUN LIST (&REST L)
(COND ((NULL L) NIL)
       ((NULL (CDR L)) L)
       (T (CONS (CAR L) (APPLY  'LIST (CDR L))))
 ))
```

OR cu oricîte argumente:

```
(DEFUN OR (&REST L)
(COND ((NULL L ) NIL)
       ((CAR L) T)
       ((APPLY 'OR (CDR L)))
))
```

APPEND cu oricîte argumente (avem APPEND2 cu 2 argumente).

Exemplu. (APPEND '(A B) '(C) '(H J) '(1 2)) → (A B C H J 1 2)

```
(DEFUN APPEND (&REST L) ; generalizare APPEND cu 2 argumente
(COND  ((NULL L) NIL)
       (T (APPEND (CAR L) (APPLY  'APPEND2 (CDR L))))
))
```

DEFMACRO **(DEFMACRO nume-functie (arg) corp)**

Funcţiile denumite MACRO -uri, întîi expandează şi apoi evaluează.
Macro-urile pot fi cu număr fix sau variabil de argumente.
Exemplu. LOOP care evaluează pe *corp* pînă cînd *test* este adevărat

```
:(LOOP (SETQ X (READ)) (PRINT (* 2 X)) )
4
8
NIL
NIL
```

Iată cum scriem pe LOOP ca un MACRO.

```
(DEFMACRO LOOP (TEST &REST CORP) ; LOOP cu testare
'(DO () (,TEST)
        ,@CORP) )
; Alt LOOP fără test care evaluează pe corp pînă la o ieşire forţată
(DEFMACRO LOOP(&REST CORP);cicleaza pe CORP fara TEST
'(DO () (NIL)
        ,@CORP))
```

IF **(IF test forma1 forma2)**

Funcţia IF selectează *forma1* dacă efectuarea lui test este diferită de NIL, altfel selectează şi execută *forma2*.
Exemple. (IF T (PRINT 'BINE) (PRINT 'RAU)) → BINE
Soluţie. Iată o variantă de a-l scrie pe IF ca un MACRO.

```
(DEFMACRO IF (TEST &OPTIONAL COND1 COND2)
'(COND (,TEST ,COND1)
       (T ,COND2)
))
```

UNLESS **(UNLESS test [forma ...])**

Funcţia UNLESS funcţionează ca un IF pe dos. Dacă condiţia din *test* este falsă (= NIL), se vor evalua pe rînd formele.

Soluţie.

```
(DEFMACRO UNLESS (TEST &REST FORMA)
'(COND ((NOT ,TEST) ,@FORMA))
))
```

WHEN **(WHEN test [forma ...])**

Scrieţi funcţia WHEN similară cu UNLESS care în urma evaluării testului şi a rezultatului diferit de NIL, va efectua formele care urmează (ca într-o secvenţă PROGN).

Exemple.

***:* (WHEN (EVENP 8) (PRINC "N-rul=")(PRINC 8) (PRINC " e par"))**

N-rul=8 e par

Soluţie.

```
(DEFMACRO WHEN (TEST &REST FORME)
'(COND (, TEST ,@FORME)) )
```

8.5. Funcţii chirurgicale

Se numesc ***funcţii chirurgicale*** sau ***distructive*** deoarece modifică lista argument. Aceste funcţii trebuie să fie folosite cu grijă de către începători. Ele sunt extrem de folositoare în ceea ce priveşte economia de memorie.

Observăm în CLISP multe funcţii care încep cu N: NREVERSE, NSUBST, NSUBLIS, ... Ele sunt variantele ***distructive*** ale funcţiilor.

RPLACA **(RPLACA cons nou)**

RPLACA are două argumente pe care le evaluează. Întoarce lista *cons* căreia îi modifică CAR-ul (primul element) prin substituirea lui ***nou***.

Exemple. (SETF L '(A B C)) → (A B C)

(RPLACA L 'D) → (D B C) L → (D B C)

Iată o funcţie similară cu RPLACA, dar care nu alterează pe *cons:*

```
(DEFUN RPLACA (CONS NOU)
(CONS NOU (CDR CONS))
)
```

RPLACD **(RPLACD cons nou)**

Are două argumente pe care şi le evaluează. Distruge pe *cons* înlocuindu-i CDR-ul cu nou.

Exemple. (SETF L '(A B C)) → (A B C)

(RPLACD L '(M)) → (A M) L → (A M)

Iată o funcţie similară cu RPLACD, dar fără distrugere.

```
(DEFUN RPLACD(cons nou)
        (CONS (CAR cons) nou)
        )
```

NCONC **(NCONC [list ...])**

NCONC este o variantă de APPEND distructivă. Întoarce concatenarea listelor modificînd valoarea primei liste.

Exemplu. (SETF X '(A B) Y '(C D)) → (C D)

(NCONC X Y) → (A B C D)

X → (A B C D)

Y → (C D)

9. Iteraţia

Ca şi recursivitatea, iteraţia este *o strategie de control a programului.* Am văzut la *Capitolul 6. Recursivitatea* că putem defini o funcţie cu ajutorul ei înşăşi dacă avem o condiţie de oprire şi o relaţie între valorile funcţiei. Am insistat mai mult asupra recursivităţii deoarece deobicei studenţii o practică mai rar. Mai populară şi iubită este tehnica iteraţiei. Putem defini o funcţie de la început, fără ca numele ei să mai apară în definiţie. Ne vom folosi de diferite variabile care cresc, contoare şi condiţii de oprire. Pentru ilustrare să începem cu prezentarea iteraţiei cu DO: DOLIST, DOTIMES, DO.

9.1. Sructura DO

DOLIST **(DOLIST (var lista [rez]) [corp...])**

Vrem să scriem pe rînd elementele unei liste (A B C).

:(DOLIST (I '(A B C)) (PRINT I))

A

B

C

NIL

Am scris ceva în genul:

(DOLIST (var lista) ; *I = variabila-contor var care a luat*

corp) *; pe rind ca valoare elementele din lista la nivel superficial*

Deci DOLIST parcurge (iterează) lista *list*, legînd pe rînd variabila *var* la fiecare element din *list* de pe nivel superficial. Argumente:
var - numele variabilei
lista - lista de parcurs
rez - expresia care se întoarce drept rezultat
corp - execută de atitea ori cît este lungimea lui *list*.
Întoarce valoarea expresiei rezultat *rez* sau NIL
Exemple. Lungimea unei liste:

```
(DEFUN LENGTH (L &AUX (REZ 0)) ; REZ variabila auxiliara
(DOLIST (I L REZ) ;  REZ va fi valoarea intoarsa ca rezultat
        (SETF REZ (1+ REZ))
))
```

(SOME predicat lista)
Funcţia SOME ne spune dacă există vreun element din *lista* care are predicatul *predicat*.
Exemplu. Ca să vedem dacă există în *lista* un număr:
(SOME 'NUMBERP '(A B C 3)) → T
Ca să vedem dacă lista (A B C 3) are un număr mai mare ca 5 în lista:
(SOME '(LAMBDA(Z)(AND (NUMBERP Z) (> Z 5))) '(A B C 3)) → NIL
Soluţie. Observăm că se poate ieşi din DO cu RETURN.

```
(DEFUN SOME (P L) ; verifica daca un element din L are predicatul P
(DOLIST (I L)
        (IF (FUNCALL P I) (RETURN T))
))
```

DOTIMES **(DOTIMES (var contor [rez]) [corp ...])**
Să scriem numerele de la 0 la N (N=3). Avem:

```
:(DOTIMES (I  3) (PRINT I))
0
1
2
NIL
```

Am scris ceva în genul:
(DOTIMES (VAR NUMAR) ; *VAR ia valori pe rind de la 0 la NUMAR*
CORP)
Deci DOTIMES: iterează (repetă) corpul *corp* dînd pe rînd valori variabilei *var* de la zero la (contor − 1).
var - numele variabilei
număr - valoarea finală (număr întreg), pînă la care se ajunge
rez - expresia care se întoarce (rezultat), în lipsă se întoarce NIL
corp - corpul lui DOTIMES se repetă pînă cînd *var* = *număr*

Iată alt exemplu de scriere pentru binecunoscutul FACTORIAL:
FACTORIAL scris cu DOTIMES:

```
(DEFUN FACTORIAL (N &AUX (REZ 1))
(DOTIMES (I N REZ)
        (SETF REZ (* REZ (1+ I)))
))
```

Formez o listă cu numere pînă la N **(MAKE-LIST număr)**
Iată o funcţie care întoarce numerele de la 1 la N într-o listă în ordine.
Exemplu. (MAKE-LIST 5) → (1 2 3 4 5)

```
(DEFUN MAKE-LIST (N &AUX REZ)
(DOTIMES (I N REZ) ; I = 0, 1,2 ... N-1
        (SETF REZ (CONS (- N I) REZ))
))
```

Suma primelor N numere
Să calculăm cu DOTIMES suma primelor N numere naturale.
Exemplu. (SUMA-PRIMELOR-N-NUMERE 10) → 45

```
(DEFUN SUMA-PRIMELOR-N-NUMERE (N &AUX (REZ 0) )
(DOTIMES (I N REZ) ; REZ= rezultatul
        (SETF REZ (+ REZ I))
))
```

DO **(DO ([(var [init [pas]] ...)(test [rez...])[corp...])**

Să încercăm să generalizăm pe DOLIST şi DOTIMES. Obţinem pe DO: iteraţia generalizată.

Să scriem pe rînd elementele unei liste:

```
: (DO ( (Z '(A B C) (CDR Z)) )  ((NULL Z) 'GATA)
(PRINT (CAR Z))
)
A
B
C
GATA
```

Am scris ceva în genul:

(DO ((var1 init1 pas1)) (test rez)
corp
)

Argumentele lui DO sunt:

var - variabile pe care se face iteraţie (pot fi mai multe)
init - este valoarea iniţială a lui *var*
pas - pasul iteraţiei lui *var. var* devine *var* = *pas*
test - expresia de test
rez - forma care trebuie întoarsă drept rezultat
corp - corpul buclei care se repetă

Întoarce valoarea ultimei expresii din *rez*ultat. Se opreşte ciclarea cînd *test* este adevărat. Variabilele din buclă sunt incrementate în paralel! Formele din *pas* sunt evaluate înaintea celorlalte.

Implicit: *init* este NIL. Lipsa lui *pas* nu schimbă valoarea variabilei.

Dacă nu există nici o expresie *rez* DO va întoarce NIL.

Lungimea unei liste scrisă cu DO

```
(DEFUN LENGTH (L); lungimea unei liste
(DO ((Z L (CDR Z)) (REZ 1 (1+ REZ))) ((NULL Z) (1- REZ))
))
```

Numerotarea elementelor unei liste **(NUMEROTEZ lista)**

Să numerotăm elementele unei liste la nivel superficial adăugînd numărul în faţa elementului, în liste pereche.

Exemplu. (NUMEROTEZ '(A B C)) → ((1 A)(2 B)(3 C))

Soluţie.

```
(DEFUN NUMEROTEZ (L &AUX REZ)
(DO ((N 1 (1+ N)) (Z L (CDR Z))) ((NULL Z) (REVERSE REZ))
        (SETF REZ (CONS (LIST N (CAR Z)) REZ))
 ))
```

Să reluăm funcţia **SUBLIS** care substituie în ***lista*** toate apariţiile elementelor din CAR-ul perechilor din A-lista cu CDR-ul corespunzător (pe nivel superficial).

Exemple. (SUBLIS '((A . 1)(B . 2)(C . 3)) '(A (A)(B) (C))) → (1 (1)(2)(3))

```
(DEFUN SUBLIS  (A L &AUX (REZ L)) ; A=A-lista , L=lista ; REZ=L
(DOLIST (I A REZ)
        (SETF  REZ (SUBST (CDR I)(CAR I) REZ))
))
```

9.2. Structura PROG

GO **(GO eticheta)**

Sare la **eticheta**. Este recunoscută numai în structurile care admit etichete: PROG, PROG*, PROGN, PROG1. După cum urmează.

PROG, PROG* **(PROG ([var | (var [val]) ...]) [form ...])**

PROG permite în cadrul ei: declararea locală de variabile în mod secvenţial; evaluarea secvenţială a expresiilor ***form***... , combinată cu folosirea lui RETURN pentru ieşire imediată; folosirea lui GO şi a etichetelor.

var - numele variabilelor locale

val - valoarea asociată (în lipsă se consideră valoarea NIL)

PROG întoarce valoarea ultimei expresii evaluate sau ceea ce se explicitează prin RETURN.

Spre deosebire de PROG , PROG* setează variabilele în paralel.
Exemple. (PROG () (+ 8 5)) → NIL
(PROG () (PRINC 'bine)(PRINC 'ai)(PRINC 'venit)) → BINE AI VENIT
(PROG (X Y) (SETF X 1) (SETF Y 2)) → NIL iar X şi Y sunt locale

Funcţia **FACTORIAL** scrisă cu PROG:

```
(DEFUN FACTORIAL (N)
(PROG ((I 0)(FACT 1)) ; initializam I=0 si FACT=1
ETICHETA
(SETF I (1+ I) FACT (* FACT I))
(IF (> N I) (RETURN FACT)) ; daca I depaseste pe N se intoarce FACT
(GO ETICHETA)
))
```

Funcţia **LENGTH** scrisă cu PROG:

```
(DEFUN LENGTH (L) ; lungimea listei L
(PROG ((N 0)) ;se initializeaza variabila locala N cu 0
(DOLIST (I L)
        (SETF N (1+ N))
) (RETURN N))
```

PROGN, PROG1 **(PROGN [form...]) (PROG1 form1 [form ...])**
La fel ca PROG. Evaluează pe rînd formele din corp. Întorc:
PROGN - ultima formă evaluată PROG1 - valoarea formei ***form1***
Admit (RETURN) precum şi salt (GO eticheta).
Exemple. (PROGN (SETF X 1)(SETF Y 2)) → 2
(PROG1 (SETF X 1)(SETF Y 2)) → 2

RETURN **(RETURN [val])**
Este ieşirea forţată. Poate fi întîlnită într-o structură de control care admite RETURN, iese întorcînd drept valoare pe ***val***. Dacă nu există *val* întoarce pe NIL. Din structurile care permit RETURN avem: DOTIMES, DOLIST, DO, PROG, PROG*, PROGN, PROG1, BLOCK.

BLOCK **(BLOCK nume [forma])**

Creează o structură care admite RETURN sau RETURN-FROM. Întoarce numele block-ului.

9.3. MAP-ări

MAPCAR **(MAPCAR func lista [lista ...])**

MAPCAR este o funcţie de tip structură de control iterativ. Ca şi DO ne ajută să parcurgem o listă de la început pînă la sfîrşit, aplicînd funcţia *func* pe fiecare argument. Intoarce lista de la început în care s-a înlocuit pe fiecare poziţie rezultatul aplicării funcţiei *func*.

Exemple. (MAPCAR 'ATOM '(A (B C) D (J K))) → (T NIL T NIL)
(MAPCAR 'FACTORIAL '(1 2 3)) → (1 2 9)
(MAPCAR '(LAMBDA(Z)(* Z 10)) '(1 2 3)) → (10 20 30)
(MAPCAR '(LAMBDA (Z)(IF (NUMBERP Z) 1 0)) '(5 (1 C) D))→(1 0 0)

Iată o descriere în LISP a unui MAPCAR cu un argument:

```
(DEFUN MAPCAR (F L); F= functie de un argument, L= o lista
(COND ((NULL L) NIL)
      (T (CONS (FUNCALL F(CAR L)) (MAPCAR (CDR L) F)))
))
```

In continuare cîteva cazuri în care MAPCAR s-a dovedit foarte bun.

PAIRLIS **(PAIRLIS lista1 lista2)**

Să concepem o funcţie care construieşte o listă de asociere (A-listă) din 2 liste date, de lungime egală. Ce se întîmplă dacă nu au lungimi egale ?

Exemplu. (PAIRLIS '(A B C) '(1 2 3)) → ((A . 1)(B . 2)(C . 3))

```
(DEFUN PAIRLIS (X Y) (MAPCAR 'CONS X Y))
```

Punerea de paranteze **(PUN-PARANTEZE lista)**

Să punem încă un rînd de paranteze peste elementele unei liste date.

Exemplu. (PUN-PARANTEZE '(A (B C) D E)) → ((A) ((B C)) (D) (E))

```
(DEFUN PUN-PARANTEZE (L) (MAPCAR 'LIST L))
```

Adunare şi înmulţire de vectori

Vectorii îi vedem ca liste. Să scriem funcţiile de adunare şi înmulţire a doi vectori.

Exemple. (+VECTORI '(1 2 3) '(1 1 1)) → (2 3 4)

(*VECTOR '(1 2 3) '(2 2 2)) → (2 4 6)

```
(DEFUN +VECTORI (X Y) (MAPCAR '+ X Y))
(DEFUN *VECTORI (X Y) (MAPCAR '* X Y))
```

Numerotarea elementelor unei liste **(NUMEROTEZ lista)**

Reluăm exemplul cu numerotarea elementelor unei liste (vezi exemple DO). Amintim că

(NUMEROTEZ '(A B C)) → ((1 A)(2 B)(3 C))

Soluţie.

```
(DEFUN NUMEROTEZ (L &AUX (N 0))
(MAPCAR  '(LAMBDA(Z)(LIST (SETF N (1+ N)) Z)) L))
```

LENGTH - lungimea unei liste. O altă definiţie posibilă. Folosim principiul răbojului, numerotăm elementele

```
(DEFUN LENGTH (L) ; L =lista
(LET ((N 0)) ; N contorul =0
(IF L (CAR (LAST (MAPCAR '(LAMBDA (Z) (SETF N (1+  N))) L)))
) ))
```

Inversează totul **(REVERSETOT lista)**

Funcţia REVERSE, funcţie oferită de sistem întoarce lista pe dos, dar numai la nivel superficial. Să concepem o funcţie care să întoarcă lista ca într-o oglindă neţinînd cont de paranteze (respectiv de niveluri).

Exemplu. (REVERSETOT '(a (b c) d (e (f g)))) → (((g f) e) d (c b) a)

```
(DEFUN REVERSE-TOT (L)
(MAPCAR '(LAMBDA(Z) (IF(ATOM Z) Z (REVERSE-TOT Z)))
(REVERSE L)) )
```

MAPC **(MAPC func lista [lista ...])**

Spre deosebire de MAPCAR, funcţia MAPC parcurge lista de argumente, aplică funcţia, dar nu construieşte o nouă listă, în care să întoarcă rezultatul.

O folosim în cazul în care nu ne interesează captarea rezultatelor. Rezultatul întors este ultima listă. Avem:

:(MAPC 'PRINT '(A (B C) D))

A

(B C)

D

(A (B C) D) ;întoarce drept rezultat ultima lista.

MAPCARN **(MAPCARN func lista [lista ...])**

Funcţia MAPCARN procedează întocmai ca şi MAPCAR cu deosebirea că va şterge NIL-urile din lista finală.

Exemple. (MAPCARN 'ATOM '(A (B C) 4)) → (T T)

(MAPCARN '= '(2 3 4) '(3 5 4)) → (T)

Soluţie.

```
(DEFMACRO MAPCARN (F &REST L)
(LIST 'REMOVE NIL '(MAPCAR ,F ,@L))
)
```

MAPCARN este definită ca un MACRO. După cum am văzut MACRO-urile întîi expandează şi apoi evaluează. Prin @L am obţinut înlocuirea lui L în listă cu desfiinţarea parentezelor astfel:

(MAPCAR F arg1 arg2 ...) şi nu (MAPCAR F (arg1 arg2 ...)), care ar fi fost greşit.

SELECTarea atomilor cu o anumită proprietate **(SELECT L P)**

Să selectăm dintr-o listă toţi atomii de pe primul nivel care au proprietatea P.

Exemplu. (SELECT '(A B 1 2 C) 'NUMBERP) → (1 2)

```
(DEFUN SELECT (L P) ;atomii din L care au T la predicatul P
(MAPCARN '(LAMBDA(Z)(IF (FUNCALL P Z) Z)) L)
)
```

GENCAR **(GENCAR func lista [lista ...])**

Funcţia GENCAR se comportă întocmai ca şi MAPCAR cu deosebirea că se opreşte la primul NIL întîlnit prin aplicarea funcţiei ***func***.

Exemple. (GENCAR 'ATOM '(A B (C D) E)) → (T T)
(GENCAR '= '(2 3 5) '(2 4 5)) → (T)
Soluţie.

```
(DEFMACRO GENCAR (F &REST L)
'(LET ((L (MAPCAR ,F ,@L)) REZ)
(DOLIST (I L REZ)
        (IF I (SETF REZ (APPEND REZ (LIST I))) (RETURN REZ)) )
))
```

MAPCAN **(MAPCAN func lista [lista ...])**

Aplică succesiv elementelor listei pe ***func***, la fel ca şi MAPCAR, dar apoi aplică pe APPEND (concatenează elementele listei rezultat). Se poate folosi şi NCONC, dar atunci se va modifica ***lista*** cu distrugere!
Exemple. (MAPCAN 'CDR '((A 1) (B 2) (C 3))) → (1 2 3)
Soluţie.

```
(DEFMACRO MAPCAN (F &REST X)
'(APPLY #'APPEND (MAPCAR ,F ,@X)) )
```

Funcţia **SCOT-PARANTEZE** cu ajutorul funcţiei MAPCAN:
Exemplu. (SCOT-PARANTEZE '((A (B 8)) (2))) → (A B 8 2)

```
(DEFUN SCOT-PARANTEZE (L) (MAPCAN '(LAMBDA (Z) Z) L) )
```

Să **INSER**ăm un obiect în faţa fiecărui element al unei liste.
Exemple. (INSER 1 '(A B C)) → (1 A 1 B 1 C)

```
(DEFUN INSER (A L)  (MAPCAN  '(LAMBDA(Z) (LIST A Z)) L))
```

9.4. Problema FUNARG

FUNCTION **(FUNCTION nume-functie)** sau **#'nume-functie**

Pentru funcţiile utilizator întoarce într-o listă lambda expresia ataşată. Pentru funcţiile sistemului întoarce codul intern al numelui funcţiei. FUNCTION este o funcţie care nu-şi evaluează argumentul!

Exemplu. (DEFUN DUBLU (N) (* 2 N))
(FUNCTION dublu) → ((LAMBDA(N)(BLOCK (* 2 N))))
Din păcate lucruirle nu stau aşa de simplu. Apare problema **FUNARG** sau a argumentelor funcţionale. Apar situaţii în care se confundă variabilele dintr-un context cu cele din alt context (dacă într-unul din contexte sunt libere). Să luăm un exemplu din cartea lui I.Streinu /35/.

(SETF X NIL)
(DEFUN $MAPCAR (F X); MAPCAR scris de noi de aceea l-am notat cu $
(COND ((NULL X) NIL)
(T (CONS (FUNCALL F (CAR X)) (MAPCAR F (CDR X))))
))
Observăm următoarea ciudăţenie:
($MAPCAR '(LAMBDA (Y)(CONS Y X)) '(A B)) → ((A A B) (B A B))

Se confundă variabila liberă X care ar trebui să fie NIL (pe care am declarat-o global aşa cu SETF) cu variabila X din corpul lui $MAPCAR. Deci cînd apelăm pe $MAPCAR variabila X este (A B). Aceasta este valoarea pe care o găseşte evaluatorul în dreptul lui X.
Pentru a evita confundarea variabilelor vom introduce pe FUNCTION:
($MAPCAR(FUNCTION (LAMBDA(Y)(CONS Y X))) '(A B)) → ((A)(B))

Forma prescurtată a lui **(FUNCTION nume-functie)** este **#'nume-functie.**
Deci pentru linia de mai înainte scriem mai scurt:
($MAPCAR #'(LAMBDA(Y) (CONS Y X)) '(A B)) → ((A) (B))

E bine să folosim tot timpul cînd aplicăm funcţii pe FUNCTION pentru a evita surprizele, deşi nu este întotdeauna necesar. Deci vă sfătuiesc să vă obişnuiţi să scrieţi în loc de:
(MAPCAR 'LIST '(A B)) pe (MAPCAR #'LIST '(A B))

Exerciţii. 9. Iteraţia

1. Scrieţi în 5 moduri funcţia LENGTH.
2. Scrieţi în 3 moduri funcţia FACTORIAL.
3. Scrieţi AND cu oricîte argumente.
4. Presupunem că avem funcţia = definită numai pentru două argumente numerice. Extindeţi-o la oricîte argumente.
5. Scrieţi DOLIST şi DOTIMES ca un MACRO cunoscînd pe DO.
6. Reluaţi *Principalele funcţii pe liste* şi scrieţi-le iterativ.
7. Observaţi în carte locurile unde ar putea apare problema FUNARG.

10. Liste de priorităţi

10.1. Lucru cu proprietăţi pe atom

Fiecărui *atom simbolic (*îl notăm cu *sym)* putem să-i ataşăm proprietăţi cu valorile respective. Proprietăţile vor fi denumite tot prin atomi simbolici, pe cînd valorile pot fi orice obiecte LISP. De exemplu să luăm atomul *MELANIA* . Avem:

Atom MELANIA

Proprietate	*Valoare*
CLASA	2
DATA-NAŞTERII	(13 11 1989)
MEDIE	9.80
ORAŞ	BUCUREŞTI

Obţinem acest efect în LISP scriind:
(PUTPROP 'MELANIA 'CLASA 2) → 2
(PUTPROP 'MELANIA 'DATA-NAŞTERII '(13 11 1989) → (13 11 1989)
(PUTPROP 'MELANIA 'MEDIE 9.80) → 9.80
(PUTPROP 'MELANIA 'ORAŞ 'BUCUREŞTI) → BUCUREŞTI

Funcţiile cu care lucrăm pentru listele de proprietăţi sunt:

(GET sym prop)	- scoate proprietatea
(PUTPROP sym prop val)	- pune proprietatea
(REMPROP sym prop val)	- şterge toate proprietăţile
(SYMBOL-PLIST sym)	- lista proprietăţilor lui sym

Scoate proprietatea **(GET sym prop)**
Întoarce valoarea atașată atomului simbolic proprietății *prop*.
Valoarea este NIL dacă proprietatea nu a fost atașată.
(GET ' MELANIA 'ORAS) → BUCURESTI
(GET 'MELANIA 'MEDIE) → 9.80
(GET 'MELANIA 'SCOALA) → NIL
;lui MELANIA nu i s-a atașat nici o valoare la proprietatea SCOALA

PUTPROP **(PUTPROP sym prop val)**
Funcția PUTPROP este oferită de sistem sau poate fi construită de noi. Dacă nu există în sistem puneți-o în biblioteca dumneavoastră de funcții care se încarcă de cîte ori lucrați.
Exemplu. (PUTPROP 'MELANIA 'AMICI) → (ALI ROXI TEO SILVA)
(PUTPROP 'MELANIA 'TELEFON "2309845") → "2309845"
Soluție. Iată cum putem defini noi funcția PUTPROP.
(DEFUN PUTPROP (S P V) (SETF (GET S P) V))

Lista proprietăților unui atom simbolic **(SYMBOL-PLIST sym)**
SYMBOL-PLIST scoate lista tuturor proprietăților lui *sym* cu valorile atașate în forma (prop1 val1 prop2 val2 prop3 val3 ...). Avem:
(SYMBOL-PLIST 'MELANIA) → (ORAS BUCURESTI MEDIE 9.8
DATA-NASTERII (11 13 1989) CLASA 2)
Observăm că proprietățile sunt puse ca într-o stivă LIFO.

Șterg proprietatea **(REMPROP sym prop)**
Șterge proprietatea *prop* a atomului *sym* cu valoarea ei atașată. Avem:
:(REMPROP 'MELANIA 'DATA-NASTERII)
NIL
:(SYMBOL-PLIST 'MELANIA)
(ORAS BUCURESTI MEDIE 9.8 CLASA 2)

10.2. Intersecția prin liste de proprietăți

În cazul în care se știe că mulțimile sunt formate din atomi simbolici putem folosi tehnica marcării elementelor. Cîte comparații se fac în acest caz varianta clasică prezentată în carte?

```
(DEFUN INTERSECTION (L1 L2) ; L1 si L2 liste din atomi simbolici
; marchez pe rînd toți simbolii din L1
(MAPC '(LAMBDA(Z)(PUTPROP Z 'MARCAT T)) L1)
```

; întreb dacă pe rînd dacă simbolii din L2 sunt marcaţi
; cei marcaţi îi las pe loc, în locul celor nemarcaţi pun NIL
; MAPCARN scoate NIL-urile
; rămîn numai atomii simbolici comuni
(MAPCARN '(LAMBDA(Z)(IF (GET Z 'MARCAT) Z)) L2))

Acest truc poate avea loc numai datorită faptului că proprietăţile atomilor simbolici sunt văzute GLOBAL.

10.3. Bridge

Avem un program de jucat bridge. Prima operaţie pe care trebuie s-o facă un jucător este să numere punctele pe care le are în mînă. Cărţile care se socotesc sunt cele care nu au numere: *asul* (notăm cu A) =4; regele (notăm cu K) = 3; regina (notăm cu Q) = 2 şi valetul (notăm cu V) = 1. Acest lucru îl transmitem programului nostru prin:
(PUTPROP 'K 'PUNCTE 3) → 3 (PUTPROP 'A 'PUNCTE 4) → 4
(PUTPROP 'V 'PUNCTE 1) → 1 (PUTPROP 'Q 'PUNCTE 2) → 2
După care atomii simbolici K, A, Q, V vor avea proprietatea PUNCTE cu valorile respective în orice loc al programului nostru. Aceste valori sunt setate global. Fie L o dare de cărţi pentru un jucător, şi anume primeşte 52/4 = 13 cărţi.

```
 (SETF CARTE
'((A CUPA) (K TREFLA) (8 CUPA) (V TREFLA)(A TREFLA) (Q CARO)
(3 PICA)(3 CUPA)(9 CARO)(4 CARO) (7 CUPA)(6 PICA) (5 TREFLA))
)
```

Vrem să calculăm numărul de puncte pe care-l are în mînă jucătorul.

```
(DEFUN MINA (CARTI) ;carti = ((A cupa)(K trefla) ...(1 pica))
; numără cîte puncte sunt în total în CARTI
(APPLY   '+
(MAPCAR '(LAMBDA (Z)(IF (NUMBERP (CAR Z)) 0
                         (GET (CAR Z) 'PUNCTE)))  CARTI)
))
```

Apelăm: (MINA L) → 14
Funcţia MINA parcurge pe rînd toate cărţile şi scoate cu GET valoarea la proprietatea *puncte*.

Aceeaşi proprietate la toti **(PUNE-LA-FEL lista prop val)**
Avem o listă de atomi simbolici L să scriem o funcţie care să pună la toţi atomii aceeaşi valoare *val* la proprietatea *prop*.

Exemplu. (PUNE-LA-FEL '(ION DAN RADU) 'ORAS 'BRASOV) →
(DEFUN PUNE-LA-FEL (L P V) ;toti atomii din L le pun aceeasi V la P
(MAPC '(LAMBDA(Z)(PUTPROP Z P V)) L))

Ordinea atomilor unei liste **(MEMO-ORDINE lista)**

Să ţinem minte ordinea în care se află atomii unei liste.

```
(DEFUN MEMO-ORDINE (L);memorez ordinea initiala a atomilor din L
(LET ((V 0))
(MAPC '(LAMBDA(Z) (PUTPROP Z 'ORDINE (SETF V (1+ V)))) L)
))
```

Presupunem că am deranjat ordinea iniţială a atomilor din L şi vrem s-o reconstituim. Avem:

```
(SETF L '(A B C))
(MEMO-ORDINE L)
(SETF L '(B A C)) ; deranjez ordinea atomilor in L
(PUN-LA-LOC L) → (A B C)
(DEFUN PUN-LA-LOC (L &AUX R); pune la loc atomii din L
(LABELS ((ORDINEA (N L)
          (DOLIST (I L)
                  (IF (= N (GET I 'ORDINE)) (RETURN I))
          )))
(DO ((I (LENGTH L) (1- I))) ((ZEROP I) R)
      (SETF R (CONS (ORDINEA I L) R))
) ))
```

Exerciţii. 10. Liste de proprietăţi

1. Scrieţi intersecţia a două mulţimi cu DOLIST şi cu DO.
2. Concepeţi o funcţie care să scoată din lista L atomii care au aceeaşi valoare *val* pentru proprietatea *prop*.
3. Scrieţi un program MEMO a formulelor de trigonometrie din manualul clasei a 9-a, care să asiste elevul.
4. Scrieţi un program GEOMETRIE care să asiste elevul furnizînd formulele de ARIE şi VOLUM pentru toate formele geometrice din spaţiu predate în şcoală.
5. Scrieţi un program de urmărire a activităţii de laborator a tuturor studenţilor: prezenţă, predări proiecte, iniţiative, notări pe parcurs.
6. Mic program de contabilitate casnică.
7. Scrieţi un program AGENDA care prezintă numere de telefon, adrese şi zilele de naştere.

11. STRINGuri (sau şiruri de caractere)

Stringurile sunt bucăţi de text care apar între ghilimele, " .
Adoptăm în carte denumirea de **string** pentru *şirurile de caractere*. Acest tip de date apare aproape în toate limbajele şi este folosit cu precădere pentru prelucrarea textelor. Stringurile nu sunt memorate cu unicitate şi spre deosebire de *atomii simbolici* **nu** li se pot ataşa valori şi **nu** pot fi nume de funcţii. Operaţiile care se pot efectua pe stringuri sunt: concatenare, comparare, căutarea unui caracter (sau substring). Putem de asemenea să convertim un atom într-un string şi invers (în anumite condiţii). Iată cîteva stringuri:
Exemple."Vine Melania." "Este cald?" "x= 8*9 sau 10" "Hai.Romania! "

11.1. Stringul şi la alte tipuri de date

STRINGP **(STRINGP obiect)**
Predicat care verifică dacă *obiect* este sau **nu** de tip *string*.
Exemple. (STRINGP "1 si 2") → T (STRINGP "Melania!") →T
(STRINGP L) → NIL ;unde (SETF L '(A B C))
(STRINGP 24.9) → NIL (STRINGP 'ATOM) → NIL

STRING **(STRING obiect)**
Converteşte un atom simbolic sau numeric într-un *string*. Dacă *obiect* este chiar de tip string este chiar el întors ca valoare a funcţiei.

STRING la un argument întreg între 1 şi 255 întoarce caracterul corespunzător codului ASCII. Prezentăm în continuare funcţia TABEL-ASCII care întoarce toate perechile cod- caracter.

Exemple. (STRING 67) → "C" (STRING 99) → "c"
(STRING 'mama) → "MAMA" (STRING "ion maria") → "ion maria"

Soluţie.

```
(DEFUN TABEL-ASCII ( ) ;de la 1 la 255 toate codurile
(DOTIMES (I 255)
        (SETF I (1+ I))
        (PRINT `(,I ,(STRING I))) ; perechi cod-ascii caracter
))
```

Am folosit BACKQUOTE, `. Amintim că toate obiectele cu virgula în faţă dintr-o listă cu BACKQUOTE sunt evaluate.

CHAR **(CHAR string număr)**

Scoate din ***string*** al n-1-lea caracter, nu caracterul propriu-zis ci corespondentul său ASCII. Se începe de la 0.

Exemple. (CHAR "Melania" 0) → 77 ;codul lui M este 77
(CHAR "melania") → 101 ; codul lui m este 101

Propunere. Iată funcţia ASCII care scoate toate codurile ASCII ale alfabetului scris cu litere majuscule.

```
:(ASCII ALFABET)
(LITERA= A CODUL= 65)
...
```

Soluţie.

```
(SETF ALFABET "ABCDEFGHIJKLMNPOQRSTUVXYZW")
(DEFUN ASCII ( A) ; SCOATE COD
(DOTIMES (I (LENGTH ALF) )
(PRINC `(LITERA= ,(STRING (CHAR A I)) CODUL= ,(CHAR A I)))
))
```

CONCATENATE **(CONCATENATE :STRING s1 s2 ...)**

Este vorba evident de concatenarea unui număr oarecare de stringuri.

Rezulatul este un string. Apare cuvîntul cheie :STRING care este cerut de specificaţiile lui CLISP, deoarece s-a prevăzut o extindere pentru implementările viitoare a altor tipuri de date de concatenat.

Exemple. *(*CONCATENATE :STRING "a" "fost" "bine") → "afostbine"

Transformă listă în string **(LISTA-STRING lista)**

Să transformăm elementele lui *lista* într-un string. Exemple:

(LIST-STRING '(A B C)) → "ABC" (LIST-STRING '(X 1)) →"X1"

Soluţie.

```
(DEFUN LISTA-STRING (L) ; L=lista
(IF (NULL L) (CONCATENATE :STRING (STRING (CAR L))
        (LISTA- STRING (CDR L)) )
))
```

Litere mari/mici **STRING-DOWNCASE, STRING-UPCASE**

Deoarece în modul text (string) are importanţă dacă literele apar scrise cu majuscule sau litere mici, avem posibiltatea să convertim un şir de caractere scris cu majuscule în litere mici şi invers.

Exemplu. (STRING-DOWNCASE "A FOST O DATA") → "a fost o data"
(STRING-UPCASE "Melania este mare") → "MELANIA ESTE MARE"

EXPLODE, IMPLODE **(EXPLODE A)** **(IMPLODE lista)**

Pentru a transforma un atom simbolic în lista caracterelor care-l compun, unele dialecte LISP au implementată funcţia EXPLODE. Invers pentru a transforma o listă de atomi simbolici într-un atom simbolic există funcţia IMPLODE. Putem s-o construim şi noi.

Exemplu. (EXPLODE 'MELANIA) → (M E L A N I A)
(IMPLODE '(M E L A N I A)) → MELANIA

Soluţie.

```
(DEFUN  EXPLODE (A) ; A atom simbolic
(LET* ((S (STRING A)) (N (1- (LENGTH S))) R)
(DO ((I N (1-  I))) ((=< -1 I) R)
        (SETF R (CONS (INTERN (STRING (CHAR  S I))) R))
 )))
```

```
(DEFUN IMPLODE (L) ;L lista de atomi simbolici
(INTERN
(EVAL (APPEND (LIST 'CONCATENATE ':STRING)
              (MAPCAR '(LAMBDA (Z) (STRING Z))L)
))
))
```

Funcţia INTERN internează atomul simbolic tocmai generat în lista cu toţi atomii simbolici. Ei sunt trataţi ca fiind unici.

11.2. Comparări de stringuri

STRING-EQUAL **(STRING-NOT-EQUAL str1 str2)**

Compararea a două stringuri. Dacă sunt egale sau diferite. Este vorba de două predicate care întorc desigur T sau NIL. Nu se face distincţia între stringuri scrise cu litere mari sau mici.

Exemple. (STRING-EQUAL "X este VARIABILA" "x este variabila") → T

STRING-GREATERP **(STRING-NOT-GREATERP s1 s2)**

STRING-LESSP **(STRING-NOT-LESSP s1 s2)**

Compară dacă stringul ***str1*** este mai mare decît ***str2***, în sensul aşezării într-un dicţionar. Deci, dacă ***str1*** vine după ***str2***, atunci STRING- GREATERP va fi T. Pentru şiruri egale răspunsul este NIL. În cazul lui STRING-NOT-GREATERP cu excepţia faptului că în cazul în care ***str1*** este egal cu ***str2***, răspunsul va fi T. Toate situaţiile ignoră dacă şirurile sunt scrise cu litere mari sau mici.

Exemple. (STRING-LESSP "ab c" "am fost") → T

STRING= , STRING/= **(STRING= str1 str2)**

Testează egalitatea a două stringuri, dar ţin cont de faptul că sunt sau nu scrise cu litere mari sau mici. Se ia în considerare reprezentarea ASCII a caracterelor.

Exemple. (STRING= "hi" "Hi") → NIL (STRING/= "ai" "aI") →T

STRING<, STRING>,STRING>= **(STRING<= str1 str2)**

Toate aceste predicate compară *str1* şi *str2* din punct de vedere al ordinii în reprezentarea ASCII a caracterelor.

Exemple. (STRING> "abc" "Abc") → NIL (STRING<= "am" "aM") → T

11.3. Căutare în string

SEARCH **(SEARCH str1 str2)**

Caută în stringul *str2* dacă apare undeva *str1*. Intoarce numărul poziţiei din *str2* la care apare *str1*, în cazul afirmativ, altfel NIL. Caracterele sunt în reprezentarea ASCII, deci are importanţă dacă sunt scrise cu litere mici sau mari. Încercaţi să scrieţi o astfel de funcţie.

Exemple. (SEARCH "la" "Melania") → 2

(SEARCH "X" "Apare X in formula") → 6

SUBSEQ **(SUBSEQ string start [end])**

Extrage din *string* un alt string care începe de la poziţia *start* şi se termină la poziţia *end*. Dacă nu apare *end* este considerată ultima poziţie a şirului. String nu este modificat, ci doar copiat.

Exemple. (SUBSEQ "ROMANIA" 3 5) → "MANIA"

(SUBSEQ "ziua= 13 noiembrie" 5) → "13 noiembrie"

(SUBSEQ "melania" 2) → "elania"

Exerciţii. 11. Stringuri

1. Scrieţi o funcţie care transformă un număr într-o listă de numere.
   (N-LISTA 5612) → (5 6 1 2)
2. Scrieţi o funcţie care transformă o listă în şiruri de caractere.
   (LIST-TO-STRING '(A B C)) → "ABC"
3. Scoateţi lista literelor care apar într-un string.
   (LITERE "alfabetul este prea lung") → "alfbetusprng"
4. Generaţi aleator cuvinte de lungime K peste un ALFABET dat.
   (SETF K 3 ALFABET '(A B))
   (GENEREZ 3 ALFABET) → "AAB"
5. Generaţi N cuvinte aleatoare de lungime oarecare (mai mică decît K) peste un ALFABET dat. ***Exemplu.***
   (GENEREZ-CUVINTE 5 ALFABET 2) → ("AAB" "ABABB")

6. Scrieţi un tutorial în LISP care să prezinte funcţiile elementare.
7. Definiţi predicatele PREFIX? şi SUFIX? *Exemplu.*
   (PREFIX? "ANTE" "ANTERIOR") → T (SUFIX? "ul" "elevul") → T
8. Selectaţi dintr-o listă de cuvinte acele cuvinte care au un PREFIX sau SUFIX dat.
9. Frecvenţa apariţiei unei litere într-un şir de caractere (un text).
10. Care este cea mai frecventă literă în limba engleză?
11. Care este litera care apare cel mai frecvent în limba română? Cîte texte trebuie să analizaţi? Cum le alegeţi?
12. Frecvenţa vocalelor (A E I O U) într-un text indiferent dacă sunt scrise cu majuscule sau nu.
13. Comparaţi frecvenţa vocalelor raportată la numărul total de litere pe mai multe texte în limba română. Obţineţi aceleaşi rezultate?
14. Comparaţi frecvenţa vocalelor în limba română faţă de limba engleză. Poate fi considerată aceasta ca o amprentă a limbii?
15. Procentul apariţiei lui î faţă de celelalte litere în limba română. Evident studiaţi cazul pe scrierea veche. Apoi comparaţi procentul obţinut pe texte în scrierea nouă.
16. Scrieţi un program care transformă un text în limba română de la scrierea veche la scrierea nouă, actuală.
17. Cît la sută din cuvintele din limba română sunt afectate de noua scriere a limbii române?
18. Care sunt combinaţiile de litere care se repetă care pot apărea în cuvintele din limba română. Exemplu: "ii" (copii), "nn" (înnoptat)
19. Studiaţi combinaţiile de 2 litere care nu apar niciodată alături în limba română. De exemplu: "XT"
20. Care este frecvenţa literelor duble în limba română?
21. Care este frecvenţa literelor duble în limba engleză? Comparaţi cu rezulattal de la 18.
22. Care este cel mai scurt program care să detecteze dacă un text este în limba română sau în limba engleză? Presupunem că textul vine pe INTERNET şi este scris cu alfabetul limbii engleze.
23. Scoateţi dintr-un text dat dicţionarul cuvintelor care apar în text.
24. Scrieţi un program care se prelucrează texte şi adaugă dinamic la un dicţionar cuvintele noi care apar.

12. Mulţimi şi combinatorică elementară

12.1. Operaţii frecvente pe liste

Umple (FILL) o listă **(FILL lista nou loc)**

Avem o listă şi vrem să punem pe numărul reprezentat de *loc* un element *nou*. Observaţi această funcţie în sistemul cu care lucraţi şi vedeţi diferenţa. Am pornit de la numărarea elementelor listei de la 1.

Exemplu. (FILL '(A B C D E F) 'X 3) → (A B X D E F)

Soluţie.

```
(DEFUN FILL (L NOU LOC &AUX (K 0) ) ; K= contor de numarare
(MAPCAR '(LAMBDA (Z)
                (SETF K (1+ K)) (IF (= K LOC) NOU Z)) L) )
```

Scoate o secvenţă din listă **(SUBSEQ lista loc1 loc2)**

Scoate o bucată din *lista* de la *loc1* la *loc2*. Dacă scriem numai (SUBSEQ lista loc) va scoate sevcenţa din *lista* de la *loc* pînă la sfîrşitul listei. Pornim numărătoarea de la 0. Observăm că avem aceeaşi funcţie pentru stringuri. Uitaţi-vă în LISP-ul cu care lucraţi.

Exemplu. (SUBSEQ '(A B C D E F) 0 2) → (A B)

(SUBSEQ '(A B C D) 1) → (B C D)

Soluţie

```
(DEFUN SUBSEQ (LISTA LOC1 &OPTIONAL LOC2)
(LET (R) ; R= rezultat
(IF LOC2
   (DO ((I LISTA (CDR I)) (K 0 (1+ K)) )
```

```
            ( (OR (NULL I)(= K LOC2)) (REVERSE R))
        (IF (>= K LOC1) (SETF R (CONS (CAR I) R)) )
    )
; altfel daca nu exista LOC2  copiaza pina la sfirsit
   (DO ((I LISTA (CDR I)) (K 0 (1+ K)) ) ((NULL I) (REVERSE R))
            (IF (>= K LOC1) (SETF R (CONS (CAR I) R)) )
    )
) ; va intoarce pe (REVERSE R)
))
```

Numărul atomilor unei liste **(NR-ATOMI lista)**

Scrieţi o funcţie care să numere cîte apariţii de atomi sunt într-o listă, cu alte cuvinte să numerăm numărul obiectelor dintr- o listă ignorînd parantezele întîlnite sau dacă se repetă.

Exemplu. (NR-ATOMI '(A (B (C)) (A D))) → 5

```
(DEFUN NR-ATOMI (L)
(COND  ((NULL L) 0)
         ((ATOM L) 1)
          (T (+ (NR-ATOMI (CAR L)) (NR-ATOMI (CDR L))))
))
```

Lista atomilor unei liste **(LISTATOM lista)**

Să scriem o funcţie care desfiinţează toate parantezele interioare ale unei liste .

Exemplu. (LISTATOM '(A (B C)(D (A)))) → (A B C D A)

(LISTATOM '(A)) → (A)

Soluţie. Prezentăm două variante:

```
(DEFUN LISTATOM (L)
(COND  ((NULL L) NIL)
         ((ATOM L)(LIST L))
        (T (APPEND (LISTATOM (CAR L))(LISTATOM (CDR L))))
))

(DEFUN LISTATOM (L); scrisa cu MAPCAN
(COND  ((NULL L) NIL)
        ((ATOM L)(LIST L))
        (T (MAPCAN  #'LISTATOM  L))
))
```

Adîncimea unei liste **(ADINCIME lista)**

Concepeţi o funcţie care să întoarcă adîncimea unei liste, adică numărul maxim de paranteze imbricate. Văzută ca un arbore, ar corespunde cu lungimea ramurii celei mai lungi.

Exemplu. (ADINCIME '(A B)) → 1 (ADINCIME '(A ((B C) D) E)) → 3

Soluţie.

```
(DEFUN ADINCIME (L) ; L-lista
(IF (ATOM L) 0
        (MAX (1+ (ADINCIME (CAR L)) )(ADINCIME (CDR L)))
))
```

Apare la orice nivel **(APARE? atom lista)**

Verifică dacă *atom* apare în *lista,* indiferent de nivelul la care se află.

Exemplu. (APARE? 'A ' (B C(D (A)) H)) → T

Soluţie.

```
(DEFUN  APARE? (A L) ; A= atom, L=lista
(COND  ((ATOM L) NIL)
        ((MEMBER A L) T)
        (T  (OR (APARE? A (CAR L)) (APARE? A (CDR L)) ))
 ))
```

Şterge toate aparitiile **(STERGE atom lista)**

STERGE toate aparitiile lui *atom* din *lista* indiferent de locul unde se află.

Exemplu. (STERGE 'E '(B (E) C ((D E) E))) → (B () C ((D)))

Soluţie.

```
(DEFUN STERGE (A L) ; sterge pe A din lista L
(COND  ((ATOM L) L)
        ((EQUAL  A (CAR L)) (STERGE A (CDR L)) )
        (T (CONS (STERGE A (CAR L)) (STERGE A (CDR L)) ))
))
```

Număr de aparitii **(COUNT e lista)**

Să presupunem că avem o listă şi ne interesează numărul aparitiilor unui element dat în acea listă la nivel superficial. Propunem mai multe variante pentru a arăta felul în care se pot folosi facilităţile sistemului. Lăsăm pe cititor să aleagă varianta cea mai bună.

Exemplu. (COUNT 'A '(A B A C (A) L A)) → 3

```
(DEFUN COUNT (E L) ; varianta recursiva
(COND ((NULL L) 0)
        ((EQUAL E (CAR L)) (1+ (COUNT E (CDR L))))
        (T (COUNT E (CDR L)))
))
```

COUNT scris cu MAPCARN, care ca şi MAPCAR parcurge şi construieşte lista resultat, cu deosebirea că scoate NIL-urile.

```
(DEFUN COUNT (E L)
  (LENGTH (MAPCARN '(LAMBDA(Z) (EQUAL E Z)) L))
)
```

COUNT scris cu DOLIST:

```
(DEFUN COUNT (E L &AUX (REZ 0))
(DOLIST (I L REZ) ; REZ= Rezultat
        (IF (EQUAL I E)(SETF REZ (1+ REZ)))
))
```

COUNT scris cu iteraţie cu PROG şi etichetă:

```
(DEFUN COUNT (E L) ; Folosind structura de control cu PROG si eticheta
(PROG ((R 0))
ETICHETA
(COND ((NULL L)(RETURN R))
        ((EQUAL E (CAR L))(SETF R (1+ R)))
)
 (SETF L (CDR L)) (GO ETICHETA)
))
```

COUNT scris cu DO generalizat:

```
(DEFUN COUNT (A L &AUX (R 0)) ; R= rezultat
(DO ((I L (CDR I))) ((NULL I) R)
        (IF (EQUAL A (CAR I)) (SETF R (1+ R)) )
))
```

Poziţiile unui atom într-o listă **(POZITII K Lista)**

Iată o funcţie care întoarce lista cu poziţiile unde se află K în L.

Exemplu. (POZITII 'A '(C A B A A))→ (1 3 4)

Soluţie. Rescrieţi funcţia POZITII cu DOLIST.

```
(DEFUN POZITII (K L) ; K = atom, L=lista
(LABELS ((POS (K L R N) ; R= rezultat , N= contor, initial 0
(COND  ((NULL L) (REVERSE R))
        ((EQUAL K (CAR L)) (POS  K (CDR L) (CONS N R) (1+ N)) )
        (T  (POS K (CDR L) (1+ N)))
 )))
(POS K L NIL 0)
))
```

12.2. Mulţimile ca liste

După cum este lesne de imaginat o mulţime poate fi reprezentată ca o listă, înlocuind parantezele mulţimii cu cele ale listelor. De exemplu:
M = {a, b, c} se scrie ca (a b c), iar N = {{ a, b }, c} ca N = ((a b) c)
Mulţimea vidă $\varnothing$ poate să corespundă destul de bine în această reprezentare cu lista vidă, ().
Nu orice listă este însă mulţime, căci dacă elementele ei se repetă nu îndeplineşte condiţia de mulţime. Putem însă să scoatem dintr-o listă elementele care se repetă şi să formăm o mulţime.

Scoaterea repetiţiilor **(REMOVE-DUPLICATES lista)**
Scoateţi obiectele unei liste care se repetă la nivel superficial.
Exemplu.
(REMOVE-DUPLICATES '(A B (B) C A E B E E)) → (A B (B) C E)
Soluţie. Vom prezenta o soluţie recursivă.

```
(DEFUN REMOVE-DUPLICATES (L) ; L-lista
(COND  ((NULL L) NIL)
       ((MEMBER (CAR L)(CDR L))
                (REMOVE-DUPLICATES (CDR L)))
       (T (CONS (CAR L) (REMOVE-DUPLICATES (CDR L))))
))
```

Intersecţie de mulţimi **(INTERSECTION lista1 lista2)**
Iată o funcţie care întoarce intersecţia a două mulţimi.
Exemplu. (INTERSECTION '(A B C) '(B D C)) → (B C)
Soluţie. Iată o variantă recursivă. Rescrieţi-o cu recursie pe coadă.

```
(DEFUN INTERSECTION (X Y) ; X si Y multimi
(COND ((OR (NULL X)(NULL Y)) NIL)
        ((MEMBER (CAR X) Y)
                (CONS (CAR X) (INTERSECTION (CDR X) Y)))
        (T (INTERSECTION (CDR X) Y))
))
```

Scriem acum intersecţia unui număr variabil de mulţimi bazîndu-ne pe intersecţia a două mulţimi. Scrisă recursiv:

```
(DEFUN INTERSECTION* (&REST L)
(COND  ((NULL L) NIL)
        ((NULL (CDR L)) (CAR L))
        (T (INTERSECTION (CAR L)
                (APPLY 'INTERSECTION*  (CDR L))))
))
```

Scrisă iterativ. Folosind iteraţia cu DOLIST şi variabila de lucru REZ iniţializată cu prima mulţime din intersecţie.

```
(DEFUN INTERSECTION* (&REST L &AUX  (REZ (CAR L)) )
(DOLIST (I (CDR L) REZ)
  (UNLESS (SETF REZ (INTERSECTION I REZ)) (RETURN NIL))
))
```

Reuniune de mulţimi **(UNION lista1 lista2)**

Scrieţi o funcţie care să efectueze reuniunea (UNION în limba engleză), a două mulţimi.

Exemplu. (UNION '(A B C) '(D A B)) → (A B C D)
(UNION '(A B C D) NIL) → (A B C D)

```
(DEFUN UNION (X Y); X si Y multimi
(COND  ((NULL X)  Y)
        ((MEMBER (CAR X) Y) (UNION (CDR X)  Y))
        (T (CONS (CAR X) (UNION (CDR X) Y)
))
```

Diferenţa de mulţimi **(SET-DIFFERENCE lista1 lista2)**

Iată o funcţie care să efectuează diferenţa a două mulţimi.

Exemplu. (SET-DIFFERENCE '(A B C D) '(A C)) → (B D)

Soluţie.

```
(DEFUN SET-DIFFERENCE (X Y) ; X si Y multimi
(COND ((NULL X) NIL)
        ((MEMBER (CAR X)  Y) (SET-DIFFERENCE (CDR X) Y))
        (T (CONS (CAR X) (SET-DIFFERENCE (CDR X) Y)))
))
```

Produs cartezian de mulţimi **(PROD-CART lista1 lista2)**

Să scriem o funcţie care să efectueze produsul cartezian a două mulţimi.

Exemplu. (PRODUS-CART '(a b c) '(1 2))→((a 1)(b 1)(c 1)((a 2)(b 2)(c 2))

Soluţie. Prima variantă foloseşte funcţia DO spre deosebire de a doua care foloseşte DOLIST care este mai la îndemînă.

```
(DEFUN PRODUS-CART (X Y &AUX  REZ); cu DO generalizat
(DO ((I X (CDR I))) ((NULL I) REZ)
        (DO ((J Y (CDR J))) ((NULL J))
        (SETF REZ (CONS (LIST (CAR I) (CAR J)) REZ ))
)))
(DEFUN PRODUS-CART (X Y &AUX REZ); cu DOLIST
(DOLIST (I X REZ)
        (DOLIST (J Y )
        (SETF REZ (CONS (LIST I J) REZ))
 )))
```

O altă alternativă recursivă de a concepe această funcţie.

```
(DEFUN PRODUS-CART (X Y) ; X si Y multimi
(IF (NULL X) NIL
(APPEND (MAPCAR '(LAMBDA(Z)(LIST (CAR X) Z )) Y)
         (PRODUS-CART (CDR X) Y))
))
```

Varianta cea mai scurtă pare la prima vedere cea mai puţin transparentă.

```
(DEFUN PRODUS-CART (X Y) ; X si Y
(MAPCAN '(LAMBDA(Z1)
        (MAPCAR '(LAMBDA(Z2) (LIST Z1 Z2)) Y))  X) )
```

Incluziune de mulţimi **(SUBSETP lista1 lista2)**

Să scriem o funcţie care să testeze incluziunea a două mulţimi.

Soluţie. Varianta recursivă propusă mai jos are de fapt ultima ramură a COND-ului de prisos. De multe ori se scrie pentru claritate.

```
(DEFUN  SUBSETP (X Y) ; X si Y = doua multimi, liste
(COND  ((NULL X) T)
        ((MEMBER (CAR X) Y) (INCLUS (CDR X)  Y))
        (T NIL) ;ramura aceasta poate sa dispara
))
```

Altă strategie ar fi să formăm lista cu T sau NIL, fiind rezultatele apartenenței fiecărui element al listei X la lista Y și apoi să-i aplicăm funcția logică AND.
Reamintim că *macro-urile întîi expandează și pe urmă evaluează.*
Virgula din fața variabilei X și Y în combinație cu backquote ` au rolul de a inhiba evaluarea lui X și Y.

```
(DEFMACRO SUBSETP (X Y &AUX REZ); X si Y =multimi, liste
`(EVAL (CONS 'AND
        (MAPCAR '(LAMBDA (Z)(IF (MEMBER  Z ,Y) T)) ,X) )  ))
```

Egalitate de mulțimi **(EGAL-MULTIME lista1 lista2)**

Reamintim că egalitatea de mulțimi nu este echivalentă cu cea de liste. Astfel două mulțimi reprezentate ca liste sunt egale dacă listele sunt egale, dar pot fi egale și dacă elementele listelor sunt egale între ele. Iată o posibilitate de a testa dacă două liste sunt egale, pornind de la funcțiile scrise mai înainte.

```
(DEFUN EGAL-MULTIME1 (L1 L2)
        (AND (SUBSETP L1 L2) (SUBSETP L2 L1))  )
```

Încercați și o variantă de sine stătătoare fără a utiliza alte funcții.

Mulțimea părților unei mulțimi **(MULTIME-PARTI lista)**

Să scriem o funcție care întoarce mulțimea părților unei mulțimi.
Amintim că dacă mulțimea are **n** elemente, mulțimea părților are 2^n elemente.

Exemplu.

(MULTIME-PARTI '(A B C)) → (()(A B C)(A B)(B C)(A)(A)(B)(C))

Soluție.

```
(DEFUN MULTIME-PARTI (M) ; M=multime
( IF (NULL M) '(NIL)
(MAPCAN '(LAMBDA(Z)(LIST (CONS (CAR  M) Z) Z))
                        (MULTIME-PARTI (CDR M)) ))
))
```

Toate funcţiile definite pe A cu valori în B (ALLF A B)

Reamintim că numărul funcţiilor definite pe A cu valori în B este **BA** (A şi B mulţimi finite). Avem:

(ALLF '(A B) '(0 1)) ->(((A 0)(B 0)) ((A 0)(B 1)) ((A 1) (B 0)) ((A 1)(B 1)))

Soluţie.

```
(DEFUN ALLF  (A B) ; ALLF (A , B) = { f | f: A → B }
(IF (NULL (CDR A))
   (MAPCAR  '(LAMBDA(Z) (LIST (LIST (CAR A) Z))) B)
   (MAPCAN  '(LAMBDA(F)
              (MAPCAR  '(LAMBDA(J) (CONS (LIST (CAR A) J) F)) B))
               (ALLF (CDR A) B))
))
```

Combinările unei mulţimi (COMBINARI lista n)

Iată o funcţie recursivă care întoarce toate combinările unei mulţimi date (ca o listă) de cîte N. Avem:

(COMBINARI '(A B C D) 2) → ((C D)(B C)(B D)(A B)(A C)(A D))

Soluţie. Folosim observaţia că putem calcula combinările unei mulţimi de ***n*** elemente a cîte ***k***, folosind calculul deja efectuat pentru combinările unei mulţimi de ***n-1*** elemente cîte ***k-1***. Formaţi drept exemplu combinările mulţimii (A B C) cîte 2 şi încercaţi să deduceţi combinările mulţimii (A B C D) cîte 3.

```
(DEFUN COMBINARI (L N)
(COND ((NULL L) NIL)
        ((= N 1) (MAPCAR 'LIST L))
        (T (APPEND (COMBINARI (CDR L) N)
        (MAPCAR '(LAMBDA(Z) (CONS (CAR L) Z))
                        (COMBINARI (CDR L) (1- N)))))
))
```

Observaţi că programul de mai sus nu foloseşte nici o altă funcţie !
Dacă folosim numai numere o alternativă şi mai scurtă ar fi:

```
(DEFUN COMB (L K N); (COMB NIL 3 8)
(COND  ((= 0 K) (PRINT L))
        (T  (COMB (CONS N L)(1-  K)(1-  N ))
            (IF (< K N)(COMB L K (1-  N ))) )
 ))
```

Observaţi că în acest caz:(COMB NIL 2 3) → ((1 2)(2 3)(1 3)) şi mulţimea iniţială L se construieşte din numere.

Permutările unei mulţimi **(PERMUT lista)**

Să scriem o funcţie care efectuează permutările unei mulţimi date.

Exemplu. (permut '(a b c)) → ((c b a)(a c b)(c a b)(b c a) (a b c) (b a c))

Soluţie.Funcţa PERMUT foloseşte funcţia auxiliară INSERT.

```
(DEFUN PERMUT (L) ;L lista cu orice elemente
(COND  ((NULL L) NIL)
        ((NULL (CDR L)) (LIST L))
        (T (MAPCAN '(LAMBDA (X)(INSERT (CAR L) X))
                   (PERMUT (CDR L))))
 ))
(DEFUN INSERT (E L)
(CONS (APPEND L (LIST E))
(MAPCAR  '(LAMBDA (X) (APPEND (REVERSE (CDR
                                  (MEMBER X (REVERSE L))))
        (CONS E (MEMBER X L))))  L)) )
```

Aranjamentele unei mulţimi **(ARANJAMENTE lista n)**

Iată o funcţie care întoarce aranjamentele unei mulţimi, folosind permutările şi combinările. Avem:

(ARANJAMENTE '(A B C) 2) → ((C B)(B C)(B A)(A B) (C A)(A C))

Soluţie.

```
(DEFUN ARANJAMENTE(L N) ; aranjamente din lista L de cite N
(MAPCAN #'PERMUT (COMBINARI L N))  )
```

Combinaţiile unei mulţimi **(COMBINATII lista nr)**

Funcţia COMBINATII întoarce toate aşezările posibile ale elementelor din lista pe numărul ***nr*** de poziţii.

Exemple. (combinatii '(1 0) 2) → ((1 1)(1 0)(0 1)(0 0))

(combinatii '(a b c) 2) → ((c c)(c b)(c a)(b c)(b b)(b a)(a c)(a b)(a a))

(combinatii '(T NIL) 2) → ((T T) (T NIL) (NIL T)(NIL NIL))

Soluție.

```
(DEFUN COMBINATII (L N &AUX R) ;obiectele din L pe N pozitii
(COND (( N 1 ) NIL)
        ((= N 1) (MAPCAR 'LIST L))
        (T (DOLIST (I L R) (DOLIST (J (COMBINATII L (1- N)))
                                   (SETF R (CONS (CONS I J) R))  )))
))
```

12.4. Şabloane pe listă

MATCH **(MATCH pattern lista)**

Avem un ***şablon*** (pattern) pe care îl comparăm cu o listă dată. Un fel de găsire de corespondenţă. Căutăm să vedem dacă în listă există anumiţi atomi coresponzători şablonului. Dacă există îi captăm în variabile. Avem două tipuri de variabile:

>V care încep cu **>** care vor căuta să se identifice cu un singur atom

+V care încep cu **+** care se identifică cu oricîţi atomi (listă).

Exemple. (VAR1 '>V) → T (VAR* '+V) → T

(MATCH '(>X + >Y) '(56 + 23)) → T X→ 56 Y → 23

(MATCH '(>A ARE +B) '(MELANIA ARE 500 LEI)) → T

A→ MELANIA B → (500 LEI)

(MATCH '(>CINE DOARME +CUM) '(MELANIA DOARME)) → T

CINE → MELANIA CUM → NIL

Soluție.

```
(DEFUN VAR1 (X) ; X este variabila de tip >V
(LET ((E (EXPLODE X))) ; pt X egal cu >V, E devine (> V)
        (IF (EQ (CAR E) '>) (IMPLODE (CDR E)))
))
(DEFUN VAR* (X) ; X este variabila de tip +V
(LET ((E (EXPLODE X))) ; pt X egal cu +V, E devine (+ V)
        (IF (EQ (CAR E) '+) (IMPLODE (CDR E)))      ))
```

```
(DEFUN MATCH (P L &AUX V); P=pattern , L= lista
(COND((NULL P) (NULL L)) ;da=T, am ajuns la capatul lui P si L deodata
        ((EQUAL (CAR P)(CAR L))(MATCH (CDR P) (CDR L)))
        ((SETF V (VAR1 (CAR P))) ;V este >X, continui, pun X=(CAR L)
                (MATCH (CDR P)(CDR L)) (SET V (CAR L)) T)
        ((SETF V (VAR* (CAR P))); V este *X,  X devine (list(car L))
          (COND((MATCH (CDR P)(CDR L))(SET V (LIST (CAR L))) T)
                ((MATCH (CDR P) L) (SET V NIL) T); X poate fi NIL
                ((MATCH P (CDR L)) ;adaug la lista X (car L)
                        (SET V (CONS (CAR L) (EVAL V))) T)))
))
```

Selectare fără repetiţie dintr-o listă **(SELECT lista prop)**

Selectaţi dintr-o listă, indiferent de nivel, atomii care au o anumită propietate fără repetiţie. Observaţi recursivitatea soluţiei noastre.

Exemplu. (SELECT '((A 3)(B 3)(A 2)) 'ATOM) → (2 B 3 A)

(SELECT '((A 3)(B 3)(A 2)) 'INTERGERP) → (2 3)

Soluţie.

```
(DEFUN SELECT ( L PROP) ; L=lista , PROP= predicat
(LABELS ((SEL (L R) ; R=rezultat
(LET ((PRIM (CAR L))(REST (CDR L)))
(COND ((NULL L) R)
        ((ATOM PRIM)(IF (OR (NOT (FUNCALL PROP PRIM))
                            (MEMBER PRIM R))
                (SEL REST R)(SEL REST (CONS PRIM R))) )
        (T (SEL REST (SEL PRIM R)))
)) ))
(SEL L NIL)
))
```

Exerciţii. 12. Mulţimi şi combinatorică elementară

1. Fiind dată o relaţie binară ca o listă de asociere. Verificaţi pe rînd dacă este reflexivă, simetrică, tranzitivă.
2. O relaţie de toleranţă este o relaţie binară reflexivă şi simetrică şi **nu** este tranzitivă. Verificaţi dacă o relaţie dată este relaţie de toleranţă.
3. Concepeţi o reprezentare pentru matrici ca liste. Scrieţi în această reprezentare adunarea şi înmulţirea.
4. Calculaţi determinantul unei matrici văzută ca listă de liste.
5. Comentaţi cea mai bună funcţie pentru calculul produsului cartezian a 2 mulţimi.
6. Concepeţi funcţia SUBSETP scrisă cu ajutorul listelor de proprietăţi.
7. Scrieţi o reuniunea UNION a unui număr oarecare de mulţimi.
8. Scrieţi o funcţie (ADJOIN x lista) care-l adaugă pe *x* la *lista* numai dacă nu se află deja înăuntru. Întoarce *lista*.
9. Scrieţi mai eficient funcţia COMBINARI
10. Scrieţi mai eficient funcţia COMBINATII. Comentaţi diferenţele.
11. Scrieţi funcţia PUN-PARANTEZE cu DOLIST.
12. Scrieţi o funcţie (PUN-PARANTEZE N L) unde N este numărul de paranteze pe care vreau să-l pun pe fiecare obiect din lista L.
13. Scrieţi o funcţie care întoarce numerele de la 1 la N în ordine într-o listă.
14. Scrieţi funcţia (REMOVE-DUPLICATES lista) iterativ folosind DOLIST.
15. Concepeţi un alt MATCH care nu foloseşte EXPLODE sau IMPLODE (folosind altă convenţie de a identifica variabilele de tip atom sau listă)
16. Funcţia SET-EXCLUSIVE-OR este mulţimea acelor elemente care nu se află simultan în cele 2 mulţimi.

    (SET-EXCLUSIVE-OR '(A B C) '(C D)) → (A B D)

19. Fiind dată o listă L generaţi toate listele circulare pornind de la L. Cîte liste circulare există?

    Exemplu. (circular '(a b c)) → ((a b c) (b c a) (c a b)).

19. Scrieţi funcţia SELECT altfel cu ajutorul unor funcţii prezentate.
20. Program pentru ajutorul întocmirii ORAR-ului la cursurile opţionale.

Avem N studenţi, cu cîte K opţiuni din M cursuri propuse. Care cursuri se suprapun ? Caz real.

21. Program EXAMEN pe reţea. Profesorul aduce întrebările cu răspunsurile alternative. Se afişează punctajul şi timpul.

22. Se dă o relaţie binară. Care sunt proprietăţile ei ? Generaţi aleator relaţii şi faceţi o statistică. Proiect: ***Ordine în dezordine.***

23. Aveţi o lege internă cu operaţie binară. Care sunt proprietăţile ei. Generaţi aleator astfel de structuri algebrice. Faceţi o statistică a proprietăţilor lor. Proiect: ***Ordine în Dezordine.***

13. Intrări şi ieşiri

13.1. Tipuri de date

Găsirea tipului unui obiect **(TYPE-OF obiect)**

Putem testa tipul unui obiect cu (TYPE-OF obiect). Să ne reamintim că în LISP există obiecte de mai multe tipuri. Avem de exemplu:

:(TYPE-OF 'MELANIA)
:SYMBOL
:(TYPE-OF 1989)
:FIXNUM
:(TYPE-OF 3.14)
:FLOAT

Iată tipurile de date principale:

(TYPE-OF obiect)	obiect
:SYMBOL	atom simbolic
:CONS	listă
:FIXNUM	număr întreg
:FLOAT	număr real (cu virgulă)
:STRING	string (şir de caractere)
:FUNCTION	numele unei funcţii
:STREAM	un loc de input/ouput
:ARRAY	o matrice
:PACKAGEP	context

Unele tipuri de date le ştim, de altele ne vom ocupa în continuare.

Pentru a lucra cu fişiere trebuie să le declarăm ca şi obiecte de tip :STREAM un loc în care se poate scrie şi se poate citi. Ieşirile şi intrările implicite se fac la consolă.Alte variabile de tip :STREAM sunt:

STANDARD-OUTPUT *STANDARD-INPUT*
ERROR-OUTPUT *STANDARD-PRINT*

Avem

:(TYPE-OF *STANDARD-OUTPUT*)
:STREAM

Încarcă fişier (LOAD numefis [:verbose vflag][:print pflag])

Se încarcă în contextul curent fişierul ***numefis***. Cuvinte cheie :VERBOSE şi :PRINT. Steagurile *vflag* şi *pflag* pot fi T sau NIL. Pentru **:VERBOSE T** se scrie numele fişierului în timp ce este încărcat. Dacă **:PRINT T** înseamnă că în timp ce încarcă se scriu numele funcţiilor. Implicit avem VFLAG = T şi PFLAG = NIL.

Exemple. Avem fişierul *lucru.lsp* cu funcţiile DUBLU şi SALUT

:(LOAD "lucru.lsp") ;încarcă *lucru.lsp* din directorul curent.
T
:(LOAD 'LUCRU) ;putem scrie şi aşa dacă fişierul are extensia *.lsp*
T

Dacă dorim să afişăm funcţiile din fişier în timp ce se încarcă scriem:

:(LOAD 'LUCRU :PRINT T) ;scrie functiile in timp ce le incarca
DUBLU
SALUT
T

Deschide un fişier

(OPEN numefis :DIRECTION dir [:element-type :unsigned- byte])

Pregăteşte un fişier pentru scriere sau citire. Întoarce un :STREAM.

numefis ier - numele fişierului simbolic sau string
dir ecţie - *p*oate fi :OUTPUT sau :INPUT sau :IO

Deschizînd un fişier pentru scriere înseamnă ştergerea celui precedent în cazul în care există. Dacă sunt folosite opţiunile :ELEMENT-TYPE :UNSIGNED-BYTE vor deschide fişierul în mod binar .

Exemple.

:(SETF OUT (OPEN "LUCRU.LSP" :DIRECTION :OUTPUT))

#<FILE-#50165568>

:(TYPE-OF OUT)

:STREAM

:(PRINT 'MELANIA OUT) ; va scrie in *lucru.lsp* atomul *melania*

Să ne construim nişte funcţii mai simple pe care să le introducem în fişierul INIT.LSP (care se încarcă automat cu lansarea interpretorului).

Soluţie.

(DEFUN OPENI (FISIER): deschide FISIER pentru citire
(OPEN FISIER :DIRECTION :INPUT))

(DEFUN OPENO (FISIER) ;deschide FISIER pentru scriere
(OPEN FISIER :DIRECTION :OUTPUT))

Acum scriem mai scurt:

:(SETF FIS (OPENO "EX.LSP")) ; pregateste EX.LSP pt. scriere

STREAMP **(STREAMP obiect)**

STREAMP este un predicat, dacă *obiect* reprezintă sau nu un *stream*.

Exemple. (SETF LOC (OPEN "LUCRU.LSP" :DIRECTION :IO))

(STREAMP LOC) → T

PACKAGEP **(PACKAGEP obiect)**

Verifică dacă *obiect* este sau nu de tip PACKAGEP. Vom traduce *package* prin *context*. Înseamnă de fapt contextele în care lucrăm. Întodeauna cînd intrăm în LISP interpretorul ne oferă drept context implicit package-ul numit *SYSTEM*. Numele package-urilor este între *.

Avem: (PACKAGEP *SYSTEM*) → T

Închide STREAM **(CLOSE stream)**

Închide un STREAM care a fost deschis cu OPEN. În acest moment pe acest stream nu se mai poate efectua nici o operaţie de citire sau scriere. Întoarce NIL. Deschiderea unui alt stream nu înseamnă că acesta este închis automat. Pot fi deschise mai multe streamuri de citire şi scriere deodată.

Exemplu. Avem de copiat un fiser în alt fişier linie cu linie. Deci în TEST.LSP se citeşte şi în POEZIE se scrie.

:(COPIAZA "TEXT.LSP" "POEZIE") ;copiaza TEXT.LSP in POEZIE

T

Soluţie.

```
(DEFUN COPIAZA (FIS1 FIS2); copiaza linie cu linie un fisier
(LET ((IN (OPENI FIS1 )) (OUT (OPENO FIS2 ))); deschid fisiere
(DO () ((NULL (PRINT (READ-LINE IN) OUT)))
) ; un DO fara corp
(CLOSE IN )(CLOSE OUT) ; trebuie inchise fisierele care s-au deschis
))
```

13.3. Scriere

PRINC **(PRINC obiect [stream])**

Scrie *obiect* fără să treacă pe linie nouă. Nu ia în considerare ghilimelele sau caracterele <ESCAPE>.

stream - numele stream-ului în care scriem; în lipsă, se scrie la consolă (implicit avem *STANDARD-OUTPUT*).

Exemple. (PRINC "mama a fost la piata") → mama a fost la piata

(PROG() (PRINC 'A)(PRINC 'B)(PRINC 'C)) → ABC

(PROG() (PRINC 'A) (PRINC " b")) → A b

```
:(SETF OUT (OPEN "EX.LSP" :DIRECTION :OUTPUT))
; pregateste EX.LSP pentru scriere
:(PRINC 5)  ; scrie pe ecran
5
:(PRINC 5 OUT) ; va scrie 5 in EX.LSP
```

În interiorul stringurilor se poate pune caracterul \ (backslash) urmat de următoarele convenţii. Scrierea lor va avea efecte speciale:

- **** caracterul backslash însuşi
- **\N** se trece la rînd nou
- **\T** <TAB> (Tab = lasă spaţii goale)
- **\R** <Carriage return>
- **\E** <ESCAPE>
- **\NNN** caracterul al cărui cod octal este NNN

Exemple.

```
:(PRINC "\TA\TB")     ; pune TAB in fata lui A si B
        A    B
```

```
:(PRINC  "\T\Ta\Nb")  ; pune 2 TAB-uri in fata lui A si sare la rind nou
                a
b
:(PRINC  "\123") S      ; Caracterul cu codul 123 este S
S
:(PROG() (PRINC "X\T=\T")( PRINC X)( PRINC  "\Ngata"))
X       =       23
GATA
```

PRINT **(PRINT obiect [stream])**

La fel ca şi PRINC numai că scrie pe ***obiect*** pe un rînd nou în ***stream*** (declarat cu :OUTPUT). Implicit este terminalul.

PRIN1 **(PRIN1 obiect [stream])**

Implicit *stream* este *STANDARD-OUTPUT*. Scrie pe ***obiect*** în *stream*. Este funcţia de bază de tipărire. Întoarce obiectul evaluat. PRINT este PRIN1 în care s-a trecut la linie nouă. PRINC este PRIN1 fără caractere ESCAPE sau QUOTE.

Exemple.

```
:(LET()(PRINT "a")(PRINC "b")(PRINT 'a)(PRINC 'b))
"a"
b
A
BB ; LET întoarce ultimul calcul
```

Sare la linie nouă **(TERPRI [stream])**

TERPRI înseamnă TERmină PRIntare pe linia curentă. Cu alte cuvinte va sări la rînd nou. Întoarce întotdeauna NIL. Este echivalent cu **NEWLINE** sau **NL**, din alte dialecte şi limbaje.

Exemplu.

```
:(LET()(PRINC "a") (TERPRI) (PRINC "B") 'GATA)
a
BGATA
```

PPRINT **(PPRINT obiect [stream])**

Scrie o expresie frumos. Vine de la **P**retty **PRINT**ing. Mai precis aranjează scrierea într-o formă lizibilă uşor, stream este implicit *STANDARD-OUTPUT*. Verificaţi funcţia oferită de sistem.

Afişarea unui fişier **(SCRIE FIS &OPTIONAL OUT)**

Iată o funcţie care scrie un fişier dat în locul unde dorim noi.
In cazul apelului:

:(SCRIE "ex.lsp") ; va scrie fisierul EX.LSP la terminal

:(SCRIE "ex.lsp" 'LP) ; va scrie fisierul EX.LSP la imprimanta

Soluţie.

```
(DEFUN SCRIE (FIS &OPTIONAL (OUT *STANDARD-OUTPUT*))
;fis = nume fisier (STRING)
 (LET ((IN (OPENI FIS))) ;scrie un fisier la terminal sau imprimanta
 (DO () ((NULL (LET ((LINIE (READ-LINE IN )))
        (IF LINIE (PRINT LINIE OUT)) LINIE)))
 )
 (CLOSE IN) ; inchidem stream-ul IN deschis pt citire
))
```

FORMAT

Ştiu că de multe ori pînă acum aţi fi vrut ca printările să arate mai frumos şi vi s-a părut foarte greu să le aranjaţi numai cu PRINC şi PRINT şi TERPRI. Funcţia cu FORMAT puteţi să săriţi oricîte rînduri şi să vă situaţi exact unde doriţi pe pagină. Urmăriţi în documentaţţia sistemului dumneavoastră toţi parametri lui FORMAT. Noi aici vă amintim doar:

~& si **~%**	sare un rind nou, **~n&** sare **n** linii
~T	sare un TAB, **~nT** sare **n** TAB-uri

```
:(FORMAT T "MELANIA")
MELANIANIL
:(FORMAT T "~A" 'melania)
MELANIANIL
:(FORMAT T "~A si ~A" 'Melania 'ALINA)
MELANIA si ALINANIL
:(FORMAT T "~A~& ~T~A~&" 'meli "cinta")
MELI cinta
NIL
```

:(FORMAT T "~A si cu ~A fac ~A ~&" 5 8 (+ 5 8))
5 si cu 8 fac 13
NIL

13.4. Citire

READ **(READ [stream [eof]])**

Citeşte o expresie LISP dintr-un stream. Întoarce ca valoare expresia citită. Evident în lipsă *stream* este considerat *STANDARD-INPUT*, iar EOF drept NIL. EOF este valoarea, mesajul pe care dorim să-l întoarcem în caz de sfîrşit fişier. Simbolii citiţi sunt internaţi în contextul curent de lucru, respectiv **PACKAGE**. Situaţii posibile:

- dacă simbolul începe cu : va fi internat în contextul :KEYWORD
- dacă simbolul citit începe cu numele contextului urmat de : va fi internat în contextul numit.

Exemplu. *LUCRU*:MELANIA înseamnă că atomul simbolic MELANIA va fi luat din package-ul *LUCRU*. Deobicei cînd scriem

:(SYMBOL-PLIST 'MELANIA)

se afişează proprietăţile atomului simbolic MELANIA din contextul curent *SYSTEM*. Dacă simbolul începe cu #: va rămîne neinternat în package. Uitaţi-vă în specificaţiile LISP-ului vostru.

Exemple.

:(SETF I (READ)) ; citeste pe I de la consola
(a b c) ; tastam noi!
:I
(A B C)
:(READ (SETF IN (OPEN "EX.LSP" :DIRECTION :INPUT)))
;ex.lsp contine prima strofa din "Luceafarul" de M. Eminescu
A
:(READ IN)
FOST
:(READ IN)
O

Caut cuvinte în texte **(NR-CUVINT cuvint text)**

Scrieţi o funcţie care să numere de cîte ori apare *cuvînt* în *text*.

Exemplu. (NR-CUVINT "fata" "LUCEAFAR.LSP") → 1

In LUCEAFAR.LSP se află doar prima strofă din Luceafărul. Cuvintele se caută indiferent de scrierea lor cu majuscule sau nu.

```
(DEFUN NR-CUVINT(CUVINT TEXT); cuvint=string, text=fisier
(LET ( I  (IN  (OPENI "TEXT" )) (CONTOR 0))
(DO ()( (NULL (SETF I (READ IN))) CONTOR)
        (IF (STRING-EQUAL (STRING I) CUVINT)
                (SETF CONTOR (1+ CONTOR)))
    )
))
```

Citeşte caracter **(READ-CHAR [stream])**

Citeşte caracterul de la un *stream*. Întoarce valoarea ASCII a caracterului, deci un întreg între 0 şi 255. Implicit *stream* este *STANDARD-INPUT*.

Exemplu.

```
:(READ-CHAR)A
65
:(STRING (READ-CHAR))a
"a"
:(READ-CHAR)a
97
```

READ-LINE **(READ-LINE [stream])**

Citeşte o linie dintr-un *stream* într-un string. Întoarce stringul citit.

În EX.LSP este scrisă prima strofă din *Luceafărul* de M. Eminescu.

```
:(READ-LINE (SETF IN (OPENI "EX.LSP")))
A fost o data ca-n povesti
```

READ-CHAR-NO-HANG **(READ-CHAR-NO-HANG [stream [eof]])**

Citeşte un caracter dintr-un stream, dacă este vreunul disponibil, altfel se întoarce-imediat. **EOF** - valoarea întoarsă în caz de sfîrşit de fişier.

Citeşte fişier

Cum se citeşte şi se printează linie cu linie un fişier dat?

Înlocuind funcţia PRINT cu PRELUCREAZA putem obţine o funcţie care să citească şi să prelucreze linie cu linie un fişier dat.

Exemplu. (CITESTE-FISIER 'MELANIA)
Se presupune că fişierul MELANIA este MELANIA.LSP şi se află într-un director imediat accesibil.
Soluţie.

```
(DEFUN CITESTE-FISIER (FISIER) ; citeste si scrie fisier
(LET ( (IN (OPENI FISIER ) ) R)
(DO () ((NULL (SETF R (READ IN))))
        (PRINT R))
(CLOSE IN)
))
```

Să generalizăm: să citim un fişier şi să aplicăm funcţia FUN fiecărui obiect citit. Avem acelaşi efect ca înainte:

```
:(CITESTE "MELANIA.LSP" 'PRINT)
(DEFUN CITESTE (FISIER FUN)
(LET ((IN (OPENI FISIER)) R)
(DO () ((NULL (SETF R (READ IN))) )
        (FUNCALL  FUN  R) ; aplic functia FUN fiecarui obiect citit
)
(CLOSE IN)
 ))
```

În cazul în care dorim prelucrarea fişierului, caracter de caracter, putem folosi în loc de READ-LINE, funcţia READ sau READ-CHAR.

13.5. Compar fişiere

Fiind date două fişiere conţinînd şiruri de caractere F1.TXT şi F2.TXT. Să comparăm fiecare caracter din primul fişier cu fiecare caracter din celălalt fişier. Pentru acest caz particular să presupun că comparaţia constă în EGALITATE. Comparaţia se va face cu EQUAL (egalitatea cea mai generală, fără a testa tipul obiectelor). Avem:

```
(DEFUN COMPARA (X  Y);scrie numai caracterele egale
        (IF (EQUAL X Y)  (PRINT '(CARACTER= ,X)) )
)
```

Prezentăm în continuare trei alternative de rezolvare a problemei.
A. Iată vesiunea în care controlul se face în manieră recursivă.

```
(DEFUN COMPAR-FISIERE()
```

```
(SETF F1 (OPENI "F1.TXT"))
(CAUTA1) (CLOSE F1)
 )
(DEFUN CAUTA1 ()
(COND ((SETF S1 (READ-CHAR F1)) (SETF F2 (OPENI "F2.TXT"))
                    (CAUTA2  S1))  )
))
(DEFUN CAUTA2 (S1)
(COND ((SETF S2 (READ-CHAR F2))
                    (COMPARA S1 S2)(CAUTA2  S1))
          (T (CLOSE F2) (CAUTA1))
))
```

B. Iată versiunea în care controlul se face prin iterație folosind etichete și GO.

```
(DEFUN COMPAR-FISIERE (FISIER1 FISIER2)
(PROG ((F1 (OPENI FISIER1) ) F2 SIR1 SIR2)
CAUTA1
(COND ((SETF SIR1 (READ-CHAR S1)) (SETF F2 (OPENI FISIER2 ))
          (PROG ()
          CAUTA2
          (COND  ((SETF SIR2 (READ-CHAR F2))
                              (COMPARA SIR1 SIR2) (GO CAUTA2))
                    (T (CLOSE F2)(GO CAUTA1))
        )) )
          (T (CLOSE F1) ))
)))
```

C. Iată varianta în care folosim funcția DO.

```
(DEFUN COMPAR-FISIERE (FISIER1 FISIER2)
(LET ((F1 (OPENI FISIER1)) F2 SIR1 SIR2)
(DO () ((NULL (SETF SIR1 (READ-CHAR S1))) (PRINT 'GATA))
(SETF F2 (OPENI  FISIER2))
          (DO () ((NULL (SETF SIR2 (READ-CHAR F2))) )
                    (COMPARA SIR1 SIR2)
          )
```

```
(CLOSE F2)
)
(CLOSE F1)
))
```

PAUZA

Generalizaţi funcţia pentru un timp dat sau oră dată.

```
(DEFUN PAUZA() (PRINC "Tastati orice tasta pentru continuare")
        (DO () ( (LISTEN)) ) )
```

Poziţia unui cuvînt într-un text

Să concepem o funcţie care caută un cuvînt într-un text. Va întoarce linia şi poziţia de la care începe cuvintul sau NIL în caz de nereuşită.

```
:(CAUTA-CUVINT 'FATA 'TEXT)
O preafrumoasa fata
(LINIE= 4 POZ= 15)
```

Soluţie.

```
(DEFUN CAUTA-CUVINT (CUVINT TEXT) ;cuvint =string,text = fisier
(LET ((I)(LINIE 0)(POZ)( IN (OPENI  TEXT ))) ; in este stream de intrare
(SETF LINIE (1+ LINIE))
(DO ()((NULL (SETF I (READ-LINE IN))) ) ; citeste pina la sfirsit
   (IF (SETF POZ (SEARCH CUVINT I)) (AND (PRINT I)(RETURN)))
)
(PRINT '(LINIE= ,LINIE POZ= ,POZ))
))
```

13.6. Introducerea datelor

(INTRODU data pred &optional text)

Cînd introducem datele trebuie să ne asigurăm că ele sunt corecte. Deci ciclul introducerii datelor, indiferent de ce introducem, este cam acelaşi:

1. tipărim un text în care explicăm ce să se introducă
2. citirea a ceea ce s-a introdus
3. verificarea datelor introduse (dacă datele sunt corecte se termină) (dacă datele sunt greşite se reia de la 1).

Să concepem o funcţie prin care introducem date şi testăm corectitudinea lor. Funcţia afişează un text, verifică dacă ceea ce s-a tipărit are predicatul anunţat şi apoi întoarce data.

Deci iată cum se comportă soluţia propusă de noi, funcţia INTRODU:

```
:(INTRODU "N=" 'NUMBERP)
N=>5
5
:(INTRODU "N=" '(LAMBDA(Z) (> Z 10)) )
N=>5
GRESIT! Se asteapta proprietatea=(LAMBDA (Z) (> Z 10))
N=>22
22
:(INTRODU "Nume (string)=" '(LAMBDA(Z)(STRINGP Z)) )
Nume (string)= "Melania"
"Melania"
:(INTRODU "N=" ) ; OBS: putem sa nu punem nici o conditie !
N=>ORICE
ORICE
```

Soluţie.

```
(DEFUN INTRODU (TEXT &OPTIONAL P) ; P =predicat
(PROG (X)
      REPETA ; ETICHETA
(FORMAT T TEXT)
(SETF X (READ))
(WHEN P
(COND ((FUNCALL  P X) (RETURN X))
        (T (FORMAT T "GRESIT! Se asteapta proprietatea=~A !~%" P)
          (GO REPETA)) ; X nu are proprietatea, mai incearca
  )
 )
(RETURN X) ; intoarce valoarea introdusa pt ca e corecta !
))
```

13.7. Cum organizăm un MENU

Să organizăm repede un MENU pentru orice program. Afişează nişte opţiuni, utilizatorul alege un număr care reprezintă opţiunea, apoi proogramul lansează opţiunea aleasă. Să ne imaginăm că în variabila globală MENU scriem toate opţiunile programului:

```
(SETF MENU '(AUTORI  HELP  INTRODUCERE  ANALIZA  EXIT))
```

- Lansăm programul cu (START). E mai comod să denumim întotdeauna functia principala cu START !

:(START)

1 AUTORI
2 HELP
3 INTRODUCERE
4 ANALIZA
5 EXIT

Alegeti => **1**

Mihaela Malita

1 AUTORI
2 HELP

...

Iată pe scurt:

```
(DEFUN AUTORI () (Princ "Mihaela Malita"))
(DEFUN HELP()
(SCRIE-FISIER "HELP.DAT"))
(DEFUN INTRODUCERE ()
 ...MODULUL de introducere a datelor))
(DEFUN ANALIZA() ... MODULUL de analiza  ))
;Dupa ce facem selectia, se lanseaza functia aleasa si se reia din nou START
(DEFUN START()
(LISTA-OPTIUNI MENU)
(START))
; folosim functia INTRODU definita inainte
(DEFUN LISTA-OPTIUNI (L) ; L=MENU
(LET (I (LUNG (LENGTH L)) (K 0) )
(MAPC '(LAMBDA(Z)(SETF K (1+ K))
        (FORMAT T "~%~A ~T~A"  K Z)) L)
(SETF I (INTRODU "~% ALEGETI="
      '(LAMBDA (Z)(AND (NUMBERP Z) (PLUSP Z)(<= Z ,LUNG))) )
)
(FUNCALL (NTH (1- I) L));se apeleaza functia aleasa din lista L
))
```

Exerciţii. 13. Intrări şi ieşiri

1. Scrieţi recursiv şi iterativ o funcţie care să sară N rînduri.
2. Scrieţi o funcţie care să lase N spaţii goale.
3. Studiaţi funcţia PPRINT oferită de sistem.De gîndit cum s-ar putea scrie o funcţie PPRINT. Program.
4. Scrieţi o funcţie WRITE de oricîte argumente cu oricîte argumente.
5. Studiaţi funcţia FORMAT oferită de sistemul dumneavoastră.
6. Număraţi de cîte ori apare o literă în mai multe texte.
7. Număraţi de cîte ori apare un cuvînt în mai multe texte.
8. Scoateţi lista cuvintelor care apar într-un text în ordine alfabetică.
9. Cum puteţi să ieşiţi din LISP şi să intraţi înapoi în sistemul de operare în care lucraţi?
10. Cum puteţi să inseraţi în programul dvs. în LISP executabile scrise în alte limbaje de programare?
11. Scrieţi o funcţie care şterge un fişier. ***Exemplu.***

```
(DEFUN DELETE-FILE (FIS)
    (DOS (CONCATENATE :STRING DEL (STRING FIS))) )
```

12. Scrieţi o funcţie care adaugă un fişier la un fişier dat.
13. Implementaţi în LISP toate funcţiile cu fişiere (COPY, DELETE-FILE, DIR,TYPE) ale sistemului de operare în care lucraţi.
14. Scrieţi în limba română un HELP pentru LISP.

Partea a II-a

Antrenamente în LISP

14. Melania învaţă cu calculatorul

14.1. Melania învaţă *TABLA INMULŢIRII*

Melania este în clasa a 2-a. Trebuie să repete toată tabla înmulţirii pentru a doua zi la şcoală. Calculatorul o întreabă pe rînd, iar ea trebuie să răspundă corect, dacă nu, calculatorul îi spune rezultatul şi merge mai departe. Iată o sesiune de lucru:

Am lansat LISP-ul şi am încărcat fisierul MELANIA.LSP care va lansa funcţia principală TABLA

>lisp melania

HELLO! MELANIA
VREI sa inveti TABLA INMULTIRII?
Completeaza te rog formulele:
1 x 1 => **3**
Nu e bine, MELANIA! Asta o stiai! 1 x 1 = 1
1 x 2 => **2**
OK, MELI
1 x 3 => **GATA**
IESE din LISP

Soluţie. Fişerul MELANIA.LSP conţine următoarele:

```
(DEFUN TABLA ( ); Melania completeaza toata tabla inmultirii
(FORMAT T " \N \T HELLO! MELANIA \N \T VREI sa inveti
        TABLA INMULTIRII ? \N \T Completeaza te rog  formulele: \N "
)
(LET (Z R); Z = rezultatul inmultirii, R = ceea ce scrie Melania
```

```
(DOTIMES (X 10) ; toate numerele de la 1 la 10
  (DOTIMES (Y 10); cu toate numerele de la 1 la 10
   (FORMAT T "\N \N \T ~A x  ~A = " (1+ X) (1+ Y))
   (SETF Z (* (1+ X)(1+  Y))) ; rzulatul corect al inmultirii
   (COND ((EQUAL Z (SETF R (READ))) (FORMAT T "\T OK, MELI "))
         ((MEMBER R '(Q QUIT EXIT BYE GATA STOP)) (EXIT))
         (T   (FORMAT T " NU e bine, MELANIA! ~A \t ~A x ~A = ~A"
                (SELECT DOJENEALA) (1+ X) (1+ Y) Z) )
)))
(FORMAT T "\N \T \T GATA! BRAVO! Ai invatat
     TABLA INMULTIRII, Melania. \N");numai daca termina toata tabla
))
; simulare de dojeneli din care se alege una la intimplare
(SETF DOJENEALA  '(" Fii mai atenta!" " Asta o stiai!"
"Repeta in gand!" "CONCENTREAZA-TE" "Nu te repezi!" "Ia-o incet!"
"Ai rabdare!" "Esti putin obosita?" "Nu te enerva!" "Continua cu rabdare!"
))
; Selecteaza dintr-o lista  un element la intimplare
; (select '(a b c d e f)) -> c  -este  RANDOM ales -
(DEFUN SELECT (LISTA) ; aleg un element la intimplare din LISTA
       (NTH (RANDOM (LENGTH LISTA)) LISTA)
)
(TABLA)
```

14.2. Test *TABLA INMULŢIRII* pentru Melania

Acum Melania ştie tabla înmulţirii. Profesoara a doua zi va da un test întregii clase în care va pune 10 întrebări şi va pune nota corespunzător cu numărul de răspunsuri bune. Acasă ea vrea să simuleze cu calculatorul testul de la şcoală. Iată o sesiune de lucru:

În fişierul TEST.LSP avem funcţia TEST care se va lansa la încărcare (nu are nici o importanţă faptul că denumim la fel şi fişierul şi funcţia principală - în unele limbaje acest lucru nu se admite -).

```
>lisp test
        TEST DE TABLA Inmultirii
              BUNA MELANIA!
```

```
Te rog raspunde la intrebari:
        3 X  5 = > 15
BRAVO, Melania
        8 X  9 = > 68
Ai GRESIT, Melania! 8 X  9 = > 72
        7 X  5 = > 35
BRAVO, Melania
        1 X  1 = > 1
BRAVO, Melania
        4 X 5 = > 20
BRAVO, Melania
        9 X  9 = > 81
BRAVO, Melania
        7 X  5 = > 35
BRAVO, Melania
        5 X  3 = > 15
BRAVO, Melania
         1 X  3 = > 3
BRAVO, Melania
        3 X  2 = > 6
BRAVO, Melania

Melania, ai NOTA= 9

Mai vrei o data (Y/N) ? = > Nu
```

: *Ramine in Lisp*

Soluţie. Iată fişierul TEST.LSP:

```
(DEFUN TEST(); se pun 10 intrebari de tabla inmultirii, apoi NOTA
(LET ((K 0)) ; K contor = nr raspunsuri corecte, K=NOTA
(FORMAT T "\t TEST DE TABLA Inmultirii \n
        BUNA M E L A N I A ! \n
        Te rog raspunde la intrebari: \n")
(DOTIMES (I 10) ; pune 10 intrebari
(LET* ((X (RANDOM 11))(Y (RANDOM 11))
          (Z (* X Y)) R)
        (FORMAT T "\N \T ~A X ~A  = " X Y)
        (SETF R (READ)); R=raspuns Melania
```

```
(COND ((EQUAL Z R)(FORMAT T "  BRAVO, Melania")(INCF K))
      (T (FORMAT T "   Ai GRESIT, Melania! ")
         (FORMAT T "\T ~A X ~A = ~A \N" X Y Z))
)))
(FORMAT T "\N \T  Melania, ai NOTA= ~A \N" K)
(IF (YES-OR-NO-P "\N \N \T Mai vrei o data ")(START))
));sfirsit functie TEST
(TEST) ; lansez la incarcare fisier functia TEST
```

Exerciţii. 14. Melania învaţă cu calculatorul

1. Este foarte important ca ***programul să se adreseze copilului pe nume.*** Refaceţi tabla înmulţirii pentru orice copil. Intrebaţi la început numele copilului şi programul i se va adresa pe nume.
2. Aşa cum este conceput programul de TEST poate pune exact aceleaşi întrebări ceea ce doamna învăţătoare nu va face desigur. Modificaţi programul de TEST astfel încît să nu se pună aceleaşi întrebări într-un test dat.
3. Imaginaţi un program care repetă cu copilul tabla înmulţirii punînd întrebări aleator. Programul va ţine minte greşelile copilului şi i le va pune din nou mai tîrziu.
4. Programele trebuie gîndite separat pentru clasa 1, 2, 3 4. Pentru clasa 1: concepeţi un program care să-l ajute pe copil să facă adunări pînă în 20. De fiecare dată cînd răspunde corect programul să-i spună ceva frumos (sau să-i arate ceva, sau să-i cînte ceva frumos). Desigur nu de fiecare dată acelaşi lucru.
5. Concepeţi un program mai general care întreabă clasa în care se află copilul (1-4) şi în funcţie de răspuns îl întreabă potrivit vîrstei lui diferite socoteli (adunări, scăderi, înmulţiri, împărţiri).
6. Scrieţi un program de învăţare care afişează expresia de calculat precum şi răspunsuri posibile din care copilul să aleagă.
7. Scrieţi pentru copii un program care întreabă cunoştinţe despre cosmos (despre planete, despre soare).
8. Scrieţi un program care să dialogheze cu copilul. Să-i pună diferite întrebări legate de cum îl cheamă, şcoală, familie, adresă, cînd s-a născut. Să reţină ceea ce a spus copilul şi cînd vine a doua oară să compare răspunsurile date.
9. Un program de tipul ***cum se spune la ... în limba engleză*** care-l întreabă pe copil diferite cuvinte şi îl ajută să le memoreze.

15. Derivarea

Derivarea funcţiilor elementare (DERIV functie nume-variabila)
Să scriem o funcţie care ne derivează principalele funcţiile elementare (obţinute prin compuneri de +, - , * , / , EXP). Să ne reamintim cîteva din regulile de derivare a funcţiilor elementare:

c' = 0 (c=constanta) x' = 1 $(C \times F(x))' = C \times F(x)'$

$(F(x) + G(x))' = F(x) + G(x)'$

$(F(x) - G(x))' = F(x) - G(x))'$

$(F(x) \times G(x))' = F(x) \times G(x)' + F(x)' \times G(x)$

$(F(x) / G(x))' = (F(x) \times G(x)' - F(x)' \times G(x)) / G^2(x)$

$(F^N(x))' = N F^{N-1}(x) \times F(x)'$

Exemple. (DERIV '(* (+ X A) ((+ X X) 2))) →
(+ (* 1 ((+ X X) 2)) (* (+ X A) (* 2 (+ X X))))

(DERIV '(EXPT (+ A X) 2) 'X) → (* 2 (EXPT(+ A X) (1- 2)) (+ 0 1))

(DERIV '(- (EXPT X 2) 3) 'X) → (-(* 2 (EXPT X (1- 2)) 1) 0)

(DERIV '(/ 1 X) 'X) → (/ (- (* 0 X) (* 1 1))(EXPT X 2))

(DERIV '(/ (+ A X)(+ B X)) 'X) →
(/ (- (* (+ 0 1) (+ B X)) (* (+ A X) (+ 0 1))) (EXPT (+ B X) 2))

Soluţie.

```
(DEFUN DERIV (F X) ; F o functie scrisa in Prefixat, X variabila de
derivat
(COND  ((EQUAL F X) 1) ; daca F(x)=X ,derivarea =1
       ((ATOM F) 0) ; daca F este constanta
       ((MEMBER (CAR F) '(+ - ) ) (DERIV+-  (CAR F) F X))
       ((EQUAL (CAR F) '*) (DERIV* F X))
```

```
        ((EQUAL (CAR F) '/) (DERIV/ F X))
        ((EQUAL (CAR F) 'EXPT) (DERIVEXPT F X))
        (T 'ALTE-CAZURI)
))
(DEFUN DERIV+- (OP F X) ;  F= (+  F1 F2) sau F=(- F1 F2)
(LET ((F1  (CADR F)) (F2  (CADDR F)))
        `(,OP ,(DERIV F1 X) ,(DERIV F2 X))
))

(DEFUN DERIV* (F X) ; F= (* F1 F2)
(LET ((F1  (CADR F)) (F2  (CADDR  F)))
        `(+ (* ,(DERIV F1 X)  ,F2)  (* ,F1 ,(DERIV B F2)))
 ))
(DEFUN DERIV/ (F X) ; F = (/  F1  F2)
(LET ((F1 (CADR F))(F2 (CADDR F)))
        `(/ (- (* ,(DERIV F1 X) ,F2) (* ,F1 ,(DERIV F2 X))) (EXPT ,F2
2))
))
(DEFUN DERIVEXPT (F  X) ; F = (EXPT G N)
(LET ((G (CADR F)) (N  (CADDR F)))
        `(*  ,N  (EXPT ,G  ,(1- N ))  ,(DERIV G X))
))
```

Exerciţii. 15. Derivarea

1. Extindeţi funcţia DERIV astfel încît să recunoască şi să deriveze şi funcţiile trigonometrice.
2. Scrieţi un program care simplifică expresia întoarsă de DERIV.
3. Scrieţi un TUTORIAL penrtu explicarea derivării şi testarea cunoştinţelor dobîndite.
4. Scrieţi un program care să efectueze operaţii cu polinoame într-o variabilă.
5. Scrieţi un program care să efectueze toate operaţiile cunoscute cu funcţii booleene. Interfaţă prietenoasă.
6. Scrieţi un tutorial pentru explicarea funcţiilor booleene.

16. Mulţimi fuzzy

16.1. Noţiuni generale

Noţiunea de mulţime fuzzy a fost introdusă în matematică de L.A.Zadeh în anul 1965, /29/. Mulţimile fuzzy (şi conceptele fuzzy în general) au apărut din necesitatea de a descrie noţiunile vagi, imprecise (în traducere *fuzzy* înseamnă *vag, neclar, estompat, imprecis*). De exemplu, proprietăţi cum ar fi *înalt, scund, bogat, sărac, mic* sau *mare* sunt **imprecise, fuzzy.**

Este greu să vorbim riguros despre ***mulţimea oamenilor scunzi,*** deoarece noţiunea de *scund* face parte din clasa noţiunilor imprecise. Se observă însa că în funcţie de înălţime, putem asocia fiecarui om un grad de apartenenţă la această mulţime. Un adult de 1,20 m va fi considerat scund în mod sigur (deci gradul de apartenenţă la mulţimea oamenilor scunzi poate fi considerat 1), iar cineva de 2m nu este scund în mod cert, (deci în acest caz gradul de apartenenţă la mulţimea oamenilor scunzi va putea fi considerat 0). Celor cu înălţimi intermediare le putem asocia corespunzător grade de apartenenţă la această mulţime.

Există teorii matematice care se ocupă cu studiul fenomenelor aleatoare: teoria probabilităţilor, statistica matematică.Trebuie însă făcută distincţia între fenomenele aleatoare şi cele fuzzy. Fenomenul aleator rezultă, în principiu, din nesiguranţa în ceea ce priveşte apartenenţa unui obiect la o clasă, în timp ce, în cadrul unui fenomen de natură fuzzy există *grade de apartenenţă* intermediare între apartenenţa completă şi neapartenenţă .

Apare deci naturală definirii unei mulţimi fuzzy prin descrierea acestor *grade*

de apartenenţă. Vom alege drept mulţimea valorilor gradelor de apartenenţă intervalul [0,1]. Vom considera că:

- grad de apartenenţă 0 înseamnă ***neapartenenţa***
- gradul de apartenenţă egal cu 1 ***apartenenţa totală.***

Definiţie. Numim **mulţime fuzzy** în X , o aplicaţie F: $X \longrightarrow [0,1]$.

De cele mai multe ori vom considera o mulţime fuzzy drept o mulţime de perechi $F = \{ (x , f(x)) \mid x \in X \}$. F este o submulţime a produsului cartezian X x f(X)), iar $f : X \longrightarrow [0,1]$ se numeşte ***funcţia caracteristică a lui F.***

Observaţie. Noţiunea de mulţime fuzzy poate fi extinsă dacă admitem ca funcţia caracteristică să ia valori într-o latice sau într-o algebră Boole.

Operaţii cu mulţimi fuzzy

Operaţiile uzuale din teoria clasică a mulţimilor se extind la mulţimi fuzzy.

Mulţimea vidă, $\emptyset$. Este mulţimea cu funcţia caracteristică

$$f : X \longrightarrow [0,1],\ f(x)=0,\ \forall\ x \in X.$$

Mulţimea totală X. Este mulţimea cu funcţia caracteristică

$$f : X \longrightarrow [0,1],\ f(x)=1,\ \forall\ x \in X\ .$$

Egalitatea mulţimilor fuzzy

Două mulţimi fuzzy (M, f) şi (N, g) sunt egale dacă funcţiile lor caracteristice sunt egale: M = N şi f=g

Incluziunea a două mulţimi fuzzy

Spunem că (M, f) este inclus în (N, g) dacă $M \subset N$ şi $f \leq g$, unde $\leq$ este relaţia de ordine punctuală între funcţii: $\forall x \in X,\ f(x) \leq g(x)$.

Reuniunea a două mulţimi fuzzy

Reuniunea a două mulţimi fuzzy (M, f) şi (N, g) este $(M \cup N, h)$ mulţimea fuzzy cu funcţia caracteristică h: $X \longrightarrow [0,1]$,

$h(x) = \max(f(x), g(x)),\ \forall\ x \in X$.

Intersecţia a două mulţimi fuzzy

Inetrsecţia lui (M, f) şi (N, f) este ($M \cap N$, h) este mulţimea fuzzy cu funcţia caracteristică $h:X \rightarrow [0,1]$, $h(x) = \min(f(x), g(x))$, $\forall x \in X$.

Complementara unei mulţimi fuzzy (M,f)

este (C_M, h) unde C_M este complementara mulţimii M, iar h este funcţia caracteristică: $h(x)=1 - f(x)$, $\forall x \in X$.

Reprezentarea mulţimilor fuzzy în LISP

Dacă ţinem seama de definiţia mulţimilor fuzzy, ele se pot reprezenta în mod natural ca liste de asociere (A-liste), fiecare din perechile cu punct avînd prima componentă un element al lui X şi a doua componentă gradul de apartenenţă al elementului la mulţime.

Avem lista ((a . 0.3)(b . 0.4)(c . 0.5)) care reprezentă mulţimea fuzzy unde M = { a, b, c }, gradul de apartenenţă al lui *a* la mulţime fiind 0.3, al lui *b*, 0.4 şi al lui *c*, 0.5.

16.2. Operaţii cu mulţimi fuzy

Mulţime fuzzy? **(MULTIME-FUZZY? A-lista)**

Să verificăm dacă o listă de asociere reprezintă o mulţime fuzzy.

Lista nu reprezintă o mulţime fuzzy:

- dacă un element apare de mai multe ori (cu grade de apartenenţa diferite sau nu)
- dacă gradul asociat unui element nu este număr sau este supraunitar

Exemplu. (MULTIME-FUZZY? '((A . 0.1) (B . 0.4) (C . 0.5))) → T

(MULTIME-FUZZY? '((A . 0.2) (B . 1.2) (C . 0.6))) → NIL

; pentru că gradul de apartenenţa al lui B este supraunitar

(MULTIME-FUZZY? '((A . RAU) (B . 1.2) (C . 0.3))) → NIL

; pentru că gradul de apartenenţa al lui A nu este număr

Soluţie.

```
(DEFUN MULTIME-FUZZY? (L); L=A-lista
(COND  ((NULL L) T)
        ((MEMBER (CAAR L)(MAPCAR 'CAR (CDR L))) NIL)
        ((AND(NUMBERP (CDAR L))(<=(CDAR L) 1)(>= (CDAR L) 0)))
                (MULTIME-FUZZY? (CDR L)))
))
```

Fuzzy apartenenţa **(FUZZY-APARTINE elem m-fuzzy)**

Să verificăm dacă un element A aparţine unei mulţimi fuzzy X şi în caz afirmativ să obţinem gradul de apartenenţă al elementului la mulţime.

Exemplu. (FUZZY-APARTINE 'A '((B . 0.1)(A . 0.4)(C . 1.0))) → 0.4

Soluţie.

```
(DEFUN FUZZY-APARTINE (A X) ; X=A-lista
(LET ( (R (ASSOC A X) ) ); R= (element .grad) sau NIL
        (IF (NULL R) 0 (CDR R)) ; daca R este NIL intorc 0
))
```

Ştergerea elementelor cu grad zero **(FUZZY-0 A-lista)**

Să scoatem dintr-o mulţime fuzzy toate elementele cu gradul de apartenenţă zero, pentru că am văzut că sunt de prisos.

Exemplu. (FUZZY-0 '((A . 0.1)(B . 0.0)(C . 0.3))) → ((A . 0.1)(C . 0.3))

Soluţie.

```
(DEFUN FUZZY-0 (X); scoate elementele cu grad=0
(COND  ((NULL X) NIL)
        ((ZEROP (CDAR X)) (FUZZY-0 (CDR X)))
        (T (CONS (CAR X) (FUZZY-0 (CDR  Y))))  ))
```

Incluziune de mulţimi fuzzy **(FUZZY-INCLUS X Y)**

Scriem un predicat care spune dacă mulţimea fuzzy X este inclusă în mulţimea fuzzy Y.

Exemple.

(FUZZY-INCLUS '((A . 0.1)(B . 1.0)) '((A . 0.2)(B . 1.0)(C . 0.3))) → T
(FUZZY-INCLUS '((A . 0.1)(B . 1.0)) '((A . 0.3)(B . 0.9)(C . 1.0))) → NIL
;pentru că gradul de apartenenţă al lui B la a doua mulţime este mai mic
;decît gradul lui de apartenenţă la prima mulţime.

Soluţie.

```
(DEFUN FUZZY-INCLUS (X Y); X, Y = A-liste
(COND ((NULL X) T)
        ((NULL (ASSOC (CAAR X) Y)) NIL)
        ((<= (CDAR X)(ASSOC (CAAR X) Y))
                        (FUZZY-INCLUS (CDR X) Y))
))
```

Intersecţie de mulţimi fuzzy **(FUZZY-INTERSECTION X Y)**

Scrieţi o funcţie pentru intersecţia a două mulţimi fuzzy.

Exemplu. (FUZZY-INTERSECTION '((A . 0.1)(B . 0.4)(C . 1.0))
'((A . 0.5)(C . 0.5)(D . 0.9))) → ((A . 0.1)(C . 0.5))
(FUZZY-INTERSECTION '((A . 0.1)(B . 0.2)(C . 0.9))
'((X . 1.0)(Y . 0.2)(Z . 0.9))) → NIL

Soluţie.

```
(DEFUN FUZZY-INTERSECTION (X Y); X, Y=A-liste
(COND  ((NULL Y) NIL)
        ((ASSOC (CAAR Y) X)  (CONS (CONS (CAAR X)
                        (MIN (CDAR Y)(CDR  (ASSOC (CAAR Y) X)) ))
                                (FUZZY-INTERSECTION X (CDR Y))))
        (T (FUZZY-INTERSECTION X (CDR Y)))
 ))
```

Reuniune de mulţimi fuzzy **(FUZZY-UNION X Y)**

Să concepem o funcţie pentru reuniunea a două mulţimi fuzzy.

Exemplu. (fuzzy-UNION '((b . 0.3)(c . 1.0)) '((a . 0.4)(c . 0.5)(d . 0.9))) →
((a . 0.4)(b . 0.3)(c . 1.0)(d . 0.9))

Soluţie.

```
(DEFUN FUZZY-UNION (A B) ; A si B multimi fuzzy
(COND  ((NULL B) A)
       ((NULL A) B)
       ((ASSOC (CAAR B) A)
            (FUZZY-UNION  (INTRODUC (CONS (CAAR B)
                  (MAX (CDAR  B)(CDR(ASSOC (CAAR B) A)))) A)
                                (CDR B)))
       (T (CONS (CAR B) (FUZZY-UNION A (CDR B))))
))

(DEFUN INTRODUC (E  LISTA); E= (element . grad-apartenenta)
(COND ((NULL LISTA) NIL)
       ((EQUAL (CAR E) (CAAR LISTA)) (CONS E (CDR LISTA)) )
       (T (CONS (CAR LISTA) (INTRODUC  E (CDR LISTA))))
))
```

Complementara **(FUZZY-C X m-fuzzy univers)**

Să concepem o funcţie pentru calculul complementarei unei mulţimi fuzzy F : X -> [0,1].

Exemplu.

(FUZZY-C '((A . 1.0)(B . 0.3)(D . 0.9)) '(A B C D)) → ((B . 0.7)(C . 1.0)(D . 0.1))

Elementul C nu apare în mulţimea fuzzy dată, deci are gradul de apartenenţă 0. În complementară va avea gradul de apartenenţă 1.

(FUZZY-C '((A .1.0)(B 0.3)(D . 1.0)) '(A B C)) → *Eroare*

Rezultatul este ***eroare*** pentru că în mulţimea fuzzy dată apare un element care nu este în universul peste care a fost definită mulţimea fuzzy.

Soluţie.

```
(DEFUN FUZZY-C (X  C); complementara lui X fata de C
(COND  ((NULL X) (MAPCAR #'(LAMBDA(Z) (CONS Z 1.0)) C) )
       ((NULL C) 'EROARE)
```

```
(T (IF  (= (CDAR X) 1.0)
        (FUZZY-C (CDR X) (DELETE (CAAR X) C))
        (CONS (CONS (CAAR X)  (1-  (CDAR X)) )
              (FUZZY-C  (CDR X)  (DELETE (CAAR X) C)))))
))
```

Normalizarea unei mulţimi fuzzy **(NORMALIZARE m-fuzzy)**

O mulţime fuzzy F se numeşte ***normală*** dacă sup {f(x) ; x ∈ X } = 1
şi ***subnormală*** în caz contrar .
Orice mulţime fuzzy nevidă subnormală se poate normaliza, considerînd drept funcţie caracteristică f(x)/sup f(x), ∀ x ∈ X.
Normalizarea mulţimii fuzzy X se obţine împărţind gradele de apartenenţă ale tuturor elementelor lui X la M, unde M este cel mai mare grad existent în X.

Exemplu. (NORM '((A . 0.01) (B . 0.02) (C . 0.1) (D . 0.09))) →
((A . 0.1) (B . 0.2) (C . 1.0) (D . 0.9)) unde
M=max (0.01 00.2 0.1 0.09) = 0.1

Soluţie.

```
(DEFUN NORM (X); X =A-lista
(LET ((M  (APPLY 'MAX (MAPCAR 'CDR  X))) )
        (MAPCAR '(LAMBDA(Z)( CONS (CAR Z)( / (CDAR Z) M )) X)
))
```

Exerciţii. 16. Mulţimi fuzzy

1. Daţi 5 exemple de atribute care pot fi considerate fuzzy.
2. Selectaţi dintr-o mulţime fuzzy toate elementele de grad mai mare ca un GRAD dat.
3. Scrieţi într-o singură funcţie folosind iteraţia cu DO, funcţia ***normalizare***.
4. Concepeţi o reprezentare a mulţimilor fuzzy ca liste de proprietăţi. Refaceţi în această reprezentare: intersecţia, reuniunea, complementara.

5. Scrieţi un predicat care testează egalitatea a două mulţimi fuzzy, reprezentate ca liste de proprietăţi
6. Scrieţi un program TUTORIAL care să explice neştiutorilor conceptul de mulţime fuzzy.
7. Scrieţi un program care să lucreze cu mulţimi fuzzy, implementînd diferitele definiţii existente pentru intersecţie, reuniune şi complementară existente în literatura de specialitate.

17. Grafuri

17.1. Grafuri. Noţiuni generale

Numim **graf neorientat** o pereche ordonată G = (V, E), unde V este o mulţime finită şi nevidă, iar E este o mulţime de perechi neordonate de elemente din V, (vezi /36/).
Elementele lui V se numesc ***noduri*** sau ***vîrfuri***, iar elementele lui E se numesc ***muchii***.

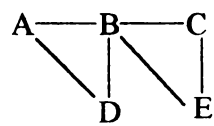


Fig.7.1. Graf neorientat

Numim **graf orientat** sau **digraf** o pereche ordonată G= (V , E) cu V o mulţime finită şi nevidă de ***noduri*** şi E mulţime de perechi ordonate de elemente din V (E este o submulţime a produsului cartezian V × V). Elementele mulţimii E se numesc ***arce***.

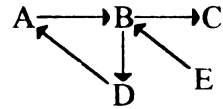


Fig.17.2. Un graf orientat=digraf

Reprezentarea grafurilor orientate
Grafurile pot fi reprezentate în mai multe moduri, sugerăm cîteva:
A. ca liste de asociere
B. prin liste de proprietăţi
C. prin valoarea nodurilor

A. Reprezentarea grafurilor ca liste de asociere

Este cel mai natural mod de a reprezenta grafurile: listele de asociere sunt liste de perechi cu punct, fiecare pereche cu punct (I . J) reprezentînd arcul de la nodul I la nodul J.

Digraful din figura 17.2 reprezintă lista de asociere:

((A . B) (B . C) (D . A) (B . D) (E . B))

B. Reprezentarea grafurilor prin liste de proprietăţi

Fiecărui nod I se asociază o proprietate numită IN, a cărei valoare este lista tuturor nodurilor J pentru care există arc de la J la I, şi o proprietate numită OUT, a cărei valoare este lista nodurilor J pentru care există arc de la I la J. Observaţi că, de fapt, nu este nevoie de ambele proprietăţi (un graf poate fi complet descris şi numai de una din acestea). Pentru digraful din Fig.17.2 avem:

NOD	IN	OUT
A	(D)	(B)
B	(A E)	(D C)
C	(B)	NIL
D	(B)	(A)
E	NIL	(B)

Avem un graful G, dat sub formă de listă de asociere. Să-l reprezentăm prin proprietăţile IN şi OUT ataşate fiecărui nod.

Exemplu. (PUNE-PROP '((A . B) (B . C) (D . A) (B . D) (E . B)))

;are ca rezultat definirea proprietăţilor IN, respectiv OUT pentru nodurile A,B,C,D din Fig.17.2.

Soluţie.

```
(DEFUN PUNE-PROP (GRAF)  ; GRAF=A-lista
(MAPC '(LAMBDA(Z)
     (PUTPROP (CAR Z) 'OUT (CONS (CDR Z) (GET (CAR Z) 'OUT)))
     (PUTPROP (CDR Z) 'IN    (CONS (CAR Z) (GET (CDR Z) 'IN)))
)
     GRAF) )
```

Presupunem că avem graful G dat cu ajutorul listelor de asociere (respectiv de arce). Să scriem o funcţie care calculează lista tuturor nodurilor din care pleacă sau în care intră cel puţin un arc. Atenţie! nu este lista nodurilor grafului, deoarece graful poate avea şi noduri izolate.

Exemplu . (NODURI '((a . b) (b . c) (d . a) (b . d) (e . b))) → (C A D E B)

Soluţie.

```
(DEFUN NODURI (L) ; L=graf ca A-lista
(REMOVE-DUPLICATES
        (MAPCAN #'(LAMBDA(Z) (LIST (CAR Z) (CDR Z))) L)
))
```

Determinarea listei nodurilor OUT **(PUN-OUTnoduri)**

Fie graful G dat cu ajutorul proprietăţii IN. Scriem o funcţie care defineşte proprietatea OUT pentru fiecare nod.

Exemplu . Considerăm graful dinainte, Fig. 17.2, dat ca A-listă:

(SETF GRAF '((A . B) (B . C) (D . A) (B . D) (E . B)))

(SETF NODURI '(C A D E B)) → (C A D E B)

Avem: (PUTPROP 'A 'IN '(D)) (PUTPROP 'B 'IN '(A E))

(PUTPROP 'C 'IN '(B)) (PUTPROP 'D 'IN '(B))

:(PUN-OUT NODURI) va efectua:

(PUTPROP 'A 'OUT '(B)) (PUTPROP 'B 'OUT '(C D))

(PUTPROP 'D 'OUT '(A)) (PUTPROP 'E 'OUT '(B))

Amintim că lista proprietăţilor unui simbol se face prin:

:(SYMBOL-PLIST 'A)

(OUT (B) IN (D))

Vrem să afişăm toate nodurile cu toate proprietăţile:

(PROPRIETATI-GRAF NODURI) → ((C IN (B)) (A OUT (B) IN (D))
(D OUT (A) IN (B)) (E OUT (B)) (B OUT (D C) IN (A E)))

Soluţie.

```
; Proprietatea OUT stiind pe IN, scris recursiv
(DEFUN PUN-OUT(NODURI); NODURI=lista nodurilor neizolate din Graf
(COND ((NULL NODURI) NIL)
        (T (IESIRE (CAR NODURI)(GET (CAR NODURI) 'IN))
            (PUN-OUT (CDR NODURI)) )
))
```

```
; Cînd scriem recursiv, deobicei avem nevoie de mai multe funcţii.
(DEFUN IESIRE (A L); A=nod, L=lista nodurilor care intra in A
(COND ((NULL L) NIL)
      (T (PUTPROP E 'OUT (CONS A (GET E 'OUT)) )
         (DEF-IESIRI A (CDR L)) )
))
; Iată o variantă cu DOLIST a lui PUN-OUT
(DEFUN PUN-OUT (NODURI) ; NODURI-lista nodurilor grafului
(DOLIST (I NODURI); iau pe rind toate nodurile I
  (DOLIST (J (GET I 'IN)) ; iau pe rind nodurile J care intra in I
      (PUTPROP J 'OUT (CONS I (GET J 'OUT)))
))
Afişează toate proprietatile tuturor nodurilor din GRAF
(DEFUN PROPRIETATI-GRAF (NODURI)
 (MAPCAR '(LAMBDA (Z) (CONS Z (SYMBOL-PLIST Z)))
         NODURI))
```

Noduri accesibile dintr-un nod dat **(ARCE nod graf)**

Fie graful G dat cu ajutorul listelor de asociere. Pentru un nod V dat să determinăm lista tuturor nodurilor V' astfel încît să existe arc de la V la V' (nodurile care ies din V).

Exemplu. (ARCE 'B '((A . B)(B . C)(D . A)(B . D)(E . B))) → (C D)

Soluţie.

```
(DEFUN ARCE (X L); X nod iar L graf ca A-lista
(MAPCARN #'(LAMBDA(Z) (IF (EQUAL X (CAR Z)) (CDR Z) )) L))
```

Grad exterior, interior şi total **(OD n graf)(ID n graf)(TD n graf)**

Gradul interior al lui x este egal cu numărul de arce care intră în x:

$$ID(x) = \text{card}\ \{\ v \in V\ ;\ (v\ ,\ x) \in E\ \}$$

Gradul exterior al lui x este egal cu numărul de arce care ies din x:

$$OD(x) = \text{card}\ \{\ v \in V\ ;\ (x\ ,\ v) \in E\ \}$$

Gradul total este suma dintre gradul exterior şi gradul interior al lui x:

$$TD(x) = ID(x) + OD(x)$$

Iată funcţiile care calculează *gradul exterior* (OD), *interior* (ID) şi *total* (TD) pentru un nod X din graful G reprezentat ca listă de asociere.

pentru un nod X din graful G reprezentat ca listă de asociere.
Exemplu. (OD 'A '((A . B) (B . C)(D . A) (B . D) (E . B))) → 1
(ID 'E '((A . B) (B . C)(D . A) (B . D) (E . B))) → 0
(TD 'B '((A . B) (B . C)(D . A) (B . D) (E . B))) → 4
Soluţie.

```
(DEFUN OD (NOD GRAF) ;nr arce care ies din NOD
(APPLY '+ (MAPCAR '(LAMBDA(Z)(IF (EQUAL (CAR Z) NOD) 1 0))
                                    GRAF)) )
(DEFUN ID (NOD GRAF) ; nr arce care intra in NOD
(APPLY '+ (MAPCAR '(LAMBDA(Z)(IF(EQUAL (CDR Z) NOD) 1 0))
                                    GRAF)) )
(DEFUN TD (NOD GRAF) ; total arce care intra si ies din NOD
(+ (ID NOD GRAF) (OD NOD GRAF)) )
```

17.3. Drum într-un graf

Fie graful G = (V , E), unde V este mulţimea nodurilor şi E este mulţimea arcelor. Numim ***drum*** în graful G o mulţime de arce (i_1, i_2), (i_2 , i_3), ... , (i_{n-1} , i_n) avînd extremitatea iniţială a fiecărui arc (cu excepţia primului) egală cu extremitatea finală a precedesorului. Pe scurt putem scrie $(i_1, i_2,...,i_{n-1} , i_n)$.
Nodurile i_1 şi i_n se numesc ***extremitatea iniţială*** respectiv ***finală*** a drumului $(i_1, i_2,...,i_{n-1} , i_n)$.
Numim ***drum elementar*** un drum în care nu se repetă nici un nod.
Numim ***ciclu*** un drum în care extremitatea iniţială coincide cu cea finală.
Numim ***ciclu elementar*** un ciclu care nu conţine subcicluri proprii.
Spunem că există ***lanţ*** între două noduri x şi y dacă există $i_1 , i_2 , ... , i_n \in V$, astfel încît i_1=x, i_n =y şi pentru k = 2,...,n , (i_{k-1} , i_k) sau (i_k , i_{k+1}) este ***arc***.
Un graf este ***conex*** dacă între orice două noduri distincte ale lui există măcar un lanţ .
Un graf este ***tare conex*** dacă între orice două noduri distincte ale lui există măcar un drum .

Drum elementar între două noduri (DRUM nod1 nod2 graf)
Avem un graf G=(V , E) şi două noduri distincte I , J. Să determinăm un drum elementar de la nodul I la nodul J .

Exemplu. (DRUM 'A 'D '((A . B)(B . C)(D . A)(B . D))) → ((A B D))
Soluţie.

```
(DEFUN DRUM (X Y L); X si Y noduri, L =graf ca lista de arce
(PUTPROP X T 'MARCAT) ; marcam nodurile prin care trecem
(COND ((EQUAL  X Y)(LIST(LIST X)))
        ((MEMBER Y (ARCE X L))(PUTPROP Y T 'MARCAT)
                                (LIST (LIST X Y)) )
        (T (MAPCAN #'(LAMBDA(Z) (IF (GET Z 'MARCAT) NIL
          (MAPCAR #'(LAMBDA(Z1) (CONS X Z1)) (DRUM Z Y L))))
(ARCE X L) ))
))
```

Exerciţii. 17. Grafuri

1. Scrieţi un program pentru citirea unui graf G, dat prin definirea proprietăţii IN pentru fiecare nod al său.
2. Calculaţi dintr-un graf dat, nodul din care ies cele mai multe arce.
3. Scrieţi un program GRAFURI care oferă cît mai multe facilităţi de explorare a unui graf dat.
4. Fie un grup de ***n*** persoane, cu relaţia de ***I cunoaşte pe J***. Numim ***celebritate*** acele noduri I care nu cunosc pe nimeni şi sunt cunoscute de toţi ceilalţi. Calculaţi celebrităţile unui graf.
5. Scrieţi o funcţie START care să citească graful (ca listă de asociere) şi să printeze un drum elementar între două noduri citite de la consolă.
6. Scrieţi o funcţie care să facă trecerea de la reprezentarea cu ajutorul proprietăţii OUT la reprezentarea prin proprietatea IN.
7. Scrieţi un program care să calculeze OD, ID şi TD pentru toate nodurile unui un graf reprezentat ca listă de arce.
8. Scrieţi o funcţie care să întoarcă lanţurile de la I la J dintr-un graf.
9. Verificaţi dacă un graf este conex şi tare conex.

18. Arbori

18.1. Arbori.Generalităţi

Numim ***arbore*** un graf conex şi fără cicluri. De exemplu (Fig.18.1).

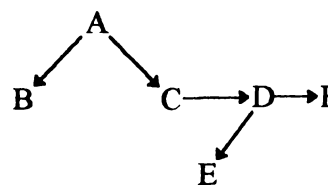


Fig. 18.1. Un arbore

Un ***arbore direcţionat*** este un digraf asimetric cu proprietatea că graful suport corespunzător lui este graf arbore. Arborele direcţionat Arb =(V , E) este ***arbore cu rădăcină*** dacă există r ∈ V astfel încît pentru orice v ∈ V , v ≠ r există drum de la r la v în Arb. În acest caz R se numeşte ***rădăcină*** .

Dacă Arb = (V , E) este arbore direcţionat cu rădăcină, atunci:

1. Pentru orice v ∈ V , v ≠ r , nu există drum de la v la r în Arb.
2. Rădăcina r este unică.
3. Dacă r este nod rădăcină, atunci gradul interior al lui r, id(r) = 0 şi pentru orice v ∈ V , v ≠ r, id(v) = 1

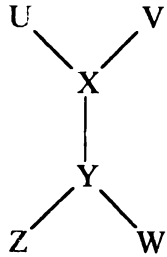


Fig.18.2. Arbore fără rădăcină

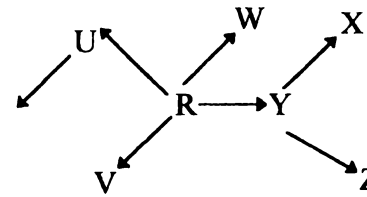


Fig.18.3. Arbore cu rădăcină

Convenţional, reprezentările grafice pentru arborii direcţionaţi cu rădacină corespund orientărilor pentru săgeţi de sus în jos. De exemplu, reprezentarea convenţională pentru arborele din Fig.18.3 este:

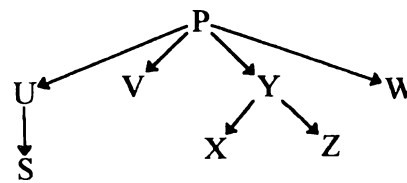


Fig.18.4. Arborele din Fig. 18.3 altfel desenat

Reprezentarea arborilor

Iată câteva modalităţi de reprezentare pentru arbori (vezi Fig.18.4):

A.Reprezentarea arborilor ca grafuri

(SETF ARBORE '((R . U) (R . V)(R .Y) (R .W) (U . S) (Y . X) (Y . Z)))

B. Reprezentarea arborilor cu ajutorul proprietăţii FIU

(PUTPROP 'R 'FIU) → (U V Y W)
(PUTPROP 'U 'FIU) → (S)
(PUTPROP 'Y 'FIU) → (X Z)

C. Lista descendenţilor unui nod reprezintă valoarea nodului

(SETF R '(U V Y W))	(SETF U '(S))	(SETF Y ' (X Z))
(SETF V NIL)	(SETF W NIL)	(SETF S NIL)
(SETF X NIL)	(SETF Z NIL)	

Amintim că pentru a obţine lista fiilor lui R:

R → (U V Y W) sau (EVAL 'R) → (U V Y W)

18.2. Parcurgerea arborilor

Studiem următoarele metode de parcurgere:
- Metoda ***depth***-first (parcurgerea arborilor în ***adîncime***)
- Metoda ***breadth***-first (parcurgerea arborilor în ***lățime***, pe nivele)

Metoda depth-first **(DEPTH nod)**

Prin metoda depth-first parcurgem arborele în ***adîncime*** astfel:
- plecăm din nodul V dat şi trecem mereu la primul dintre descendenții nodului curent, atîta timp cît există descendent.
- cînd nu există descendenți, trecem la următorul descendent al tatălui nodului curent.

Iată o funcție pentru parcurgerea unui arbore prin metoda depth-first.

Exemplu. (DEPTH 'R) → (R U S V Y X Z W)

Soluție. Arborele este reprezentat prin valoare nod = lista descendenților săi (varianta C). Vezi Fig.18.4

```
(DEFUN DEPTH (RAD); RAD= nodul radacina
(IF (NULL (EVAL RAD)) (LIST RAD)
    (CONS RAD (MAPCAN #'DEPTH (EVAL RAD)))
))
```

Parcurgere breadth-first **(BREADTH nod)**

Parcurge în ***lățime*** (breadth) a arborelui: vizităm nodul V, apoi vecinii lui, apoi vecinii nevizitați ai vecinilor, etc. Iată funcția pentru parcurgerea unui arbore binar prin metoda breadth-first.

Exemplu. (BREADTH 'R) → (R U V Y W S X Z)

Soluție. Arborele este reprezentat prin valoare nod = lista descendenților săi (varianta C).

```
(DEFUN BREADTH (RAD) ; RAD = nodul radacina
(LABELS ((BR-AUX (NIVEL LISTA)
(IF  (NULL NIVEL) LISTA
        (BR-AUX  (MAPCAN #'(LAMBDA(X)(EVAL X)) NIVEL)
                        (APPEND LISTA NIVEL) ))
      ))
(BR-AUX (LIST RAD) NIL)
))
```

Un *arbore binar* este un arbore orientat în care *fiecare nod are cel mult doi descendenţi*, făcînd distincţie între descendentul **stîng** şi descendentul **drept.** Vom prezenta cîteva reprezentări a arborilor binari cu exemplificare din Fig. 18.5

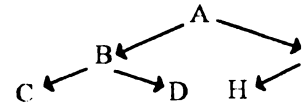


Fig. 18.5. Un arbore binar

1.Reprezentarea unui arbore binar ca listă de forma:
(rădăcină subarbore-stîng subarbore-drept)

(A (B (C NIL NIL)(D NIL NIL)) (G (H NIL NIL) NIL))

(subarbore-stîng rădăcină subarbore-drept)

(((NIL C NIL) B (NIL D NIL)) A ((NIL H NIL) G NIL))

(subarbore-stîng subarbore-drept rădăcină)

(((NIL NIL C) (NIL NIL D) B) ((NIL NIL H) NIL G) A)

2.Reprezentarea unui arbore binar ca liste de asociere:
(NOD . (STINGA . DREAPTA))

(A . ((B . (C . D)) . (G . (H . NIL))))

3. Reprezentarea unui arbore binar prin liste de proprietăţi:
Fiecărui nod i se asociază proprietăţile STINGA şi DREAPTA. Proprietatea STINGA a nodului V are valoarea egală cu descendentul stîng al nodului V, iar proprietatea DREAPTA are valoarea egală cu descendentul drept al nodului.

Proprietăţile STINGA, DREAPTA pentru graful din Fig.18. 5

NOD	Stinga	Dreapta
A	B	G
B	C	D
C	NIL	NIL
D	NIL	NIL
G	H	NIL
H	NIL	NIL

Parcurgerea arborilor binari

Parcurgerea arborilor în preordine şi postordine (presupune a ataşa un arbore

Parcurgerea arborilor binari

Parcurgerea arborilor în preordine şi postordine (presupune a ataşa un arbore binar unui arbore oarecare).

Ne punem din nou problema parcurgerii arborilor, dar acum ştim că avem de aface cu **arbori binari**.

Prezentăm parcurgerile arborilor binari cunoscute sub numele de parcurgere în *inordine*, *preordine* şi *postordine*.

În toate cele trei strategii de parcurgere, pentru fiecare nod vizităm întîi subarborele *stîng* şi apoi subarborele *drept*.

La parcurgerea în **preordine** vizitarea nodului se face înainte de vizitarea subarborelui stîng, la parcurgerea în **inordine** vizitarea nodului se face între vizitarea subarborelui stîng şi a celui drept şi la parcurgerea în **postordine**, vizitarea nodului se face după vizitarea celor doi subarbori.

Iată pentru arborele binar din Fig. 18.5 cum arată aceste parcurgeri:

preordine: Rădăcina, Stînga, Dreapta: A B C D G H
inordine: Stînga, Rădăcina, Dreapta: C B D A H G
postordine: Stînga , Dreapta , Rădăcina: C D B H G A

Parcurgerea arborilor în INORDINE (INORDINE arbore)

Să scriem o funcţie pentru parcurgerea în inordine a unui arbore binar dat. Funcţia depinde de modul în care este reprezentat arborele binar.

Exemplu. Prin parcurgerea arborelui din Fig.18.5:

(INORDINE1 'A) → (C B D A H G)
(INORDINE2 '((A (B (C NIL NIL)(D NIL NIL))(G (H NIL NIL) NIL))) → (C B D A H G)

Soluţia 1. Arborele binar cu proprietăţile ST şi DR pentru fiecare nod.

```
(DEFUN INORDINE1 (RAD); RAD=radacina arborelui binar
(IF (NULL RAD) NIL
        (APPEND (INORDINE1 (GET RAD 'ST))
                (CONS RAD (INORDINE1 (GET RAD 'DR)))    )
))
```

Soluţia 2. Arbore reprezentat ca: (nod subarbore-sting subarbore-drept)

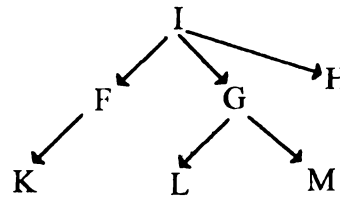
```
(DEFUN  INORDINE2 (X); X=arborele ca lista  (nod st dr)
(IF (NULL X) NIL  (APPEND (INORDINE2 (CADR X))
                          (CONS (CAR X) (INORDINE2 (CADDR X)))
))
```

Arborele binar asociat unui arbore oarecare

O modalitate standard de reprezentare a unui arbore orientat (pe nivele) oarecare, în care avem o ordine între descendenţii fiecărui nod, este să memorăm pentru fiecare nod I:

• proprietatea FRATE cu valoarea egală cu acel descendent al tatălui lui I care urmează imediat lui I

Fig. 18.6. Un arbore



NOD	FIU	FRATE
I	F	NIL
F	K	G
G	L	H
H	NIL	NIL
K	NIL	NIL
L	NIL	M
M	NIL	NIL

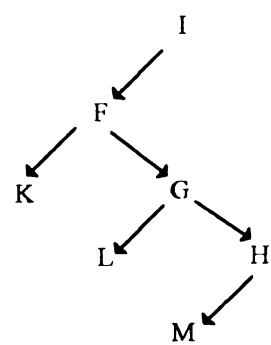


Fig. 18.7. Arborele binar asociat arborelui din Fig. 18.6

Parcurgerea în **preordine** constă în vizitarea rădăcinii urmată de vizitarea în

Parcurgerea în **preordine** constă în vizitarea rădăcinii urmată de vizitarea în preordine a subarborilor ce au ca rădăcini descendenţii acesteia. Prin parcurgerea în preordine a arborelui din Fig.18.7 obţinem: I F K G L M H

Parcurgerea în **postordine** constă în parcurgerea în postordine a subarborilor ce au ca rădăcini descendenţii rădăcinii arborelui urmată de vizitarea rădăcinii arborelui. Astfel arborele din Fig.18.7 devine

K F L M G H I

Parcurgerea arborilor în PREORDINE (PREORDINE rad)

Iată o funcţie pentru parcurgerea în preordine a unui arbore reprezentat cu ajutorul proprietăţilor FIU şi FRATE.

Exemplu. Pentru arborele din Fig.18. 7

(PREORDINE 'I) → (I F K G L M H)

Soluţie. Arbore reprezentat cu proprietăţie FIU şi FRATE.

```
(DEFUN PREORDINE (RAD); RAD- radacina
(IF (NULL RAD) NIL
(APPEND (CONS RAD (INORD (GET RAD 'FIU)))
        (INORD (GET RAD 'FRATE)))
))
```

Parcurgerea în preordine a unui graf este echivalentă cu parcurgerea în preordine a arborelui binar ataşat. Metoda de parcurgere DEPTH constituie o generalizare a parcurgerii în preordine a arborilor.

Parcurgerea arborilor în POSTORDINE (POSTORDINE rad)

Parcurgerea în postordine este echivalentă cu parcurgerea în inordine a arborelui binar ataşat.

Să scriem o funcţie pentru parcurgerea unui arbore reprezentat cu ajutorul proprietăţilor FIU şi FRATE în postordine.

Exemplu. Pentru arborele din Fig.18. 7

(POSTORDINE 'I) → (K F L M G H I)

Soluţie.

```
(DEFUN POSTORDINE (RAD)
(IF (NULL RAD) NIL
        (APPEND (INORD (GET RAD 'FIU))
                (CONS RAD (INORD (GET RAD 'FRATE))) )
))
```

Exerciţii. 18. Arbori

1. Scrieţi un program care introduce de la consolă un arbore binar reprezentat sub forma de liste de proprietăţi.
2. Testaţi dacă un graf dat este arbore binar sau nu.
3. Generaţi aleator grafuri neorientate. Cîte sunt arbori ? Proiect: ***Ordine în dezordine***.
4. Scrieţi programe pentru parcurgerea în preordine (respectiv postordine) a unui arbore binar (cu diferitele reprezentări).
5. Generaţi aleator arbori mari dintr-o mulţime fixă de noduri. Căutaţi locul unde se află un nod dat în arbore mai multe metode de căutare. Faceţi teste de timp. Comparaţi.
6. Scrieţi un TUTORIAL de explicat conceptul de arbore şi proprietăţile sale.
7. Acoperirea tablei de şah cu regine. Plasaţi cele 8 regine pe o tablă de şah astfel încît să nu se atace nu pe alta. Cîte soluţii sunt?

19. Sortare

Deseori trebuie să aranjăm liste în ordine, după un predicat care descrie o relaţie de ordine totală. Această situaţie o numim ***problema sortării după predicatul P.***

Deobicei numărul de înregistrări ce trebuie sortate este foarte mare, de aceea ne interesează algoritmii de sortare cu număr minim de comparaţii. În decursul timpului s-au dezvoltat numeroşi algoritmi de sortare, dintre care vom prezenta (vezi /5/, /21/) :

- sortare prin inserare în listă
- sortare prin interclasare de liste (MERGESORT)
- sortare rapidă (QUICKSORT)
- sortare prin inserarea elementelor într-un arbore
- sortare prin inversarea elementelor (BUBBLE SORT)

Sortare prin inserare în listă **(INSORT lista pred)**

Algoritmul de sortare prin inserare constă în parcurgerea listei iniţiale şi crearea unei liste ordonate după predicatul PRED astfel:

1. Începem cu lista rezultat vidă.
2. Parcurgem lista iniţială şi inserăm de fiecare dată elementul curent al acesteia în lista rezultat, astfel încît lista să rămînă sortată după predicatul P.
3. După ce am parcurs lista iniţială, lista rezultat conţine toate elementele acesteia, dar sortate.

Procedeul poate fi descris recursiv astfel:

Primul element al listei este inserat în lista formată prin sortarea restului listei inițiale.

Exemplu. lista inițială: (9 3 1 7 8) lista rezultat: (1 3 7 8 9)

1. Varianta iterativă: lista rezultat: ( )

inserăm 9 în lista rezultat: (9)

inserăm 3 în lista rezultat: (3 9)

inserăm 1 în lista rezultat: (1 3 9)

inserăm 7 în lista rezultat: (1 3 7 9)

inserăm 8 în lista rezultat: (1 3 7 8 9)

2. Varianta recursivă:

inserăm 9 în lista formată prin sortarea restului listei inițiale (3 1 7 8).

Sortarea listei (3 1 7 8) se face analog, inserînd 3 în lista (1 7 8), formată prin sortarea listei (1 7 8) și așa mai departe.

Scriem mai întîi INSER care inserează un element A într-o lista L sortată în raport cu un predicat P, astfel încît lista obținută să rămînă sortată după predicatul P.

Complexitatea. Numărul de comparații necesar în cazul cel mai defavorabil este 1+2+...+n = n(n+1)/2

Exemplu. (INSER 5 '(1 3 6 7) '<) → (1 3 5 6 7)

(INSORT '(3 1 5 8 9 7 '<) → (1 3 5 7 8 9)

Soluție.

```
(DEFUN INSER (A L P); A =element, L= lista, P = predicat 2-ar
(COND  ((NULL L)(LIST A))
       ((FUNCALL  P A  (CAR L))  (CONS A L)) ; caut loc pt  A
         (T (CONS (CAR L) (INSER A (CDR L) P)))
))

(DEFUN  INSORT (L P); L= lista, P=predicat
; inserează pe rînd fiecare element din L în L
(IF (NULL L) NIL  (INSER (CAR L) (INSORT (CDR L) P) P)  )
)
```

Sortare prin interclasare-MERGESORT (SORT-INTERCL L P)

Metoda sortării prin interclasare constă din sortarea primei jumătăţi a listei, apoi a doua jumătate a acesteia şi listele obţinute se interclasează. Complexitatea algoritmului este O(n × log2 n).

Exemplu. Pentru a sorta lista (9 3 1 7 8):

- sortăm prima jumătate: (9 3 1) obţinem (1 3 9)
- sortăm a doua jumatate: (7 8) obţinem (7 8)
- se interclasează listele (1 3 9) şi (7 8) obţinem lista (1 3 7 8 9)

Exemplu. (INTERCL '< '(1 3 5 7) '(2 4 6 8)) → (1 2 3 4 5 6 7 8)

(PARTEA1 '(3 1 5 2 4 7)) → (3 1 5)

(PARTEA2 '(3 1 5 2 4 7)) → (2 4 7)

(SORT-INTERCL '(3 1 5 8 9 2) '<) → (1 2 3 5 8 9)

```
(DEFUN INTERCL (P A B); P=predicat , A si B listele sortate
(COND ((NULL A) B)
        ((NULL B) A)
       ((FUNCALL P (CAR A) (CAR B)) (CONS (CAR A)
                                          (INTERCL P (CDR A) B ) )) )
        (T (CONS (CAR B) (INTERCL P A (CDR B) )))
))
(DEFUN SORT-INTERCL (L P)
(COND ((NULL (CDR L)) L )
        (T (INTERCL P (SORT-INTERCL (PARTEA1 L) P)
                (SORT-INTERCL (PARTEA2 L) P) )
))

(DEFUN PARTE1 (L)
(LET ((N (/ (LENGTH L) 2)))
(SUBSEQ L 0 N)
))
(DEFUN PARTE2 (L)
(LET ((N (/ (LENGTH L) 2)))
(SUBSEQ L N)
))
```

Sortarea rapidă- QUICKSORT **(QSORT lista pred)**

Ideea metodei este următoarea: se împarte lista L care trebuie sortată în două subliste: A şi B. In A sunt toate elementele mai mici decît primul element al lui L, iar în B sunt toate elementele mai mari decît primul element al lui L. Listele A şi B se sortează mai departe cu aceeaşi metodă. Rezultatul va fi lista formată alăturînd lista A, primul element al lui L şi lista B.

În cazul cel mai defavorabil complexitatea algoritmului este O (n2).

Exemplu.

Pentru sortarea listei (7 9 3 1 8), formăm listele după 7:

A= (3 1) elemetele mai mici ca 7 şi B= (8 9) elementele mai mari ca 7

Sortăm, cu aceeaşi metodă, listele A şi B. Acestea devin:

A= (1 3) şi B= (8 9)

Iar lista rezultat va fi concatenarea lui A , 7 şi B: (1 3 7 8 9)

Soluţie. Scriem o funcţie ajutătoare IMPART care împarte o listă în două subliste, prima conţinînd elementele mai mici sau egale decît un element X, cealaltă elementele mai mari decît X.

(IMPART 3 '(2 4 1 5) '<) → ((2 1) (4 5))

(QSORT '(3 1 5 8 9 2) '<) → (1 2 3 5 9 11)

```
(DEFUN IMPART (X L P &AUX L1 L2); L impartita in L1 si L2
(DOLIST (I L (LIST L1 L2))
(IF (FUNCALL P I X) (SETF L1 (CONS I L1))
      (SETF L2 (CONS I L2)))
))

(DEFUN QSORT(L P); L=lista, P=predicat 2-ar
(LABELS ((S-AUX (X L P)
        (APPEND (QSORT (CAR L) P) (LIST X)
                (QSORT (CADR L) P))
         ))
(IF (NULL L) NIL (S-AUX (CAR L)(IMPART (CAR L)(CDR L) P ) P))
))
```

Sortare prin inserare în arbore **(SORT-ARBORE lista pred)**

Sortarea unei liste de numere mai poate fi realizată şi prin metoda înserării în arbore:

Parcurgînd lista dată secvenţial se construieşte un arbore binar în care, pentru fiecare nod, descendentul stîng este mai mic, iar descendentul drept mai mare. Arborele se construieşte inserînd, de fiecare dată, în arborele deja existent (la început arborele vid) elementul curent al listei. Inserarea unui element în arbore se face astfel:

- dacă arborele este vid, se creează un arbore cu un singur nod: rădăcina egală cu elementul dat
- dacă arborele este nevid, se compară elementul cu rădăcina arborelui
- dacă el este mai mic, este inserat în subarborele arborelui, acesta este parcurs în inordine.

Reamintim că arborele este reprezentat prin

(NOD STINGA DREAPTA)

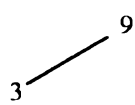
Exemplu.

Fie lista: (9 3 1 7 8) pe care o sortăm după predicatul >

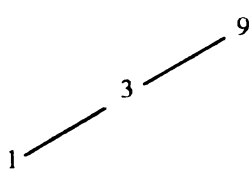
- arborele vid NIL
- se inserează 9 în arbore: (9 NIL NIL)

9

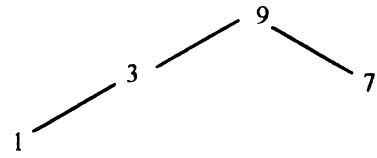
- se inserează 3 în arbore: (9 (3 NIL NIL) NIL)



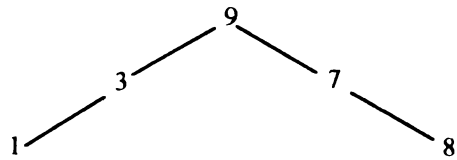
- se inserează 1 în arbore : (9 (3 (1 NIL NIL) NIL) NIL)



• se inserează 7 în arbore: (9 (3 (1 NIL NIL) NIL) (7 NIL NIL))



• se inserează 8 în arbore: (9 (3 (1 NIL NIL) NIL) (7 NIL (8 NIL NIL)))



• parcurgînd în inordine arborele format, obţinem lista sortată: (1 3 7 8 9)
Iată funcţia care sortează lista L după un predicat P, folosind metoda inserării în arbore.

Exemplu. (SORT-ARB '(9 1 3 7 8) '<) → (1 3 7 8 9)

Soluţie.

```
(DEFUN  SORT-ARB (L P)
   (INORDINE (ARBORE L P))
)
(DEFUN ARBORE (L  P); construirea arborelui
(LABELS  ((ARB (L TREE P)
       (IF (NULL L) TREE  (ARB (CDR L)(INSERT (CAR L) TREE P) P))
            ))
(ARB L NIL P)
))
(DEFUN INSERT (A  X  P); inserez elementul X in arbore la locul lui
(COND ((NULL X)(LIST A NIL NIL))
        ((FUNCALL P  A (CAR X))( LIST (CAR X)
                              (INSERT A (CADR X) P) (CADDR X)))
        (T (LIST (CAR X) ( CADR X) (INSERT A (CADDR X) P)))
))
```

Metoda bubble sort **(BUBBLE lista pred)**

Se compară pe rînd elementele alăturate şi se inversează ordinea lor dacă nu sunt cum trebuie (cum arată predicatul P). Se parcurge lista pînă cînd nu mai are loc nici o inversiune.

Iată un BUBBLE SORT scris intenţionat nerecursiv.

Care este complexitatea sa?

Exemplu. (BUBBLE '("Melania" "Anda" "Dan") 'STRING<) →
("Anda" "Dan" "Melania")

(BUBBLE '((1 A) (3 C) (4 D) (2 B))
'(LAMBDA(X Y)(< (CAR X)(CAR Y)))) → ((1 A) (2 B) (3 C)(4 D))

Soluţie.

```
(DEFUN SORT (L  P) ;L = lista, P = predicat
(PROG (NOU  I  AMTRECUT (LUNG (1- (LENGTH L))) X Y)
      DIN-NOU
(SETF NOU NIL AMTRECUT NIL I 0);amtrecut=nil nu s-a inversat nimic
      BINE
(SETF X (NTH I L)) ;X, Y elementele alaturate pe care le comparam
      RAU
(SETF Y (NTH (1+ I) L)) ; vedem lista ca un vector
(COND ((OR (EQUAL X Y)(FUNCALL P  X Y))
            (SETF NOU (CONS X NOU) I (1+ I))
            (IF (< I LUNG) (GO BINE)
      (AND (SETF NOU (CONS Y NOU)) (GO FINAL))))
      (T (SETF NOU (CONS Y NOU) AMTRECUT T I (1+ I))
       (IF (< I LUNG) (GO RAU)
      (AND (SETF NOU (CONS X NOU))(GO FINAL))) )
)
FINAL
(IF AMTRECUT  (AND (SETF L (REVERSE NOU))(GO DIN-NOU))
      (RETURN (REVERSE NOU)))
))
```

Exerciţii. 19. Sortare

1. Scrieţi recursiv algoritmul de sortare BUBLE-SORT.
2. Care este cel mai bun algoritm? Care este cea mai rapidă funcţie SORT?
3. Scrieţi un program care să testeze mai multe funcţii SORT pe aceleaşi exemple generate aleator şi să prezinte timpul de execuţie.
4. Imbunătăţiţi sortarea prin inserare în arbore.
5. Faceţi comparaţii de SORT în C, PASCAL, LISP, SCHEME pe mai multe sisteme şi calculatoare.

20. Logică

20.1 Calculul cu propoziţii (Cp)

Visul vechi al marelui gînditor G. W. Leibniz (1646-1716) aproape că s-a realizat: o maşină care să gîndească. Mai mult decît calcule vrem ca şi calculatorul să facă deducţii, să raţioneze logic. Încă de la începutul inventării limbajului LISP, primul deziderat al acestui limbaj a fost să proceseze uşor simboluri, deci şi simboluri logice. S-a scris în LISP primul demonstrator automat pentru calculul cu propoziţii. Vom încerca împreună să stăpînim manipularea în LISP a formulelor logice.

În calculul cu propoziţii notat pe scurt **Cp** avem obiecte de bază (atomi) ***propoziţiile*** pe care le considerăm **obiecte indivizile**. Propoziţiile au ca valoare de adevăr: **adevărat** şi **fals**. Ne situăm în cadrul logicii cu două valori (bivalentă) denumită şi ***logică chrysipiană***. Cu ajutorul propoziţiilor formăm formule mai complicate: formule propoziţionale, legînd propoziţiile prin conectori logici:

NOT, AND, OR, IMPLICA, ECHIVALENT etc.

Iată o formulă propoziţională:

F = (P AND Q) OR (NOT P)

în care au apărut numai propoziţiile P şi Q. Ne întrebăm *ce valoare de adevăr are formula?*, dacă ştim valorile de adevăr pentru P şi Q.

Conectorii logici

Conectorii pot fi unari (NOT), binari (AND, OR, ECHIVALENT, IMPLICA, etc) sau *n*-ari, după cum leagă una sau *n* variabile propoziţionale.

Sunt foarte multe moduri de a scrie conectorii clasici, vom da doar cîteva notaţii pe care le puteţi găsi în cărţile de logică:

NOT	NEGAŢIE	-	|	NON	NOT	
AND	CONJUNCŢIE		.	ŞI	^	&
OR	DISJUNCŢIE		+	SAU	ORI	V
IMPLICA	->	=>				
ECHIVALENT	<->	=	<=>			

Ca să apropiem cît mai mult reprezentările din logică cu notaţiile din LISP convenim să scriem şi noi **NIL** în loc de **F** (Fals) şi să alegem denumirile de conectori pe care limbajul LISP le are deja: AND, OR, NOT. Reamintim tabelele de adevăr ataşate conectorilor clasici cu care vom lucra. Au fost stabilite încă de la Aristotel (384-322 î.e.n).

P	Q	P AND Q	P OR Q	P -> Q	P <-> Q
T	T	T	T	T	T
T	NIL	NIL	T	NIL	NIL
NIL	T	NIL	T	T	NIL
NIL	NIL	NIL	NIL	T	T

Observaţi că putem să completăm tabelul în 16 feluri (din care conectorii prezentaţi sunt doar 4). Numărul 16 rezultă din: cîte funcţii putem avea definite pe {T,NIL}×{T,NIL} cu valori în {T,NIL}, adică $2^4 = 16$.

Faptul că în logică se folsesc cu precădere aceştia se datorează faptului că toţi ceilalţi pot fi exprimaţi cu uşurinţă ca şi compuneri din conectorii clasici.

20.2. Formula bine formată, fbf

Scrierea în *forma infix* este scrierea obişnuită pe care o folosim în matematică, cu alte cuvinte operatorul se află între operanzi.

În scrierea în *forma prefix* operatorul se pune la început şi operanzii după aceea. Iată aceeaşi expresie scrisă în *infix* şi *prefix*:

INFIX	PREFIX
A*B+ C* B	+ (* A B)(* C B))
P or (not Q)	(OR P (NOT Q))
(P & (NOT Q)) <-> Q &P	(<-> (& P (NOT Q)) (& Q P))
(P OR Q) -> (NOT P)	(-> (OR P Q) (NOT P))

Amintim că limbajul LISP foloseşte scrierea în forma ***prefix***.

Fie o formulă în Cp, deci avem o înşiruire după anumite reguli de: simboluri de variabile propozitionale (P, Q, R, S,etc), conectori (putem să-i numim şi ***operatori logici)*** -(OR AND NOT, -> , <->) şi paranteze. Pentru simplificare considerăm pe AND , OR ->, <- , <-> operatori binari pe care îi punem între paranteze.

Numim o ***formulă bine formată*** în calculul propoziţional dacă îndeplineşte regulile sintactice corecte ale punerii conectorilor. Pe scurt vom numi ***formula bine formată*** cu ***fbf*** (în engleză se spune ***well formed formula, wff)***.

Vrem să ştim dacă formula este corectă în ***forma infixată*** sau nu.

Dacă avem (INFIX? F) → T atunci formula F este ***fbf*** în INFIX.

Exemplu. (INFIX? '((P OR (NOT Q)) AND R)) → T

Deci scriem ((P OR Q) OR R) şi nu (P OR Q OR R) !

Soluţie. Verific dacă F este de forma (F1 CONECTOR F2). Unde F1 si F2 sunt la rîndul lor formule infixate. Sau F este de forma (NOT F).

```
(DEFUN  INFIX? (F); F formula in Cp in INFIX?
(COND  ((ATOM F) F) ; F e atom, deci variabila propozitionala
        ((EQUAL (CAR F)  'NOT)(INFIX? (CADR F)))
        ((MEMBER (CADR F)  '(AND NOT OR  - > <->))
                (AND (INFIX? (CAR F)) (INFIX? (CADDR F))))
))
```

Trecerea din infix în prefix **(PREFIX formula)**

Considerăm formula în infix, deci este formulă bine formată ***fbf***. Dacă este fbf în forma infixată va fi fbf şi în forma prefixată, depinde numai de hotărîrea noastră în ce formă să lucrăm. Dar pentru că LISP-ul foloseşte scrierea prefixată vom trece şi noi formula în PREFIX.

Urmărind în continuare principiul că ***dacă apropiem două reprezentări avem numai de cîştigat.*** Avem deci:

INFIX= (F1 conector F2)	**PREFIX = (conector F1 F2)**
((P OR Q) OR R)	(OR (OR P Q) R)
((NOT P) -> (R OR (P AND S)))	(-> (NOT P) (OR R (AND P S))
(P <-> (NOT (NOT P)))	(<-> P (NOT (NOT P)))

Exemplu.

(PREFIX '((P OR (NOT Q)) AND R)) → (AND (OR P (NOT Q)) R)

Soluţie.

```
(DEFUN  PREFIX (F)  ;transformarea unei fbf din INFIX in PREFIX
(COND ((ATOM F) F)
        ((EQUAL (CAR F) 'NOT) '(NOT ,(PREFIX (CADR F))))
        ((MEMBER (CADR F) '(AND OR  -> <-> <-))
            '(,(CADR F) ,(PREFIX (CAR F)) ,(PREFIX (CADDR F))))
        (T  F); intoarce formula
))
```

În ambele cazuri nu am considerat posibilitatea de a avea o formulă în genul (AND P AND) sau (P AND OR) care ar ieşi drept corectă la testul nostru, pentru că nu am luat în considerare decît formule corecte în infix.

Este o formulă în Cp bine formată? (FBF? f)

Scopul nostru este să pregătim cît mai bine formulele pentru a putea afla valoarea lor de adevăr, adică construirea unui "demonstrator automat". Vom începe cu metoda tabelului de adevăr şi apoi metoda principiului rezoluţiei. Am văzut condiţiile ca o formulă în Cp, care este o concatenare de semne de paranteze, operatori (AND, OR, NOT,...) şi semne de variabile propoziţionale, să fie formulă bine formată ***fbf.***

Vom lucra de acum în colo cu formulele în forma PREFIX.

Să scriem o funcţie care verifcă dacă formula prezentată la intrare în demonstratorul nostru este corectă în PREFIX.

Această etapă se referă la ***verificarea corectitudinii sintactice a formulelor.***

Exemplu. (FBF? '(AND P (-> P R))) →T

(FBF '(OR P P R)) → NIL (FBF '()) → NIL

(FBF '(AND (NOT Q) (-> P Q))) →T (FBF '(P)) → T

Soluţie.

```
(DEFUN  FBF? (F);daca F este o formula bine formata (fbf),
(COND  ((ATOM F) T)
        ((EQUAL (CAR F) 'NOT)
                        (AND (= (LENGTH F) 2)  (FBF?  (CADR F))))
        ((MEMBER (CAR F) '(AND  OR  - > <- <->))
                (AND (= (LENGTH F) 3)(FBF? (CADR F))
                                (FBF? (CADDR F))))
))
```

20.3. Valoarea unei formule fbf

Variabilele propoziţionale **(L-VARIABILE f)**

Fiind dată o formulă fbf (formula bine formată) a cărui sintaxă am verificat-o cu funcţia FBF?. Vrem să ştim lista variabilelor propoziţionale care apar în formulă.

Exemplu. (L-VARIABILE '(AND (NOT P) (OR P S))) → (P S)

Soluţie.

```
(DEFUN CONECTOR? (X)
        (MEMBER X '(NOT AND OR -> <-> <-)))
(DEFUN  L-VARIABILE (F) ; lista variabilelor dintr-o singura parcurgere
(LABELS  ((LV (F  R); in R tinem minte rezultatul care se construieste
(COND  ((NULL F) R)
        ((ATOM (CAR F)) (IF (OR (CONECTOR? (CAR F))
                                        (MEMBER (CAR F) R))
                (LV (CDR F) R) (LV (CDR F) (CONS (CAR F) R))))
        (T (LV (CDR F) (LV (CAR F) R)))
)))  (LV   F NIL)
))
```

Valoarea unei FBF **(CALCUL-fbf f l-var val)**

Să aflăm valoarea de adevăr (adevăr=T sau fals=NIL) a unei fbf (formule bine formate) din calculul propoziţional (Cp) pentru o anumită instanţiere a variabilelor (adică dăm valori variabilelor).

Această funcţie va fi folosită la calculul tabelului de adevăr al unei fbf. Avînd în vedere că vom da pe rînd tuturor variabilelor toate valorile posibile

nu punem mare accent pe asociere. Altfel trebuie să fim atenţi la asocierea variabila propoziţională - valoare, i.e. lui P îi corespunde T şi lui Q , NIL. Această funcţie se bazează pe un truc şi anume avem deja în LISP funcţiile AND, OR, NOT şi ele pot fi calculate de interpretorul nostru.

Exemplu. Calculăm valoarea formulei (AND (NOT P)(NOT Q)) pentru valorile (T şi NIL) adică P are valoarea T şi Q are valoarea NIL. Formez lista ((LAMBDA (P Q) (AND (NOT P) (NOT Q))) (T NIL))

(CALCUL-FBF '(AND (NOT P)(NOT Q)) '(P Q) '(T NIL))) → NIL

Formez ((LAMBDA(P Q)(AND (NOT P)(NOT Q))) (T NIL)))

Soluţie.

```
(DEFUN CALCUL-FBF (F  L-VAR VALORI)
; F =fbf , L-VAR= lista variabilor, VALORI=valorile asociate variabilelor
    (EVAL (CONS (LIST 'LAMBDA L-VAR F ) VALORI))
)
```

Tabelul de adevăr al unei fbf din Cp **(TABEL-adevar f)**

A scrie tabelul de adevăr al unei ***fbf***, înseamnă a calcula valorile ***fbf*** pentru toate combinaţiile posibile ale variabilelor. Iată cum ar fi acest tabel pentru: F = (AND (NOT P)(OR P Q))

P	Q	(NOT P)	(OR P Q)	F
T	T	NIL	T	NIL
T	NIL	NIL	T	NIL
NIL	NIL	T	NIL	NIL
NIL	T	T	T	T

Pentru a calcula tabelul de adevăr să scriem funcţia CALCUL-FBF pe care o aplicăm apoi pe fiecare linie a tabelului.

Exemplu. (COMBINATII '(T NIL) 2) → ((T T)(T NIL)(NIL T)(NIL NIL))

:(TABEL-ADEV '(AND (NOT P)(NOT Q)))

FORMULA= (AND (NOT P)(NOT Q))

Lista variabilelor= (Q P)

(NIL NIL) T

(NIL T) NIL

(T NIL) NIL

(T T) NIL

NIL

Soluţie.

```
(DEFUN  TABEL-ADEV (F) ;f  este  fbf avind conectorii:  and , or , not
 (LET* ((L (L-VARIABILE F))  (N (LENGTH L))
        (C (COMBINATII '(T NIL) N)))
(FORMAT T  "FORMULA = ~A"  F)
(FORMAT T  "Lista variabilelor=~A"  L)
 (DOLIST (I C)
       (PRINC I) (PRINT (CALCUL-FBF F L I))
 )))
```

Tipul unei fbf? **(TIPUL fbf)**

Tautologie înseamnă că formula este adevarată pentru orice valoare dată variabilelor. Este foarte important să ştim dacă o formulă este tautologie sau nu. Metoda tabelului de adevăr este bună (completă), căci ne generează toate combinaţiile şi astfel vedem dacă este adevarată sau nu. Totuşi observaţi că metoda tabelului presupune generarea tuturor combinaţiilor care cresc exponenţial cu numărul de variabile (2^n unde ***n*** este numărul de variabile). Modificînd uşor funcţia care calculează tabelul de adevăr al formulei *fbf* putem să vedem dacă este adevarată sau nu. O formulă poate fi:

- ***adevarată*** (sau ***tautologie***) cu valoarea T pe orice combinaţie
- ***inconsistentă*** cu valoarea NIL pentru orice combinaţie
- ***consistentă*** (sau ***satisfiabilă)*** cînd există combinaţii de T şi NIL pentru care este adevarată.

Exemplu. (TIPUL '(OR P (NOT P))) $\rightarrow$ F e tautologie

Soluţie.

```
(DEFUN TIPUL   (F )  ;daca F=fbf este tautologie
(LET* ((L (L-VARIABILE F))  (N (LENGTH L))
                  (C (COMBINATII `(T NIL) N)) R)
(SETF  R   (MAPCAR '(LAMBDA(Z)(CALCUL-FBF F L Z))  C))
(COND  ((EVAL (CONS  'AND R))  "F e tautologie")      ;este T peste tot
       ((NOT(EVAL (CONS  'OR R)))"F e inconsistenta") ;NIL peste tot
       (T  "F e consistenta")  ;are si T si NIL
) ))
```

Din păcate ca să putem hotărî că formula noastră *fbf* este o tautologie prin metoda tabelului trebuie să parcurgem toate liniile. Refaceţi programul astfel încît să vedeţi numai dacă formula este tautologie. Deci la primul NIL să ieşiţi din program.

20.4. Trecerea unei fbf în fnc

Pentru a aplica tehnicile specifice de demonstrare automată, care urmează, trebuie să pregătim puţin formula. Vrem să transformăm formulele cu care lucrăm în *forma normal conjunctivă* (*fnc*). În **fnc** formula este o conjuncţie de disjuncţii. În formulă nu mai există decît operatorii NOT, AND, OR. Iată o formulă fbf în *fnc:*

Exemplu. (A OR B OR C) & (A OR (NOT B)) & (B OR (NOT C)) sau
(AND (OR (OR A B) C) (OR A (NOT B)) (OR B (NOT C))) sau
((A B C) (A (NOT B)) (B (NOT C)))

Etapele transformării unei formule în **fnc** sunt:

1) Scoaterea implicaţiilor
2) Mutarea negaţiei pe variabilele propoziţionale
3) Transformarea disjuncţiilor în conjuncţii. Adică:
   (P OR (Q AND R)) devine (P OR Q) AND (P OR R)
4) Scoaterea operatorilor AND şi OR. În forma **fnc** ştim exact unde se află operatorii aceştia, deci nu mai este nevoie să-i scriem. Adică în loc să scriem
   (AND (OR P Q) (OR P (NOT R)))) scriem ((P Q) (P (NOT R)))

Dacă în urma transformărilor unei formule F obţinem în final lista vidă înseamnă că formula este tautologie.

Prezentăm funcţia TRANSFORMA care reuneşte toate aceste etape. Vom lua apoi fiecare funcţie în parte.

Exemplu. (TRANSFORMA '(OR P (NOT P))) → NIL ; F este tautologie
:(TRANSFORMA '(OR (NOT (AND P (NOT Q))) (-> P Q)))
Scot-implicatii= (OR (NOT (AND P (NOT Q))) (OR (NOT P) Q))
Mut-negatii= (OR (OR (NOT P) Q) (OR (NOT P)Q))
Transform fnc= (OR (OR (NOT P) Q) (OR (NOT P) Q))
Transform in clauze= ((NOT P) Q) (NOT P) Q))

```
Sortez clauzele= (Q Q (NOT P) (NOT P))
Scot repetitiile din fiecare clauza=  ((Q  (NOT P)))
Scot tautologiile=  ((Q  (NOT P)))
Scot clauzele egale= ((Q  (NOT P)))

(DEFUN TRANSFORMA (F) ; F = fbf  in PREFIX
(PRINC  "Scot-implicatii= ") (PRINT (SETF  F (SCOT-IMPLICATII F)))
(PRINC  "Mut-negatii= ") (PRINT (SETF  F (MUT-NOT F)))
(PRINC  "Transform fnc= ") (PRINT (SETF F (FNC F)))
(PRINC  "Transform in clauze= ") (PRINT (SETF  F (CLAUZE F)))
(PRINC  "Sortez clauzele= ")
(PRINT (SETF  F (MAPCAR '(LAMBDA (Z)(SORT Z '$<)) F)) )
(PRINC  "Scot repetitiile din fiecare clauza= ")
(PRINT (SETF  F (MAPCAR 'REMOVE-DUPLICATES F)))
(PRINC  "Scot tautologiile=  ")
(PRINT (SETF  F (ELIMINA-TAUTOLOGIE F)))
(PRINC  "Scot clauzele egale= ")
(PRINT (SETF  F (REMOVE-DUPLICATES F)) )
)
```

Scot din implicaţiile din fbf **(SCOT-IMPLICATII f)**

Deci avem formula bine formată *fbf* din calculul propoziţional. Este pregatită pentru demonstratorul pe care-l construim, deci este în forma prefix. Dorim să scoatem semnele de <-, -> , <-> şi să le scriem cu ajutorul lui AND, OR şi NOT. Avem echivalenţele logice:

```
(-> P Q)    echivalent cu (OR (NOT P) Q)
(<- P Q)    echivalent cu (OR (NOT Q) P)
(<-> P Q)   echivalent cu (AND (-> P Q) (<- P Q))
```

Exemplu. (SCOT-IMPLICATII '(-> (AND (NOT P) Q) P)) -→
(OR (NOT (AND (NOT P) Q)) P)

Soluţie.

```
(DEFUN   SCOT-IMPLICATII (F) ;scot <-, -> ,<-> din P=fbf
(COND ((ATOM F) F)
```

```
        ((EQ (CAR F) '->) '(OR  (NOT ,(SCOT-IMPLICATII (CADR F)))
                                ,(SCOT-IMPLICATII (CADDR F)) ))
        ((EQ (CAR F) '<-) '(OR  ,(SCOT-IMPLICATII (CADR F))
                                (NOT  ,(SCOT-IMPLICATII (CADDR F))) ))
        ((EQ (CAR F) '<->) '(AND
                        ,(SCOT-IMPLICATII (-> (CADR F) (CADDR F)))
                        ,(SCOT-IMPLICATII (<- (CADR F)(CADDR F))) ) )
       (T (MAPCAR  'SCOT-IMPLICATII  F))
 ))
```

Mut negaţia pe variabilele propoziţionale **(MUT-NOT fbf)**

În forma forma normală conjunctivă **fnc** a unei *fbf* din Cp - trebuie să avem negaţiile pe simbolii propoziţionali. Cunoaştem din logica clasică echivalenţele:

(NOT (NOT P)) = P
(NOT (P OR Q)) = (NOT P) AND (NOT Q)
(NOT (P AND Q)) = (NOT P) OR (NOT Q)

Numim ***dualul*** operatorului AND pe OR, iar ***dualul*** lui OR este AND.

(DUAL 'OR)→OR (DUAL 'AND) → AND

Scriem funcţia DUAL care ne va simplifica funcţia MUT-NOT.

```
(DEFUN DUAL (OP); dualul operatorului AND , OR
(COND  ((EQ  OP 'AND) 'OR)
       ((EQ  OP 'OR) 'AND)
       (T   OP)
))
```

Exemple. (MUT-NOT '(NOT (OR P Q))) → (AND (NOT P) (NOT Q))

```
(DEFUN  MUT-NOT (F);mut conectorul NOT linga variabilele prop
(COND ((ATOM F) F) ; F=fbf din Cp in PREFIX fara implicatii
      ((EQ (CAR F) 'NOT)
         (COND ((ATOM (CADR F)) F)
               ((EQ (CAADR F) 'NOT) (MUT-NOT (CADADR F)))
                 (T (MUT-NOT '(,(DUAL (CAADR F))
                                ,(LIST 'NOT (CADADR F))
                                ,(LIST 'NOT (CADDADR F)))))  ))
         (T (MAPCAR 'MUT-NOT F))
))
```

Avem formula fbf deja într-o formă apropiată de scopul final.
(MUT-NOT (SCOT-IMPLICATII F)) = F cu care lucrăm
- nu conţine alte semne de conectori decît NOT, AND, OR
- negaţia NOT este lipită de atomi

Dorim să scriem formula ca o conjuncţie de disjuncţii. Numim *conjuncţie* aplicarea operatorului AND şi *disjuncţie* aplicarea lui OR.

Exemplu. (FNC '(AND P Q)) → (AND P Q)
(FNC '(OR (AND P Q) R)) → (AND (OR P R) (OR Q R))
(FNC '(OR (AND (NOT P) Q)(OR R Q))) →
(AND (OR (NOT P) (OR R Q)) (OR Q (OR R Q)))

Soluţie.

```
(DEFUN  FNC (L)
( IF  (OR (ATOM L ) (< (LENGTH  L)  3))  L
   (LET ((OP (CAR L)) (F1 (CADR L)) (F2 (CADDR L))  )
   (COND ((EQ OP 'AND) '(AND ,(FNC F1) ,(FNC F2) ))
         ((EQ OP 'OR)
           (COND ((AND (LISTP F1) (EQ (CAR F1) 'AND))
                    '(AND ,(FNC (LIST 'OR  (CADR F1) F2))
                          ,(FNC (LIST 'OR (CADDR F1) F2)) ))
                 ((AND (LISTP F2) (EQ (CAR F2) 'AND))
                          (FNC  '(OR ,F2 ,F1)) )
                 ((NU-APARE? 'AND L)  L)
                 (T (FNC  '(OR ,(FNC  F1) ,(FNC F2))))
         )))
)))

(DEFUN  NU-APARE? (E L); cauta atomul E pe toate nivelele in lista L
        (NOT (APARE? E L))
)
```

Numim *literali* variabilele propoziţionale negate sau nenegate.
Numim *clauze* disjuncţiile de literali, formate din grupările legate cu AND dintr-o formulă scrisă în forma normal conjunctivă. Fie formula
F = (AND (OR (OR P Q) R) (OR P (NOT Q)) (OR (OR P R) (NOT Q)))
Sunt **literali**:
P (NOT Q) (NOT (NOT P))
Apar trei **clauze**:
(OR P Q R) (OR P (NOT Q)) (OR P R (NOT Q))
Deoarece clauzele sunt întotdeauna grupări de disjuncţii, nu mai este nevoie să scriem conectorul OR. Deci putem scrie numai:
(P Q R) (P (NOT Q)) (P R (NOT Q))
Deci putem scrie pe F= ((P Q R) (P (NOT Q)) (P R (NOT Q))
Scopul nostru ar fi să transformăm formula F în această formă prescurtată, fără conectori. Parcurgem două etape:
A. Scriem conectorii ca şi cum ar fi *n*-ari (am considerat pînă acum că OR şi AND sunt conectori binari)
(AND P (AND R Q)) devine (AND P R Q)
B. Scoatem semnele de operatori: AND şi OR.
(CLAUZE '(AND (OR P (NOT Q)) (OR P (OR P R)))) →
((P (NOT Q)) (P P R))

Soluţie.

```
(DEFUN  SCOT-OP (OP  LISTA) ; OP=conector
(COND ((NU-APARE? OP  LISTA) (LIST LISTA))
      (T (APPEND (SCOT-OP  OP (CADR LISTA))
                 (SCOT-OP  OP  (CADDR LISTA)) ))
))
(DEFUN  CLAUZE (F) ; F=formula
(MAPCAR #'(LAMBDA(Z) (SCOT-OP 'OR Z))
                     (SCOT-OP 'AND F)) )
```

Scot tautologii din *fnc*

Considerăm mai departe formula *fbf* în forma *fnc* (ca o conjuncţie de clauze disjunctive). Reamintim că clauzele sunt disjuncţii de literali, iar o ***tautologie*** este o formulă adevarată pentru orice valori date variabilelor propoziţionale. În transformarea formulei putem ajunge la situaţii de genul:

F=((P (NOT P) Q) (P Q))

Clauza (P (NOT P) Q) poate fi scoasă, ea fiind o tautologie. Scoaterea ei neafectînd valoarea de adevăr a lui F, ci dimpotrivă poate să ne încurce în algoritmul de demonstrare automat propus în continuare.

Exemple. (TAUTOLOGIE? '(P Q (NOT P))) → T

(TAUTOLOGIE? '(P Q (NOT R))) → NIL

(ELIMINA-TAUTOLOGII '((P Q (NOT R)) (P R (NOT R)))) →
((P Q (NOT R)))

Soluţie.

```
(DEFUN TAUTOLOGIE? (CLAUZA); CLAUZA= literali legati prin OR
(COND ((NULL  CLAUZA) NIL)
        ((MEMBER (OPUS (CAR CLAUZA)) (CDR CLAUZA)
                                                :TEST 'EQUAL) T)
         (T (TAUTOLOGIE? (CDR CLAUZA)))
))
(DEFUN ELIMINA-TAUTOLOGII (F)
(MAPCARN '(LAMBDA(Z)(IF (TAUTOLOGIE? Z) NIL Z)) F)
)
(DEFUN OPUS (LIT); LIT=literal este P sau (NOT P)
(IF (ATOM LIT) '(NOT ,LIT) (CADR LIT) )
)
```

Ordonarea literalilor ($< literal1 literal2)

Pentru a compara mai repede dacă două formule sunt egale, desigur dacă ele sunt ordonate, comparaţia se face mai repede. Pentru aceasta să stabilim o relaţie de ordine între numele literalilor. Vom alege o extindere a ordinii alfabetice.

Exemplu. ($< 'P '(NOT Q)) →T ($< 'P 'Q) →T

Soluţie.

```
(DEFUN $< (P Q) ;predicat pentru ordinea literalilor
(COND ((AND (ATOM P)(ATOM Q))
                    (STRING< (STRING P) (STRING Q)))
      ((ATOM P) T)
      ((ATOM Q) NIL)
      (T ($< (CADR P)(CADR Q)))
))
```

20.5. Principiul rezoluţiei

Rezolventul a două clauze **(REZOLVENT c1 c2)**

Fie două clauze c1 = (P Q (NOT R)), c2 = (H R S). Numim ***rezolventul*** lor clauza c3= (H P Q S), obţinută prin ştergerea unui literal opus R şi (NOT R) şi adăugarea tuturor celorlalţi literali.

În cazul în care clauzele sunt de forma c1= (P) şi c2= ((NOT P)), rezolventul lor este NIL (vid). Sunt cazuri în care două clauze nu au nici un rezolvent. Iată cîteva exemple de clauze cu rezolventul lor:

```
c1= (R (NOT S))     c2= (R Q)      nu au rezolvent
c1= (P (NOT R))     c2= ((NOT P) S)  rezolvent= (S (NOT  R))
c1= (P)             c2 (H (NOT Q))   nu au rezolvent
c1= (P Q (NOT R))   c2 = (H  R  S )  rezolvent= (H P Q S)
c1= (P (NOT R))     c2= ((NOT P) S)  rezolvent= (S (NOT  R))
```

Soluţie.

```
(DEFUN  REZOLVENT (C1 C2 &AUX  OP REZ ); C1 si C2=clauze
(DOLIST (I C1 REZ) ; OP=literal opus care se cauta in C2
  (IF (MEMBER (SETF OP (OPUS I)) C2 :TEST 'EQUAL)
   (RETURN
        (IF (NULL (SETF R (APPEND (REMOVE I C1 :TEST 'EQUAL)
                    (REMOVE OP C2 :TEST 'EQUAL) )))
            'CL-VIDA
            (REMOVE-DUPLICATES (SORT R '$<)) ))
)))
```

Principiul rezoluţiei **(PRINCIPIU-REZ fnc)**

Principiul rezoluţiei afirmă că dacă căutăm toţi rezolvenţii posibili ai unui sistem de clauze, şi obţinem rezolventul vid, sistemul de clauze este inconsistent, altfel nu este inconsistent. Principiul rezoluţiei este o ***procedură de respingere***, numit astfel pentru că noi urmărim să obţinem ***falsul***. Problema iniţială este rezolvată practic prin metoda reducerii la absurd. Luăm formula dată (problema) şi adăugăm la ea concluzia negată. Din sistemul rămas dacă deducem falsul (rezolventul vid), problema noastră este adevărată.

Să observăm raţionamentul pe formula în fnc:

((A B C) ((NOT A))((NOT B)) ((NOT C))

Formăm pe rînd toţi rezolvenţii posibili.

PAS 1:

S= (((NOT C)) ((NOT B))((NOT A)) (A B C))

S1=((A B C) ((NOT A))((NOT B)) ((NOT C)))

PAS 2:

S= (((NOT C)) ((NOT B)) ((NOT A)) (A B) (A C)(B C) (A B C))

S1= ((B C) (A C) (A B))

PAS 3:

S= ((B)(C)(A)((NOT C))((NOT B))((NOT A))(A B)(A C)(B C)(A B C))

S1= ((A) (C) (B))

Din c1 = ((NOT A)) şi c2= (A) obţinem rezolventul vid (NIL) în consecinţă sistemul de clauze este inconsistent (QED- ***quod erat demonstrandum*** - ceea ce aveam de demonstrat).

Exemplu. Introducem formula: (AND P (AND (-> P Q) (NOT Q)))

Întîi transformăm formula în clauze după tehnicile prezentate:

(TRANSFORMA ' (AND P (AND (-> P Q) (NOT Q)))) →

((P)(Q (NOT P)) ((NOT Q)))

Acum putem lansa demonstratorul nostru bazat pe principiul rezoluţiei:

```
(PRINCIPU-REZ  '((P)(Q (NOT P)) ((NOT Q))) )
(S=((P) (Q (NOT P)) ((NOT Q))) )
(S1=((P) (Q (NOT P)) ((NOT Q))) )
(C1= (P) C2= ((NOT Q)) R=((NOT P)) )
(S=((P) (Q (NOT P)) ((NOT Q)) ((NOT P) (Q)) )
(S1=((NOT P)) (Q)) )
CL-VIDA
```

Soluţie.

```
(DEFUN START (); Principiul rezolutiei pentru calculul cu propozitii
(PROG (FORM); FORM=formula
INTRODU
(FORMAT T "Introduceti  formula in forma prefixata cu conectorii:
        AND,OR, ->, <->, <-, NOT=")
(SETF FORM (READ))
(IF (NOT (FBF? FORM)) (GO INTRODU))
(RETURN (PRINCIPIU-REZ (TRANSFORMA FORM)))
))
(DEFUN MULT-REZ (S S1 &AUX REZ R); S si S1 multimi de clauze
(PRINT '(S= ,S)) ; S sistemul initial de clauze
(PRINT '(S1= ,S1)); S1 sistemul cu rezolventii obtinuti
(DOLIST (I S REZ); iau pe rind toate clauzele din S
(DOLIST (J S1) ; si toate clauzele din S1 si calculez rezolventii
(COND ((NULL (SETF R (REZOLVENT I J))))
        ((EQ R 'CL-VIDA) (RETURN (SETF REZ R)))
        ((OR (TAUTOLOGIE? R)(MEMBER R S :TEST 'EQUAL)
                (MEMBER R S1 :TEST 'EQUAL)
                (MEMBER R REZ :TEST 'EQUAL)) )
        (T (PRINT '(C1= ,I C2= ,J R= ,R)) (SETF  REZ (CONS R REZ)))
))
 (IF (EQ REZ 'CL-VIDA) (RETURN REZ))
))
(DEFUN PRINCIPIU-REZ (S)   ; S = multimea  de clauze
(PROG ((S1 S))
```

```
DIN-NOU
(SETF S1 (MULT-REZ S S1))
(COND ((NULL S1)(RETURN 'CONSISTENT));nu sunt rezolventi noi
      ((EQ S1 'CL-VIDA)(RETURN S1)) ;a aparut rezolvent=NIL
)
(SETF S (APPEND S S1))
(GO DIN-NOU)
))
```

Nu există nici o metodă care să ne spună dacă o formulă este tautologie într-un timp polinomial (este o problemă deschisă).
Metoda principiului rezoluției a fost preluată în mod spectaculos pentru demonstrarea automată pentru calculul cu predicate.

Ce am învățat în acest capitol:

- ce este un conector logic
- cum arată o formulă în calculul cu propoziții
- cum verificăm dacă o formulă este bine formată
- cum aflăm tipul unei formule (dacă este tautologie, satisfiabilă, inconsistentă) în calculul cu propoziții prin ***metoda tabelului***
- cum se transformă o formulă în forma normal conjunctivă
- altă metodă de a afla tipul formulei (dacă este inconsistentă): ***principiul rezoluției***

Exerciții. 20. Logică

1. Scrieți în LISP pornind de la definiție, deci fără ajutorul altor conectori pe *implică* - > și <- și echivalent <->.
2. Scrieți de mînă tabelul de adevăr al tuturor conectorilor.
3. Generați cei 16 conectori. Căutați-le numele. Determinați toate familiile de conectori minimali (care generează ceilalți conectori).

4. Generaţi aleator formule în Cp, formulele fiind scrise numai cu AND, OR, NOT,-> şi <->. Generatorul de formule va furniza date pentru demonstrator.
5. Scrieţi un program care numără cîte formule sunt tautologii din 1000 de formule generate aleator în Cp. Proiect: ***Ordine în dezordine***
6. Testaţi mai multe tehnici de demonstrare (tabel, principiul rezoluţiei) pe 1000 formule din Cp generate aleator. Comentarii.
7. Verificaţi teoremele din ***Principia Mathematica*** de Bertrand Russell.

21. Automate celulare. Jocul vieţii

Jocul vieţii a fost propus de John Conway, matematician de la Cambridge. Prima oară a fost descris de Martin Gardner în 1970.
Este un exemplu de AUTOMAT CELULAR.
Imaginaţi-vă o matrice de celule unde fiecare celulă are 2 stări:
mort (0) sau viu (1).
Matricea trăieşte după un ceas discret care bate un tact.
Fiecare celulă la fiecare tact va trebui să decidă în funcţie de regulile jocului, ce va face: ***moare*** (0) sau ***trăieşte*** (1) ?
Iată regulile:
a. Dacă celula este vie la momentul **t**, va ramîne vie şi la momentul **t+1** numai dacă are 2 vecini vii şi nu mai mult de 3 vecini vii.
(interpretare: dacă nu este superpopulată sau nehrănită)
b. Dacă celula este moartă la timpul **t**, va ramîne moartă dacă nu va avea 3 vecini vii (interpretare: cel puţin 3 părinţi ca să se nască din nou)
Predicţia lui Conrad:
Populaţiile (mulţimea celulelor) sau mor sau ciclează.

Exemplu derulare a programului "JOCUL VIETII" a lui J. Conrad
:(START)
Cite generatii doriti => **3**
Dimensiunea matricii= > **4**
Nume-fisier de tiparire(pt ecran=NIL)= > **"life.dat"**
Tactul nr= 0
(0 0 1 0)
(1 1 0 0)
(1 0 1 0)
(0 0 0 1)

Matricea noua este:
(0 1 0 0)
(1 0 1 0)
(1 0 1 0)
(0 0 1 0)
Tactul nr= 1
(0 1 0 0)
(1 0 1 0)
(1 0 1 0)
(0 0 1 0)
Matricea noua este:
(0 1 0 0)
(0 0 1 0)
(0 0 1 1)
(0 0 0 0)
Tactul nr= 2
(0 1 0 0)
(0 0 1 0)
(0 0 1 1)
(0 0 0 0)
Matricea noua este:
(0 0 0 0)
(0 1 1 1)
(0 0 1 1)
(0 0 0 0)

Folosim funcţii LISP prezentate deja în carte: INTRODU şi FILL.
Reprezentarea noastră este o matrice A cu 0 (mort) sau 1 (viu).
Pentru simplificare considerăm matricea A (4×4) ca fiind dată.
Se poate să pornim şi de la o matrice A generată aleator. Puteţi scrie funcţia (GENEREAZA-RANDOM-MATRICE DIM).

```
;
(SETF DIM 4)
(SETF A '((0 0 1 0) (1 1 0 0)(1 0 1 0)(0 0 0 1)) )
;
(DEFUN INTRODUCERE-DATE()
```

```
(SETF CEAS (INTRODU "cite generatii doriti=" 'INTEGERP))
(SETF LOC-TIPARIRE
(INTRODU "Nume-fisier de tiparire (pt ecran=nil)"
            '(LAMBDA(Z) (OR (NULL Z)(STRINGP Z))) ))
(IF LOC-TIPARIRE (SETF OUT (OPENO LOC-TIPARIRE))
        (SETF OUT *STANDARD-OUTPUT*))
)
Reprezentăm matricile prin liste de liste!
(DEFUN A (I J); (A I J)  valoarea la linia I si coloana J
      (NTH J (NTH I A) ) ; atentie numaram de la 0
)
```

Funcţia principală o numim START. E bine să numiţi întotdeauna funcţia principală cu START. Nu aveţi idee de cîte ori credem că ţinem minte ceva şi de fapt uităm instantaneu. În LISP nu există un indicator special ca să-ţi evidenţieze funcţia ***principală*** sau funcţia care le apelează pe toate.

```
(DEFUN START (); matricea A este de 4 x 4 in ex. nostru
(INTRODUCERE-DATE)
(DOTIMES (K CEAS)
      (FORMAT OUT "~% TACTUL NR= ~A" K)
      (SCRIE-MATRICE A)
      (SETF NOU-A A)
 (DOTIMES (I DIM)
  (DOTIMES (J DIM)
    (LET ((VECINI (SUMA-VII I J)) )
    (COND ((= (A I J) 1)
(IF (MEMBER VECINI '(2 3)) T (PUT I J 0)) )
            ((= (A I J) 0) (IF (= VECINI 3) (PUT I J 1) ) )
    ))
 ))
(FORMAT OUT "~%MATRICEA NOUA ESTE:")
(SCRIE-MATRICE NOU-A)
(SETF A NOU-A)
))
```

```
Modifica pe NOU-A punind in locul lui I  si J  pe  VAL
(DEFUN PUT (I J VAL)
 (SETF NOU-A
   (FILL NOU-A (FILL (NTH I NOU-A) VAL (1+ J)) (1+ I))
 ))  ; fill numara de la 1

(DEFUN SCRIE-MATRICE (X)
   (DOLIST (I X)(FORMAT OUT "~%~T~A" I))
)
```

1,1	*1,2*	...	**1,n**
2,1	2,2	...	*2,n*
...	...	...	...
n,1	*n,2*	...	**n,n**

Fig 21.1. Automatul celular (matrice NxN)

```
(DEFUN SUMA-VII (I J); suma de 1 a vecinilor lui A(I J)
(LET ((N (1- DIM)))
(COND ((AND (> I 0)(> J 0) (< I N) (< J N)); celule cu 8 vecini
          (+ (A (1- I) (1- J)) (A I (1- J)) (A (1- I) J)
             (A (1+ I) (1+ J)) (A (1+ I) J) (A I (1+ J))
             (A (1+ I) (1- J)) (A (1- I)(1+ J))  ))
      ((AND (= I 0)(= J 0)); celula de colt stinga sus: 3 vecini
          (+ (A (1+ I) (1+ J)) (A I (1+ J)) (A (1+ I) J)  ))
      ((AND (= I 0)(= J N)); celula de colt dreapta sus: 3 vecini
           (+ (A I (1- J)) (A (1+ I) (1- J)) (A (1+ I) J) ))
      ((AND (= I N) (= J 0)); celula de colt stinga jos: 3 vecini
          (+ (A (1- I) J) (A I (1+ J)) (A (1- I)(1+ J))  ))
      ((AND (= I N) (= J N)); celula de colt dreapta jos: 3 vecini
          (+ (A (1- I) (1- J)) (A I (1- J)) (A (1- I) J)) )
      ((= I 0) ; celulele de pe linia de sus: 5 vecini
          (+ (A I (1- J)) (A (1+ I) (1+ J)) (A I (1+ J))
               (A (1+ I) J) (A (1+ I) (1- J)) ) )
      ((= I N); linia de jos: 5 vecini
```

```
        (+ (A (1- I) (1- J)) (A I (1- J)) (A (1- I) J)
           (A I (1+ J)) (A I (1+ J)) (A (1- I)(1+ J)) ))
    ((= J 0); celulele de pe prima coloana:5 vecini
        (+ (A (1- I) J)(A (1+ I) (1+ J)) (A I (1+ J))
           (A I (1+ J)) (A (1- I)(1+ J)) ))
    ((= J N); celulele de pe ultima coloana: 5 vecini
        (+ (A (1- I) (1- J)) (A I (1- J)) (A (1- I) J)
           (A (1+ I)  J) (A (1+ I) (1- J))  ))
)))
```

Exerciţii. 21. Jocul vieţii

1. Scrieţi mai bine, mai compact funcţia SUMA-VII. Imaginîndu-vă o convenţie în care matricea noastră se află cufundată într-una mai mare, deci adăugîndu-i o bordură, astfel ca fiecare celulă să aibă de fapt 8 vecini.
2. Tipul de dată ***matrice*** se află implementat în unele sisteme LISP prin :AREF. Încercaţi să rescrieţi programul cu funcţiile de lucru pentru matrici.
3. Faceţi teste pe Jocul Vieţii.

22. Algoritmi genetici (AG)

Conceptul de **evoluţie** a fost propus de savantul englez Charles Darwin în 1859 în celebra sa carte *Originea speciilor prin selecţie naturală.* Principala sa teză este:

Speciile evoluează prin *variaţii aleatoare* urmate de selecţie naturală în care *cel mai potrivit supravietuieşte.*

Algoritimi genetici (pe scurt **AG**) în informatică au fost inventaţi de John Holland în 1960. S-au impus abia în 1975 prin cartea sa *Adaptation in Natural and Artificial Systems,* unde el prezintă algoritmii genetici ca o abstractizare a noţiunii de **evoluţie** darwiniană.
Evoluţia este o metodă masivă de căutare în paralel: în loc de a rezolva o specie o dată, evoluţia testează şi schimbă milioane de specii în paralel.
Conceptul de evoluţie inspiră tehnicile de rezolvare a problemelor mai ales pentru:

- probleme care presupun căutarea în spaţii imense ale soluţiilor
- probleme care necesită o *adaptare* a programelor într-un context care se schimbă.

Pînă acum în inteligenţa artificială a predominat ideea că a găsi reguli şi ale aplica este suficient pentru a scrie probleme "inteligente". Realitatea ne arată însă, că regulile necesare unui sistem expert sunt mult prea complexe pentru a fi abordate într-o manieră de sus în jos (top-down).

Din biologie ştim:

- Organismele vii sunt formate din celule.
- Fiecare celulă conţine aceeaşi mulţime de cromozomi - şiruri de DNA - care servesc de şablon pentru întregul organism.
- Un cromozom este divizat în gene. Putem vedea o genă ca fiind o codificare a unei trăsături. De exemplu: culoarea ochilor.
- Posibilele valori ale unei gene se numesc **alele.** De exemplu: albastru, negru, verde pentru culoarea ochilor.
- Fiecare genă este localizată într-o poziţie particulară a cromozomului.
- Mulţimea cromozomilor unui organism se numesc **genom.**

Vocabularul unui algoritm genetic:

In cele ce urmează termenul de **cromozom** corespunde cu **individ**.
Mulţimea indivizilor (a cromozomilor) la un moment dat se numeşte **populaţie.**
De obicei indivizii unei populaţii din AG sunt reprezentaţi ca şiruri de biţi.
Genele sunt biţi (sau blocuri de biţi) dintr-un şir (individ-cromozom).
Fiecare loc dintr-un şir are două valori posibile (*alele*): 0 sau 1 (în cazul în care alfabetul este binar).
Fiecare individ este considerat ca un punct în spaţiul stărilor de candidaţi posibili.
Funcţia de fitness măsoară fiecărui individ **potriveala** în populaţia curentă.
Fitnessul depinde de cum răspunde individul la problema curentă şi stabileşte o distanţă între candidaţi în spaţiul soluţiilor.

Operaţiile din AG

Deci pornim de la o populaţie de indivizi asupra cărora facem următoarele operaţii:

Cross-over pentru a produce un nou individ. Constă din schimbarea materialului genetic dintre 2 părinţi (indivizi).
Operatorul CROSS-OVER ia doi indivizi (cromozomi) şi îi rupe în acelaşi punct şi schimbă între ei restul şirului.

De exemplu: AAABBB şi XYXYXYX se rupe la locul 3 (obţinut aleator) şi obţinem 2 cromozomi noi: AAXYXYX şi XYABBB.

Mutaţia unui individ. Constă în schimbarea valorii unui bit dintr-un şir. Operatorul MUTATIE ia in mod aleator un bit dintr-un individ (şir) şi-l schimbă cu altă valoare aleasă în mod aleator din alfabetul peste care sunt formate şirurile de indivizi. De exemplu: (1 0 0 0) poate fi schimbat în poziţia 2 (obţinut aleator) cu 1. Obţinem deci (1 1 0 0).
De obicei operaţia de mutaţie are o probabilitate mai mică.
Selecţia după o funcţie de fitness alege un individ din populaţie pentru a fi propus pentru reproducere.
O metodă posibilă de selecţie în AG este *selecţia proporţională cu fitness*-ul: în care probabilitatea ca un individ să fie selectat pentru reproducere este egală cu fitness-ul lui împărţit la media fitness-ului populaţiei.
O metodă simplă de a implementa selecţia fitness-proporţională este **metoda ruletei** (Goldberg, 1989) unde se atribuie fiecărui individ o bucată dintr-o ruletă circulară egală cu fitness-ul lui. Ruleta este învirtită, iar bila rămîne într-un loc care corespunde cu categoria de fitness din care este individul selectat.

Un algoritm genetic simplu (vezi /28/)
Avem o problemă care se pretează la o abordare prin AG. Spaţiul soluţiilor problemei sunt şiruri. Avem următorii paşi:
1. Incepem cu o mulţime aleator generată de indivizi (şiruri) numită **populaţie.**
2. Calculăm fitness-ul fiecărui individ din populaţie.
3. Repetăm următorii paşi pînă cînd avem N urmaşi:
   - Selectează o pereche de indivizi din populaţia curentă. Selecţia se face aleator, dar crescător odată cu funcţia de fitness. Selecţia se poate face sau nu *cu punere la loc* (un individ poate fi ales să fie părinte de mai multe ori).
   - Cu probabilitatea PC (probabilitatea cross-over) se sparge perechea de indivizi într-un punct ales aleator şi se formează doi indivizi noi. Dacă nu are loc nimic nou se copiază părinţii.

- Cu probabilitatea PM (probabilitate mutaţie) se face mutaţia pe cei 2 indivizi şi se pun indivizii rezultaţi într-o nouă populaţie.

4. Schimbă toată populaţia curentă cu noua populaţie.
5. Du-te la 2.

Fiecare iteraţie este numită **o nouă generaţie**. Un algoritm tipic de AG are între 50 şi 5000 generaţii.

Mulţimea tuturor generaţiilor se numeşte **o rulare.**

La sfirşitul unei rulări în mod normal apar indivizii mult mai **potriviţi** în populaţie.

In exemplul nostru avem următoarele variabile:
Indivizii sunt şiruri de lungime LUNGIME-SIR = 4 peste alfabetul ALFABET= {1, 0}.
Populaţia iniţială de şiruri este formată din NR-POPULATIE= 4 indivizi (şiruri) generate aleator. De obicei lungimile şirurilor sunt cu mult mai mari, iar numărul populaţiei iniţiale este între 50-500.

Funcţia de fitness este dată de numărul de 1. Cu cît sunt mai mulţi de 1 cu atît individul (şirul) este mai bun. Deci: FITNESS (individ) = numărul aparţiilor lui 1 din şirul care reprezintă individul.

Probabilitatea de cross-over este citită de la consolă PC = 0.9

Probabilitatea de mutaţie citită de la consolă PM = 0.3

In funcţia ALEG-INDIVID am facut în aşa fel încît cu cît un individ este mai bun cu atît şansele lui să fie ales sunt mai mari. Am simulat metoda ruletei lui Goldberg.
Odată o pereche selectată cu probabilitatea PC facem cross-over pentru a forma un nou urmaş sau dacă nu are loc, se fac copii egale cu părinţii. Apoi fiecare urmaş se supune la o mutaţie cu o probabilitate PB.

Iată rezultatul unei rulări pe ecran al programului pe care îl prezentăm după aceea:

```
:(START)
Alfabetul= > 4
Lungime-sir= > 4
Probabilitate Cross-OVER = > 0.9
Probabilitate Mutatie = > 0.3
Marimea populatiei = > 4
Numar de generatii = > 5
Nume-fisier de tiparire (pt ecran= NIL) => "genetic.dat"

GENERATIE= 0
populatie=((1 1 1 1) (1 1 0 0) (0 0 0 1) (0 0 1 1))
(1 1 1 1)  Fitness= 4
(1 1 0 0)  Fitness= 2
(0 0 0 1)  Fitness= 1
(0 0 1 1)  Fitness= 2
Medie fitness= 2.250000

Aleg indivizi: x= (0 0 1 1) Fitness= 2 si y=(0 0 1 1) Fitness=2
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 0 0) Fitness= 2 si y=(1 1 1 1) Fitness=4
Fac cross-over:
Adaug individ nou= (1 1 1 1)
Adaug individ nou= (1 1 0 0)
Aleg indivizi: x= (1 1 0 0) Fitness= 2 si y=(0 0 1 1) Fitness=2
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(0 0 0 1) Fitness=1
Aleg indivizi: x= (0 0 1 1) Fitness= 2 si y=(1 1 1 1) Fitness=4
Fac cross-over:
Adaug individ nou= (0 0 1 1)
Adaug individ nou= (1 1 1 1)
```

GENERATIE= 1
populatie=((1 1 1 1) (0 0 1 1) (1 1 0 0) (1 1 1 1))
(1 1 1 1) Fitness= 4
(0 0 1 1) Fitness= 2
(1 1 0 0) Fitness= 2
(1 1 1 1) Fitness= 4
Medie fitness= 3.000000

Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Fac cross-over: Fac mutatie:
Adaug individ nou= (1 1 1 1)
Adaug individ nou= (1 1 1 1)
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(0 0 1 1) Fitness=2
Fac cross-over:
Adaug individ nou= (1 1 1 1)
Adaug individ nou= (0 0 1 1)
GENERATIE= 2
populatie=((0 0 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1))
(0 0 1 1) Fitness= 2
(1 1 1 1) Fitness= 4
(1 1 1 1) Fitness= 4
(1 1 1 1) Fitness= 4
Medie fitness= 3.500000
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Fac cross-over: Fac mutatie:
Adaug individ nou= (1 1 1 1)
Adaug individ nou= (0 1 1 1)
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(0 0 1 1) Fitness=2

Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (0 0 1 1) Fitness= 2 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (0 0 1 1) Fitness= 2 si y=(0 0 1 1) Fitness=2
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Fac cross-over:
Adaug individ nou= (1 1 1 1)
Adaug individ nou= (1 1 1 1)
GENERATIE= 3
populatie=((1 1 1 1) (1 1 1 1) (0 1 1 1) (1 1 1 1))
(1 1 1 1) Fitness= 4
(1 1 1 1) Fitness= 4
(0 1 1 1) Fitness= 3
(1 1 1 1) Fitness= 4
Medie fitness= 3.750000

Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Fac cross-over: Fac mutatie:
Adaug individ nou= (0 1 1 1)
Adaug individ nou= (1 0 1 1)
Aleg indivizi: x= (1 1 1 1) Fitness= 4 si y=(1 1 1 1) Fitness=4
Fac cross-over:
Adaug individ nou= (1 1 1 1)
Adaug individ nou= (1 1 1 1)
Populatia finala=((1 1 1 1) (1 1 1 1) (1 1 1 1) (1 1 1 1))
Avind media fitness=4.000000

Prezentăm în continuare programul care a scos listarea de mai sus.

La început populaţia este o listă vidă.
(SETF POPULATIE NIL NOU-POPULATIE NIL)
Exemplu. (SETF ALFABET '(0 1))

```
(SETF POPULATIE
      '((0 0 0 0 1 0)(0 0 1 1 0 1)(0 1 0 1 0 1)(0 1 1 1 0 0 0 1)))
```

Populaţia e formată din indivizi

Individ = un şir (lista în reprezentarea noastră) peste un alfabet.

Deci (0 0 0 0 1 0) este un individ în interpretarea noastră!

```
(DEFUN INTRODUCERE-DATE ()
(SETF ALFABET (INTRODU "Alfabetul= " 'LISTP))
(SETF LUNG-SIR (INTRODU "Lungime-sir= " 'NUMBERP))
(SETF PC (INTRODU "Probabilitate CROSS-OVER= " 'FLOATP))
(SETF PM (INTRODU "Probabilitate MUTATIE= " 'FLOATP))
(SETF NR-POPULATIE
        (INTRODU "Marimea POPULATIEI= " 'INTEGERP))
(SETF NR-GENERATII
        (INTRODU "Numar de GENERATII= " 'INTEGERP))
(SETF LOC-TIPARIRE
(INTRODU "Nume de fisier de TIPARIRE (pt ecran:nil)="
                  '(LAMBDA(Z)(OR (NULL Z)(STRINGP Z))) ))
(IF LOC-TIPARIRE (SETF OUT (OPENO LOC-TIPARIRE))
          (SETF OUT *STANDARD-OUTPUT*))
)
```

Funcţia principală este (START). Am adoptat acelaşi principiu de a denumi întotdeauna funcţia principală la fel ca să ţinem minte.

```
(DEFUN START ()
(PROG (L X Y ); X  si Y indivizi, iar L= (X Y)
(INTRODUCERE-DATE)
(SETF POPULATIE (MAKE-POPULATIE NR-POPULATIE))
(DOTIMES (K NR-GENERATII)
 (FORMAT OUT "~%GENERATIE= ~A ~%" K)
 (IF NOU-POPULATIE (SETF POPULATIE NOU-POPULATIE))
 (TIPARESTE-DATE)
 (SETF NOU-POPULATIE NIL)
```

```
 (DO () ((>= (LENGTH NOU-POPULATIE)(LENGTH POPULATIE)) T)
      ; aleg o pereche de indivizi X si Y
    (SETF X (ALEG-INDIVID) Y (ALEG-INDIVID))
    (FORMAT OUT
    "~% Aleg indiviz: X= ~A Fitness= ~A si  Y=~A Fitness=~A~T"
       X  (FITNESS X) Y  (FITNESS  Y) )
     (WHEN (FAC-CROSS-OVER? PC)
           (FORMAT OUT "~% Fac cross-over: ")
           (SETF L (CROSS-OVER X Y (RANDOM LUNG-SIR)) )
           (SETF X (CAR L) Y (CADR L))
           (IF (FAC-MUTATIE? PM)
             (PROGN (FORMAT OUT " Fac mutatie: " )
               (ADAUG (MUTATIE X))(ADAUG (MUTATIE Y)))
           (PROGN (ADAUG X)(ADAUG Y)) )
    )
) )
(FORMAT OUT "~%POPULATIA FINALA=~A ~% AVIND MEDIA
FITNESS=~A~%"  POPULATIE (MEDIE-FITNESS))
(IF OUT (CLOSE OUT))
))
```

Exemplu. (MAKE-SIR 6) → (1 0 0 1 1 1)

```
(DEFUN MAKE-SIR (LUNG-SIR &AUX R)
(DOTIMES (I LUNG-SIR R)
      (SETF R (CONS (ALEGE ALFABET) R))
))
```

Exemplu. (ALEGE '(0 0 1 1 0 0 1 1)) → 1 *-PRESUPUNEM-*

```
(DEFUN ALEGE (LISTA); alege la intimplare din LISTA
(NTH (RANDOM (LENGTH LISTA)) LISTA))
(DEFUN MAKE-POPULATIE (K &AUX POPULATIE)
(DOTIMES (I K POPULATIE)
(SETF POPULATIE (CONS (MAKE-SIR LUNG-SIR) POPULATIE))
))
```

Exemplu. (MUTATIE '(0 1 0 0 1 1 1 1)) → (0 1 1 0 1 1 1 1) *-RANDOM-*

```
(DEFUN MUTATIE (SIR &AUX LOC)
(LET ((LOC (RANDOM (LENGTH SIR))) (NOU (ALEGE ALFABET)) )
;scrie in LOC din SIR pe NOU peste cel vechi
        (FILL SIR NOU LOC ) ; FILL e scris de noi
))
```

Exemplu. (FITNESS '(0 1 1 0 0 1 1 1)) → 5

```
(DEFUN FITNESS (INDIVID) ; individ din populatie, individ=lista
        (NR-DE-1 INDIVID)) ; citi de 1 sunt in lista
```

Exemplu. (NR-DE-1 '(0 0 0 1 1 1 1 0)) → 4

```
(DEFUN NR-DE-1 (INDIVID)
  (COUNT 1 INDIVID)); COUNT numara aparitiile lui 1 in lista INDIVID
```

Exemplu. (CROSS-OVER '(A B C D) '(1 2 3 4) 2) → ((A B 3 4) (1 2 C D))

(CROSS-OVER '(0 0 0 1 1 1 0 0) '(1 0 1 0 1 0 1 0) 2) →
((0 0 0 0 1 0 1 0) (1 0 1 1 1 1 0 0))

Soluție.

```
(DEFUN CROSS-OVER (X Y LOC); X si Y indivizi din populatie
; LOC= locul unde se rupe
(LET ((X1 (SUBSEQ$ X 0 LOC)) (X2 (SUBSEQ$ X LOC))
        (Y1 (SUBSEQ$ Y 0 LOC)) (Y2 (SUBSEQ$ Y LOC)))
(LIST (APPEND X1 Y2)(APPEND Y1 X2))
))

(DEFUN ADAUG (INDIVID); adaug individ la nou-populatie (var globala)
        (FORMAT OUT "~% ADAUG INDIVID NOU= ~A " INDIVID)
        (SETF NOU-POPULATIE (CONS INDIVID NOU-POPULATIE)) )

(DEFUN FAC-MUTATIE? (PM) ; PM=probabilitatea de a face mutatie
        (FAC-CU-PROB? PM))
(DEFUN FAC-CROSS-OVER? (PC) ;PC=probabilitatea de a face cross-over
        (FAC-CU-PROB? PC))
```

Exemplu. (FAC-CU-PROB? 0.98) → T *-PRESUPUNEM-*

```
(DEFUN FAC-CU-PROB? (P) ; P=probabilitatea de a face mutatie
        (<= (RANDOM 1000) (* P 1000))
)
```

Simularea ruletei (metoda Goldberg)

```
(DEFUN ALEG-INDIVID (); depinde de distributia de fitness
(LET* ((SUMA (SUMA-FITNESS)) (ZAR (RANDOM SUMA)) (LOC 0))
; fiecare individ are un interval alocat
(DOLIST (I POPULATIE)
        (SETF LOC (+ (FITNESS I) LOC ))
        (IF (> LOC ZAR)(RETURN I))
)))
(DEFUN TIPARESTE-DATE ()
(FORMAT OUT "POPULATIE=~A ~%" POPULATIE)
(MAPC '(LAMBDA (Z)
          (FORMAT OUT "~A  FITNESS= ~A ~%" Z (FITNESS Z)))
         POPULATIE)
(FORMAT OUT "MEDIE FITNESS= ~A ~%" (MEDIE-FITNESS))
)
(DEFUN SUMA-FITNESS () ;POP= populatie sau nou-populatie
(APPLY '+ (MAPCAR '(LAMBDA(Z)(FITNESS Z)) POPULATIE ))
)
(DEFUN MEDIE-FITNESS ()
(/ (FLOAT (SUMA-FITNESS)) (LENGTH POPULATIE) ))
```

Aplicaţii posibile de AG

Toate metodele de căutare din AG au ceva în comun:

- pornesc de la nişte candidaţi posibili
- evaluează rezulatele după un criteriu de optimizare (funcţia de fitness)
- decid pe baza evaluării candidaţilor care să fie reţinute şi care nu (selecţia)

Evidenţiem mai multe **tipuri de căutări:**

- Căutare pentru regăsirea datelor: care este strategia cea mai bună pentru a căuta într-o bază un nume dat? In acest caz informaţia există. Ea nu se generează.

- Căutare de drumuri pentru a ajunge la un scop: a găsi eficient o mulțime de acțiuni care te duc de la o stare dată la un scop. Vezi multe probleme de tip Prolog (reginele pe tabla de șah, etc). Algoritmi tipic de inteligență artificială de căutare într-un arbore. Soluțiile posibile sunt generate pe rînd.
- Căutare de soluții: căutarea unei soluții într-un spațiu mult prea mare de indivizi posibili. In acest gen de probleme nu putem să generăm toate soluțiile și apoi să le verificăm. Nu se garantează găsirea soluției optime absolute.

Iată pe scurt cîteva domenii unde apar în rezolvarea problemelor căutări de acest tip:

- Optimizare: circuite și job scheduling
- Programare automată
- Reguli pentru sisteme de învățare a clasificării pentru roboți
- Economie. Piețe economice
- Sistemul imunitar natural
- Ecologie
- Genetică
- Evoluție și învățare
- Aspecte evoluționare ale sistemelor sociale și în sistemele multi-agent.

Pentru detalii sugerăm cartea lui Margaret Mitchell, /28/ și pentru o excelentă prezentare generală a problemelor evoluționiste moderne pe Richard Dawkins, /10/.

Exerciții. 22. Algoritmi genetici

1. Rescrieți algoritmul de mai sus reprezentînd indivizii ca string-uri.
2. Faceți experimente cu programul de mai sus.
3. Imaginați alte funcții de FITNESS.

Bibliografie

1. Atanasiu Adrian, ***Lingvistica computaţională***, Ed. Universităţii Bucureşti,1998.
2. Abelson, Harold, Gerald Sussman, J.Sussman. ***Structure and Interpretation of Computer Programs***, MIT Press, McGraw-Hill Book Company,1985.
3. Allen James, ***Natural Language Understanding***, The Benjamin/ Cummings Pub., 1987.
4. Allen John. ***The Anatomy of LISP***, McGraw Hill Inc, 1978.
5. Andonie R., Gârbacea I., ***Algoritmi Fundamentali, O perspectivă în C++***, Editura Libris, Cluj, 1995.
6. Barendregt H.P. ***The Type-Free Lambda Calculus***, în Barwise (ed.), Handbook of Mathematical Logic, North Holland, 1977.
7. Boyer R.S. , J.S. Moore, ***Proving Theories about LISP Functions***, pp. 486-493, Proc. 3rd IJCAI, Stanford, Ca.1973.
8. Calude C., V. E. Căzănescu, ***Bazele Informaticii: Lecţii de logică matematică***, Tipografia Universităţii Bucureşti,1984.
9. Chang C.L., Lee R.C.T., ***Symbolic Logic and Mechanical Theorem Proving***, Academic Press, New York, 1973.
10. Richard Dawkins, ***Un rîu pornit din Eden***, Humanitas, 1995.
11. Florea Adina, D. Tecuci, B. Panghe, ***Programe LISP pentru inteligenţa Artificială***, Editura Sfera, Bucureşti, 1998.
12. Friedman D.P., ***Scheme and the Art of Programming***, The MIT Press, 1990.
13. Friedman, D. P., Matthias Felleisen,. ***The Little LISPer***, Chicago, IL:SRA, Cambridge, MA: MIT Press, 1988.

14. Gallier Jean H. *Logic for Computer Science*. Foundations of Authomatic Theorem Proving, Harper & Row, Publ, N.Y. 1986.
15. Gazdar Gerald, *Natural Language Processing in LISP*, Addison-Wesley, 1989.
16. Giumale C., *Programare Funcţională*, Editura Tehnică, 1997.
17. Harvey, Brian, *Simply SCHEME : Introducing Computer Science*, MIT Press, 1994.
18. Hasemer T., Dominique J., *Common LISP Programming for AI*, Int. Comp. Science Series, Addison-Wesley , 1989.
19. Keene Sonya E., *Object-Oriented Programming in Common LISP*, Addison Wesley, 1989.
20. Kleene S.C. *Mathematical Logic*, John Wiley, New York, 1967.
21. Knuth D. E. *Tratat de programarea calculatoarelor, Sortare şi Căutare*, Editura Tehnică, Bucureşti, 1974.
22. Lawless, Molly M. Miller, *Understanding CLOS The Common Lisp Object System*, Digital Press, 1991.
23. McCarthy, John. *Recursive Functions of Symbolic Expressions and their Computation by Machine*. C. ACM, 3(4):184-195, 1960.
24. Maliţa Mihaela, Maliţa Mircea, *Bazele inteligenţei artificiale. Logici propoziţionale*, Editura Tehnică, 1987.
25. Marcus Solomon, *Language, logic, cognition and communication. A semiotic computational and historical approach*, Universitat Roviza i Virgilii, 1995.
26. Miller Mally M., Benson Eric, *LISP - Style and Design*, Digital Equipment Corporation, Cambridge, MA 1990.
27. Milner Wendy L., *COMMON LISP-A Tutorial*, Prentice Hall, '88.
28. Mitchell Melanie, *An Introduction to Genetic Algorithms*, A Bradford Book, The MIT Press, 1996.
29. Negoiţă C., *Mulţimi vagi si aplicaţiile lor*, Editura Tehnică, 1974 .
30. Norvig Peter, *Artificial Intelligence Programming: Case Studies in Common LISP*, Morgan Kaufmann Pub., 1992.
31. Popovici C., H. Georgescu, Luminiţa State, S.Rudeanu, *Bazele Informaticii*, Tipografia Universităţii Bucureşti, 1990-91 (2 vol).
32. Rudeanu, Sergiu, *Lecţii de calculul predicatelor şi calculul propoziţiilor*, Editura Universităţii Bucureşti, 1997.

33. Stark Richard W., *Lisp, Lore, and Logic, An Algebraic View of Lisp Programming, Foundations, and Applications*, Springer-Verlag, 1990.
34. Steele G.L. *CommonLISP. The Language*, Digital Press, 1990.
35. Streinu Ileana, *LISP. Limbajul de programare al inteligenţei artificiale*, Editura Ştiinţifică Enciclopedică, 1986.
36. Tomescu I., *Data Structures*, Editura Universităţii din Bucureşti, 1997.
37. Turner Raymond, *Constructive Foundations for Functional languages*, McGraw-Hill Book Comp, 1991.
38. Wertz Harald, *Common LISP une introduction à la programmation*, Masson, 1991.
39. Wilensky R., *Common LISPcraft*, Norton&Co., New York, 1984.
40. Winston Patrick H., Horn B.K.P., *LISP*, Addison Wesley, 1993.
41. Yuasa Taiichi, *Common Lisp Drill*. Academic Press, 1988.
42. Yuasa Taiichi, Hagiya Masami, *Introduction to Common Lisp*, Academic Press, 1987.

ANEXA. Funcţiile din carte

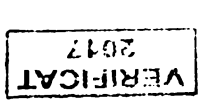
Funcţiile definte de autor sunt scrise cu **BOLD.**
Funcţiile din CLISP, rescrise de autoare sunt cu caractere *ITALICE.*
Funcţiile din CLISP sunt cu caractere obişnuite.

- 48
88
$< 179
* 48
***VECTORI 85**
/ 48
/= 40
+ 47
+VECTORI 85
<,<= 33
= 33, 38
>,>= 33
1-,1+ 41
ABS 39
ACK 57
ACOS 49
ADAUG 199
ADINCIME 103
ADUN-CU-1 17, 18
ALEG-INDIVID 200
ALEGE 198
ALLF 109
AND 62
APARE? 103
APPEND 63, 75
APPLY 42
ARANJAMENTE 110
ARCE 148
ASCII 96
ASIN 49
ASSOC 65
ATAN 49
ATOM 36
BACKQUOTE ` 61
BLOCK 84
BREADTH 153
C...R 37
CALCUL-FBF 172
CAUTA-CUVINT 125
CITESTE 94
CDR 37
CHAR 96
CLAUZE 178
CLOSE 117
COMBINARI 109
COMBINATII 110
COMPAR-FISIERE 124
CONCATENATE 96
COND 39
CONS 38

COPIAZA 118
COS 49
COUNT 103
CROSS-OVER 199
CUB 41
DEFMACRO 75
DEFUN 41, 28
DEPTH 153
DERIV 135
DO, DO* 81
DOLIST 79
DOS 128
DOTIMES 80
DRUM1 50
DUAL 176
EGAL MULTIPLE 108
EQ, EQUAL 38
ELIMINA-TAUTOLOGII 179
EVAL 35
EVENP 49
EXIT 16
EXP, E XPT 48
EXPLODE 97
FAC-CROSS-OVER? 200
FAC-CU-PROB? 200
FAC-MUTATIE? 200
FACT 54, 81 83
FBF? 171
FIB 55
FILL 101
FITNESS 199
FNC 177
FORMAT 120
FUNCALL 74
FUNCTION 88
FUNCTIONP 72
FUZZY-0 140
FUZZY-APARTINE 139
FUZZY-C 141
FUZZY-INCLUS 140
FUZZY-INTERSECTION 140
FUZZY-UNION 141
GCD 50
GENCAR 87
GET 92
GET-DECODED-TIME 50
GO 83
ID 149
IESIRE 148
IF 75
IMPART 162
IMPLODE 97
INFIX? 169
INORDINE 155
INSER 87,160
INSORT 159
INTEGERP 33
INTERCL 160
INTERN 97
INTERSECTION 92, 105
INTRODU 126
INTRODUCERE-DATE 187, 197
LABELS 71
LAST 53
LENGTH 64, 71, 73, 82, 83, 85
LET, LET* 70, 71
LIST 62, 74
LISTA-OPȚIUNI 127
LISTA-STRING 97
LISTATOM 102
LISTEN 125
LOAD 116
LOG 49
LOOP 75
L-VARIABILE 171
MAKE-LIST 81
MAKE-POPULATIE 198
MAKE-SIR 198
MAPC 86
MAPCAN 87

MAPCAR 84
MAPCARN 86
MATCH 111
MAX 40
MEDIE-FITNESS 200
MEMBER 63
MEMO-ORDINE 94
MENU 126
MIN 40, 47
MINA 93
MINUSP 33
MULTIME-FUZZY? 139
MULTIME-PARTI 108
MULT-REZ 182
MUTATIE 199
MUT-NOT 176
NCONC 77
NEXT 64
NODURI 147
NORM 142
NOT 61
NR-ATOMI 102
NR-CUVINT 122
NTH 64
NULL 61
NUMBERP 32
NUMEROTEZ 82, 85
OD 149
ODDP 49
OPEN 116
OPENI,OPENO 117
OPUS 179
OR 62
PACKAGEP 117
PAIRLIS 85
PERMUT 110
PLUSP 33
POSTORDINE 157
POZITII 104
PPRINT 119
PREFIX 170
PREORDINE 157
PRIN1 119
PRINC 118
PRINCIPIU-REZ 181
PRINT 119
PRODUS-CART 107
PROG, PROG* 83
PROG2,PROGN 84
PROPRIETATI-GRAF 148
PUNE-LA-FEL 94
PUN-LA-LOC 94
PUNE-PROP 146
PUN-OUT 147
PUN-PARANTEZE 85
PUT 188
PUTPROP 92
QSORT 161
QUOTE 34
RANDOM 50
RASSOC 65
READ 121
READ-CHAR 122
READ-CHAR-NO-HANG 122
READ-LINE 122
REM 49
REMOVE 67
REMOVE-DUPLICATES 105
REMPROP 92
RETURN 84
RETURN-FROM 84
REVERSE 67, 71
REVERSE-TOT 68, 86
REZOLVENT 180
RPLACA, RPLACD 77
SCOT-IMPLICATII 175
SCOT-OP 178
SCOT-PARANTEZE 87
SCRIE 120
SCRIE-MATRICE 188

SEARCH 98
SELECT 87, 112, 132
SET 111
SET-DIFFERENCE 106
SETF 33
SIN 49
SOME 80
SORT 164
SORT-ARB 164
SORT-INTERCL 160
SQRT 48
START 185, 194
STREAMP 117
STRING 95
STRING/= 98
STRING<= 98
STRING>= 98
STRING< 98
STRING> 98
STRING-DOWNCASE 97
STRING-EQUAL 98
STRING-GREATERP 98
STRING-LESSP 98
STRING-NOT-EQUAL 98
STRING-NOT-GREATERP 98
STRING-NOT-LESSP 98
STRINGP 95
STRING-UPCASE 97
SUBLIS 66, 82
SUBSEQ 99, 101
SUBSETP 107
SUBST 66
SUMA-CUBURI 41
SUMA-FITNESS 200
SUMA-PRIMELOR-NUMERE 81
SUMA-VII 188
SYMBOL-PLIST 92
TABEL 58
TABEL-ADEV 173
TABEL-LOG 58
TABLA 131
TAKE 67
TAN 49
TAUTOLOGIE? 179
TD 149
TERPRI 119
TEST 133
TIPARESTE-DATE 200
TIPUL 173
TRANSFORMA 174
TRUNCATE 49
TYPE-OF 115
UNION 106
UNLESS 76
WHEN 76
YES-OR-NO-P 73
ZEROP 33



DATA RESTITUIRII		
11 MAR 2004		
17 IUN 2011		
→		
2015		
2016		
2018		

ISBN: 973 - 575 - 204 - 2

Lei 20000