

## CURS 7

### Programare funcțională. Introducere în limbajul LISP

#### Cuprins

Bibliografie .....	1
1. Programare funcțională .....	1
1.1 Evaluare întârziată ( <i>lazy evaluation</i> ) .....	2
2. Limbajul Lisp .....	3
3. Elemente de bază ale limbajului Lisp .....	4
4. Structuri dinamice de date .....	5
4.1 Exemple .....	5
5. Reguli sintactice .....	6
6. Reguli de evaluare .....	7
6.1 Exemple .....	8
7. Funcții Lisp .....	8
(CONS e <sub>1</sub> e <sub>2</sub> ): l sau pp .....	8
(CAR l sau pp): e .....	9
(CDR l sau pp): e .....	9
7.1 Exemple .....	9

#### Bibliografie

**Capitolul 2.** Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligență Artificială*, Ed. Albastră, Cluj-Napoca, 2012

#### Programare logică

1. “*Learning Prolog provides you with a different perspective on programming that, once understood, can be applied to other languages as well.*”
2. “*You should learn Prolog as a part of your personal quest to become a better programmer.*”
3. Constraint programming
4. Inductive logic programming

#### 1. Programare funcțională

- Programare orientată spre valoare, programare aplicativă.
- Se focalizează pe valori ale datelor descrise prin expresii (construite prin definiții de funcții și aplicări de funcții), cu evaluare automată a expresiilor.
- Apare ca o nouă paradigmă de programare.

- Programarea funcțională renunță la instrucțiunea de atribuire prezentă ca element de bază în cadrul limbajelor imperative; mai corect spus această atribuire este prezentă doar la un nivel mai scăzut de abstractizare (analog comparației între **goto** și structurile de control structurate **while**, **repeat** și **for**).
- Prințipiile de lucru ale programării funcționale se potrivesc cu cerințele programării paralele: absența atribuirii, independenta rezultatelor finale de ordinea de evaluare și abilitatea de a opera la nivelul unor întregi structuri.
- **Inteligenta artificială** a stat la baza promovării programării funcționale. Obiectul acestui domeniu constă din studiul modului în care se pot realiza cu ajutorul calculatorului comportări care în mod obișnuit sunt calificate ca inteligente. Inteligența artificială, prin problematica aplicațiilor specifice (simularea unor procese cognitive umane, traducerea automată, regăsirea informațiilor) necesită, în primul rând, prelucrări simbolice și mai puțin calcule numerice.
- Caracteristica limbajelor de prelucrare simbolică a datelor constă în posibilitatea manipulării unor structuri de date oricât de complexe, structuri ce se construiesc dinamic (în cursul execuției programului). Informațiile prelucrate sunt de obicei siruri de caractere, liste sau arbori binari.
- **Pur funcțional** = absența totală a facilităților procedurale - controlul memorării, atribuirii, structuri nerecursive de ciclare de tip FOR
- **CE , NU CUM.**
- Limbaje funcționale – LISP (1958), Hope, ML, Scheme, Miranda, Haskell, Erlang (1995)
  - **Haskell**
    - pur funcțional, concurrent
    - aplicații industriale, aplicații web
  - **Erlang** – programare funcțională concurrentă
    - aplicații industriale: **telecomunicații**
- LISP nu e PUR funcțional
- Programare funcțională în limbaje precum Python, Scala, F#

### 1.1 Evaluare întârziată (*lazy evaluation*)

- Evaluarea întârziată** (*lazy evaluation*) este una dintre caracteristicile limbajelor funcționale.
- se întârzie evaluarea unei expresii până când valoarea ei e necesară (*call by need*)
    - în limbajul Haskell se evaluatează expresiile doar când se știe că e necesară valoarea acestora
  - în contrast cu cu *lazy evaluation* este *eager evaluation (strict evaluation)*
    - se evaluatează o expresie cand se știe că e posibil să fie folosită valoarea acesteia
  - **exemplu:**  $f(x, y) = 2 * x$ ;  $k=f(d, e)$ 
    - *lazy evaluation* – se evaluatează doar  $d$

- *eager evaluation* – evaluăm  $d$  și  $e$  când calculăm  $k$ , chiar dacă  $y$  nu e folosit în funcție
- majoritatea limbajelor de programare (C, Java, Python) folosesc evaluarea strictă ca și mecanism implicit de evaluare
- Lazy Python

## 2. Limbajul Lisp

- 1958 John McCarthy elaborează o primă formă a unui limbaj (**LISP - LISt Processing**) destinat prelucrărilor de liste, formă ce se baza pe ideea transcrierii în acel limbaj de programare a expresiilor algebrice.
- Câteva caracteristici
  - calcule cu expresii simbolice în loc de numere;
  - reprezentarea expresiilor simbolice și a altor informații prin structura de listă;
  - compunerea funcțiilor ca instrument de formare a unor funcții mai complexe;
  - utilizarea recursivității în definiția funcțiilor;
  - reprezentarea programelor Lisp ca date Lisp;
  - funcția Lisp **eval** care servește deopotrivă ca definiție formală a limbajului și ca interpret;
  - colectarea spațiului disponibil (*garbage collection*) ca mijloc de tratare a problemei dealocării.
- Consideră *funcția* ca obiect fundamental – transmis ca parametru, returnat ca rezultat al unei prelucrări, parte a unei structuri de date.
- Domeniul de utilizare al limbajului Lisp este cel al *calculului simbolic*, adică al prelucrărilor efectuate asupra unor siruri de simboluri grupate în expresii. Una dintre cauzele forței limbajului Lisp este posibilitatea de manipulare a unor obiecte cu structură ierarhizată.
- **Domenii de aplicare** – învățare automată (*machine learning*), bioinformatică, comerț electronic (Paul Graham - <http://www.paulgraham.com/avg.html>), sisteme expert, *data mining*, prelucrarea limbajului natural, agenți, demonstrarea teoremelor, învățare automată, înțelegerea vorbirii, prelucrarea imaginilor, planificarea roboților.

*“Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.”* (Paul Graham, 2001)

- Editorul GNU Emacs de sub Unix e scris în Lisp.
- **GNU CLisp**, GNU Emacs Lisp.

### 3. Elemente de bază ale limbajului Lisp

- Un program Lisp prelucrează expresii simbolice (*S-expresii*). Chiar programul este o astfel de S-expresie.
- Modul ușual de lucru al unui sistem Lisp este cel conversațional (interactiv), interpretorul alternând prelucrările de date cu intervenția utilizatorului.
- În Lisp există standardul **CommonLisp** și standardul **CLOS** (Common Lisp Object System) pentru programare orientată obiect.
  - **Common Lisp** e un *limbaj funcțional* (dar și imperativ, orientat-obiect, putând fi folosit în manieră declarativă).
- Mecanismul implicit de evaluare în **CommonLisp** – evaluarea strictă (*eager/strict evaluation*).
  - Extensie CLAZY – evaluare întârziată (*lazy evaluation*) în Common Lisp
- Verificarea tipurilor se face dinamic (la execuție) – *dynamic type checking*.
- Obiectele de bază în Lisp sunt *atomii* și *listele*.
  - Datele primare (*atomii*) sunt numerele și simbolurile (simbolul este echivalentul Lisp al conceptului de variabilă din celelalte limbi). Sintactic, simbolul apare ca un sir de caractere (primul fiind o literă); semantic, el desemnează o S-expresie. Atomii sunt utilizați la construirea listelor (majoritatea S-expresiilor sunt liste).
    - O *listă* este o secvență de atomi și/sau liste.
      - *Liste liniare* → (1 2 3 4 5)
      - *Liste neliniare* → (1 (2 (3) 4) 5 6 (7))
- În Lisp s-a adoptat notația *prefixată* (notație ce sugerează și interpretarea operațiilor drept funcții), simbolul de pe prima poziție a unei liste fiind numele funcției ce se aplică.
- *Evaluarea* unei S-expresii înseamnă extragerea (determinarea) valorii acesteia. Evaluarea valorii funcției are loc după evaluarea argumentelor sale.
- În LISP nu există nedeterminism.

## 4. Structuri dinamice de date

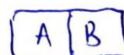
- Una dintre cele mai cunoscute și mai simple SDD este **lista liniară simplu înlanțuită** (LLSI).
- Un element al listei este format din două câmpuri: **valoare** și **legătură** spre elementul următor. Legăturile ne dău informații de natură structurală ce stabilesc relații de ordine între elemente):

valoare	legătură
C1	C2

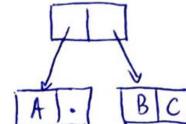
- Variațuni ale conținutului acestor două câmpuri generează alte genuri de structuri: dacă C1 conține un pointer atunci se generează arbori binari, iar dacă C2 poate fi și altceva decât pointer atunci apar **așa-numitele perechi cu punct** (elemente cu două câmpuri-dată).
  - se pot astfel forma elemente ale căror câmpuri pot fi în egală măsură ocupate de informații atomice (numere sau simboluri) și de informații structurale (referințe spre alte elemente). Reprezentarea grafică a unei astfel de structuri este cea a unui arbore binar (structura arborescentă).
- Orice listă are echivalent în notația perechilor cu punct**, însă NU orice pereche cu punct are echivalent în notația de listă (în general numai notațiile cu punct în care la stânga parantezelor închise se află NIL pot fi reprezentate în notația de listă). În Lisp, ca și în Pascal, NIL are semnificația de pointer nul.
- Definiția recursivă a echivalenței între **liste și perechi cu punct** în Lisp:
  - dacă A este *atom*, atunci lista (A) este echivalentă cu perechea cu punct (A . NIL)
  - dacă lista (l<sub>1</sub> l<sub>2</sub>...l<sub>n</sub>) este echivalentă cu perechea cu punct <p> atunci lista (l<sub>1</sub> l<sub>2</sub>...l<sub>n</sub>) este echivalentă cu perechea cu punct (l<sub>1</sub> . <p>)
 
$$(A \ B) \quad (A \cdot (B))$$

### 4.1 Exemple

- (A . B) nu are echivalent listă;



- ((A . NIL) . (B . C)) nu are echivalent listă



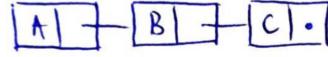
- (A)  $\leftrightarrow$  (A . NIL)



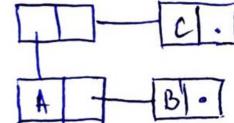
- $(A \ B) \leftrightarrow (A \ . \ B)$



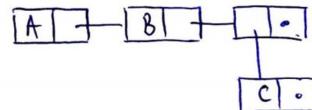
- $(A \ B \ C) \leftrightarrow (A \ . \ (B \ . \ (C \ . \ NIL)))$



- $((A \ B) \ C) \leftrightarrow ((A \ . \ (B \ . \ NIL)) \ . \ (C \ . \ NIL))$



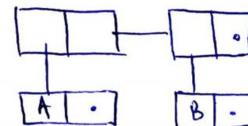
- $(A \ B \ (C)) \leftrightarrow (A \ . \ (B \ . \ ((C \ . \ NIL) \ . \ NIL)))$



- $((A)) \leftrightarrow ((A \ . \ NIL) \ . \ NIL)$



- $((A) \ (B)) \leftrightarrow ((A \ . \ NIL) \ . \ ((B \ . \ NIL) \ . \ NIL))$



**TEMĂ** Desenați structura fiecăreia din listele de mai jos:

- $((A \ . \ NIL) \ . \ ((B \ . \ NIL) \ . \ NIL)) = ((A) \ (B))$
- $((NIL)) = ((NIL \ . \ NIL)) \ . \ NIL$
- $(( )) = (NIL \ . \ NIL)$
- $(A \ (B \ . \ C)) = (A \ . \ ((B \ . \ C) \ . \ NIL))$

## 5. Reguli sintactice

- atom numeric** (n) - un sir de cifre urmat sau nu de caracterul '.' și precedat sau nu de semn ('+' sau '-')
- atom sir de caractere** (c) - sir de caractere cuprins între ghilimele

3. **simbol** (s) - un sir de caractere, altele decât delimitatorii: spațiu () ' " . virgula = [ ]
  - delimitatorii pot să apară într-un simbol numai dacă sunt evitați (folosind convenția cu "\")
4. **atom** (a) – poate fi
  - n - atom numeric
  - c – atom sir de caractere
  - s – simbol
5. **listă** (l) - poate fi
  - () lista vidă – NIL **!!! Lista vida este singura lista care este considerată atom**
  - (e<sub>1</sub> e<sub>2</sub> ... e<sub>n</sub>), n≥1 unde e, e<sub>1</sub>...,e<sub>n</sub> sunt **S-expresii**
6. **pereche cu punct** (pp) – este o construcție de forma (e<sub>1</sub> . e<sub>2</sub>) unde e<sub>1</sub> și e<sub>2</sub> sunt **S-expresii**
7. **S-expresie** (e) – poate fi
  - a - atom
  - l – listă
  - pp – pereche cu punct
8. **formă** (f) - o S-expresie evaluabilă
9. **program Lisp** este o succesiune de **forme** (S-expresii evaluabile).

## 6. Reguli de evaluare

- (a) un **atom numeric** se evaluatează prin numărul respectiv
- (b) un **sir de caractere** se evaluatează chiar prin textul său (inclusiv ghilimelele);
- (c) o **listă** este evaluabilă (adică este **formă**) doar dacă primul ei element este numele unei funcții/operator, caz în care mai întâi se evaluatează toate argumentele, după care se aplică funcția acestor valori.

**Observație** Funcția QUOTE întoarce chiar S-expresia argument, ceea ce este echivalent cu oprirea încercării de a evalua argumentul. În locul lui QUOTE se va putea utiliza caracterul ' (apostrof).

*Esența Lisp-ului constă din prelucrarea S-expresiilor. Datele au aceeași formă cu programele, ceea ce permite lansarea în execuție ca programe a unor structuri de date precum și modificarea unor programe ca și cum ele ar fi date obișnuite.*

## 6.1 Exemple

> 'A A	> (quote A) A
> (A) the function A is undefined	> (NIL) the function NIL is undefined
> '(A) (A)	> '(NIL) (NIL)
> () NIL	> ()() the function NIL is undefined
> NIL NIL	> '()() (())
> (A.B) the function A\B is undefined	> '(A.B) (A\B)

## 7. Funcții Lisp

Prelucrările de liste se pot face la:

- nivel superficial
- la orice nivel

**Exemplu.** Să se calculeze suma atomilor numerici dintr-o listă neliniară

- la nivel superficial    (suma '(1 (2 a (3 4) b 5) c 1)) → 2
- la toate nivelurile    (suma '(1 (2 a (3 4) b 5) c 1)) → 16

### (CONS e<sub>1</sub> e<sub>2</sub>): I sau pp

- funcția **constructor**
- se evaluatează argumentele și apoi se trece la evaluarea funcției
- formează o pereche cu punct având reprezentările celor două SE în cele două câmpuri. Elementul CONS construit în acest fel se numește *celulă CONS*. Valorile argumentelor nu sunt afectate.

Iată câteva exemple:

- (CONS 'A 'B) = (A . B)
- (CONS 'A '(B)) = (A B)
- (CONS '(A B) 'C)) = ((A B) C)
- (CONS '(A B) '(C D)) = ((A B) C D)
- (CONS 'A '(B C)) = (A B C)
- (CONS 'A (CONS 'B '(C))) = (A B C)

**(CAR I sau pp): e**

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- extrage primul element al unei liste sau partea stângă a unei perechi cu punct

**(CDR I sau pp): e**

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- extrage lista fără primul element sau respectiv partea dreaptă a unei perechi cu punct.

Iată câteva exemple:

- (CAR '(A B C)) = A
- (CAR '(A . B)) = A
- (CAR '((A B) C D)) = (A B)
- (CAR (CONS '(B C) '(D E))) = (B C)
- (CDR '(A B C)) = (B C)
- (CDR '(A . B)) = B
- (CDR '((A B) C D)) = (C D)
- (CDR (CONS '(B C) '(D E))) = (D E)

CONS reface o listă pe care CAR și CDR au secționat-o. De observat, însă, că obiectul obținut ca urmare a aplicării funcțiilor CAR, CDR și CONS, nu este același cu obiectul de la care s-a plecat:

- (CONS (CAR '(A B C)) (CDR '(A B C))) = (A B C)
- (CAR (CONS 'A '(B C))) = A
- (CDR (CONS 'A '(B C))) = (B C)

La folosirea repetată a funcțiilor de selectare se poate utiliza prescurtarea  $Cx_1x_2\dots x_nR$ , echivalentă cu  $Cx_1R$  sau  $Cx_2R$  sau ... sau  $Cx_nR$ , unde caracterele  $x_i$  sunt fie 'A', fie 'D'. În funcție de implementări, se vor putea utiliza în această compunere cel mult trei sau patru  $Cx_iR$ :

- (CAADDR '((A B) C (D E))) = D
- (CDAAAR '(((A) B) C) (D E))) = NIL
- (CAR '(CAR (A B C))) = CAR

## 7.1 Exemple

### Operare CLisp

- (ed) – editor
- se scrie funcția **fct** în fișierul **f.lsp**. De exemplu, fișierul **f.lsp** conține următoarea definiție
 

```
(defun fct(l)
  (cdr l)
)
```

- se încarcă fișierul **f.lsp**  
    > (load 'f)
- dacă nu sunt erori, se evaluează funcția **fct**  
    > (fct '(1 2)) → (2)

**EXEMPLU 7.1.1** Să se genereze o listă cu numerele întregi din intervalul [a, b].

```
(defun interval (a b)
    ;; returnează o listă cu numerele întregi din intervalul [a, b]
    (if (> a b)
        nil
        (cons a (interval (+ a 1) b)))
    )
)
```

**EXEMPLU 7.1.2** Să se calculeze suma atomilor numerici de la nivelul superficial dintr-o listă neliniară.

Model recursiv

$$\text{suma}(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + \text{suma}(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ \text{suma}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
; suma atomilor de la nivelul superficial al unei liste neliniare
;(suma '(1 (2 (3 4) 5) 1)) → 2
(defun suma(l)
    ; forma COND – forma condițională: permite ramificarea prelucrărilor
    (cond
        ((null l) 0)
        ; NUMBERP – returnează T dacă argumentul e număr
        ((numberp (car l)) (+ (car l) (suma (cdr l))))
        (t (suma (cdr l)))
    )
)
```

**EXEMPLU 7.1.2** Să se calculeze suma atomilor numerici de la orice nivel dintr-o listă neliniară.

Model recursiv

$$\text{suma}(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + \text{suma}(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ \text{suma}(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom} \\ \text{suma}(l_1) + \text{suma}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

; să se calculeze suma atomilor numerici dintr-o listă neliniară  
; (la toate nivelurile) (suma '(1 (2 a (3 4) b 5) c 1)) → 16

```
(defun suma(l)
  (cond
    ((null l) 0)
    ((numberp (car l)) (+ (car l) (suma (cdr l))))
    ; ATOM – returnează T dacă argumentul e atom
    ((atom (car l)) (suma (cdr l)))
    ; ultima clauză e pentru primul element listă
    (t (+ (suma (car l)) (suma (cdr l)))))
  ))
```

## CURS 8

### Funcții LISP. Exemple

#### Cuprins

1.	Funcții Lisp (cont.).....	1
1.1	Predicate de bază în Lisp .....	2
1.2	Operații logice.....	3
1.3	Operații aritmetice .....	4
1.4	Operatori relaționali .....	4
2.	Ramificarea prelucrărilor. Funcția COND .....	4
3.	Definirea funcțiilor utilizator. Funcția DEFUN.....	5
4.	Exemple .....	7

#### 1. Funcții Lisp (cont.)

##### (LIST e1 e2...): l

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- întoarce lista valorilor argumentelor la nivel superficial.

Câteva exemple:

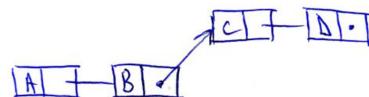
- (LIST 'A 'B) = (A B)
- (LIST '(A B) 'C) = ((A B) C)
- (LIST 'A) = (A) ; echivalent cu (CONS A NIL)
- (LIST '(A B C) NIL) = ((A B C) NIL)

##### (APPEND e1 e2...en): e

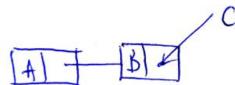
- se evaluatează argumentele și apoi se trece la evaluarea funcției
- copiază structura fiecărui argument, înlocuind CDR-ul fiecărui ultim CONS cu argumentul din dreapta. Întoarce lista rezultată.
- dacă toate S-expresiile argument sunt liste, atunci efectul este concatenarea listelor
- în CLisp argumentele nu pot fi atomi, cu excepția ultimului
- funcția APPEND este puternic consumatoare de memorie; prima listă argument este recopiată înainte de a fi legată de următoarea

Câteva exemple:

- (APPEND '(A B) '(C D)) = (A B C D)



- (APPEND '(A B) 'C) = (A B . C)



## Funcții LISP (cont). Exemple

Page 2 of 13

- (APPEND '(A B C) '()) = (A B C)
- (APPEND (cons 'A 'B) 'C) = (A . C)
- (APPEND '(A . B) 'F) = (A . F)

	cons	list	append
'A 'B	(A . B)	(A B)	Error
'A '(B C D)	(A B C D)	(A (B C D))	Error
'(A B C)'(D E F)	((A B C) D E F)	((A B C) (D E F))	(A B C D E F)
'(A B C)'D	((A B C) . D)	((A B C) D)	(A B C . D)
'A 'B 'C 'D	Error	(A B C D)	Error
'(A B) '(C D) '(E F)	Error	((A B) (C D) (E F))	(A B C D E F)
'(A B) 'C '(E F) 'D	Error	((A B) C (E F) D)	Error
'(A B) '(E F) 'D	Error	((A B) (E F) D)	(A B E F . D)

### (LENGTH l): n

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- întoarce numărul de elemente ale unei liste la nivel superficial

## 1.1 Predicate de bază în Lisp

- Limbajul Lisp prezintă atomii speciali NIL, cu semnificația de **fals**, și T, cu semnificația de **adevărat**. Ca și în alte limbi, funcțiile care returnează o valoare logică vor returna exclusiv NIL sau T. Totuși, se acceptă că orice valoare diferită de NIL are semnificația de adevărat.

### (ATOM e) : T, NIL

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- T dacă argumentul este un atom și NIL în caz contrar.

Câteva exemple:

- (ATOM 'A) = T;
- (ATOM A) = depinde la ce anume se evaluatează A;
- (ATOM '(A B C)) = NIL;
- (ATOM NIL) = T;

### (LISTP e) : T, NIL

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- T dacă argumentul este o listă și NIL în caz contrar.

Câteva exemple:

- (LISTP 'A) = NIL;
- (LISTP A) = T dacă A se evaluatează la o listă;
- (LISTP '(A B C)) = T;
- (LISTP NIL) = T;

### (EQUAL e1 e2) : T, NIL

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- întoarce T dacă valorile argumentelor sunt S-expresii echivalente (adică dacă cele două S-expresii au aceeași structură)

De exemplu,

## Functii LISP (cont). Exemple

Page 3 of 13

$e_1$	$e_2$	EQUAL
'(A B)	'(A B)	T
'(A B)	'(A (B))	NIL
3.0	3.0	T
8	8	T
'a	'a	T

### (NULL e) : T, NIL

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- Întoarce T dacă argumentul se evaluatează la o listă vidă sau atom nul și NIL în caz contrar.

### (NUMBERP e) : T, NIL

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- Verifică dacă argumentul este număr sau nu.

## 1.2 Operații logice

### (NOT e) : T, NIL

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- Întoarce T dacă argumentul e este evaluat la NIL; în caz contrar, întoarce NIL. E echivalentă cu funcția NULL.

### (AND e1 e2 ... ) : e

- Evaluarea argumentelor face parte din procesul de evaluare al funcției
- Se evaluatează de la stânga la dreapta până la primul NIL, caz în care se returnează NIL; în caz contrar rezultatul este valoarea ultimului argument.

De exemplu, forma **(AND l (car l))** este echivalentă cu următoarea structură alternativă

**Dacă**  $l = \emptyset$  **atunci**  
                returnează  $\emptyset$   
**altfel**  
                returnează  $(\text{car } l)$   
**SfDacă**

### (OR e1 e2 ... ) : e

- Evaluarea argumentelor face parte din procesul de evaluare al funcției
- Se evaluatează de la stânga la dreapta până la primul element evaluat la o valoare diferită de NIL, caz în care se returnează acea valoare; în caz contrar rezultatul este NIL.

De exemplu, forma **(OR (cdr l) (car l))** este echivalentă cu următoarea structură alternativă

**Dacă**  $(\text{cdr } l) \neq \emptyset$  **atunci**  
                returnează  $(\text{cdr } l)$   
**altfel**

## Functii LISP (cont). Exemple

Page 4 of 13

returnează (car l)

**SfDacă**

!!! AND și OR nu sunt funcții, sunt operatori speciali

### 1.3 Operații aritmetice

**(+ n1 n2 ... ) : n**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n_1 + n_2 + \dots$

**(- n1 n2 ... ) : n**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n_1 - n_2 - \dots$

**(\* n1 n2 ... ) : n**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n_1 * n_2 * \dots$

**(/ n1 n2 ... ) : n**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n_1 / n_2 / \dots$

**(MAX n1 n2 ... ) : n**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- rezultat: maximul valorilor argumentelor

**(MIN n1 n2 ... ) : n**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- rezultat: minimul valorilor argumentelor

Pentru numere întregi se poate folosi MOD (rest).

### 1.4 Operatori relaționali

Sunt cei uzuali: = (doar pentru numere), <, <=, >, >=

## 2. Ramificarea prelucrărilor. Funcția COND

Funcția COND este asemănătoare selectorilor CASE sau SWITCH din Pascal, respectiv C.

**(COND l1 l2...ln): e**

- evaluarea argumentelor face parte din procesul de evaluare al funcției

In descrierea de mai sus **I1, I2,...In** sunt liste nevide de lungime arbitrară (**f<sub>1</sub> f<sub>2</sub> ... f<sub>n</sub>**) numite *clauze*. COND admite oricâte clauze ca argumente și oricâte forme într-o clauză. Iată modul de funcționare a funcției COND:

- se parcurg pe rând clauzele în ordinea apariției lor în apel, evaluându-se doar primul element din fiecare clauză până se întâlnește unul diferit de NIL. Clauza respectivă va fi selectată și se trece la evaluarea în ordine a formelor  $f_2, f_3, \dots, f_n$ . Se întoarce valoarea ultimei forme evaluate din clauza selectată;
- dacă nu se selectează nici o clauză, COND întoarce NIL.

Următoarea secvență

```
(COND
  ((> X 5) (CONS 'A '(B)))
  (T 'A)
)
```

returnează (A B) dacă X este 7, respectiv A dacă X este 4.

Ce returnează secvența

```
(COND
  ((> X 5) (LIST 'A) (CONS 'A '(B)))
  (T 'A)
)
```

în cazul în care X este 10?

### 3. Definirea funcțiilor utilizator. Funcția DEFUN

**(DEFUN s l f1 f2...): s**

Funcția DEFUN creează o nouă funcție având ca nume primul argument (simbolul s), iar ca parametri formalii elementele simboluri ale listei ce constituie al doilea argument; corpul funcției create este alcătuit din una sau mai multe forme aflate, ca argumente, pe pozițiile a treia și eventual următoarele (evaluarea acestor forme la execuție reprezintă efectul secundar). Se întoarce numele funcției create. Funcția DEFUN nu-și evaluatează nici un argument.

Apelul unei funcții definită prin  
**(DEFUN fnume (p<sub>1</sub> p<sub>2</sub> ... p<sub>n</sub>)**

```
  ...
)
```

este o formă

**(fnume arg<sub>1</sub> arg<sub>2</sub> ... arg<sub>n</sub>)**

unde **fnume** este un simbol, iar **arg<sub>i</sub>** sunt forme; evaluarea apelului deține astfel:

## Funcții LISP (cont). Exemple

Page 6 of 13

- (a) se evaluatează argumentele  $arg_1, arg_2, \dots, arg_n$ ; fie  $v_1, v_2, \dots, v_n$  valorile lor;
- (b) fiecare parametru formal din definiția funcției este legat la valoarea argumentului corespunzător din apel ( $p_1$  la  $v_1$ ,  $p_2$  la  $v_2$ , ...,  $p_n$  la  $v_n$ ); dacă la momentul apelului simbolurile reprezentând parametrii formalii aveau deja valori, acestea sunt salvate în vederea restaurării ulterioare;
- (c) se evaluatează în ordine fiecare formă aflată în corpul funcției, valoarea ultimei forme fiind întoarsă ca valoare a apelului funcției;
- (d) se restaurează valorile parametrilor formalii, adică  $p_1, p_2, \dots, p_n$  se “dezleagă” de valorile  $v_1, v_2, \dots$  și se leagă din nou la valorile corespunzătoare salvate (dacă este cazul).

**Observație.** DEFUN poate redifini și funcțiile sistem standard. Spre exemplu dacă se redefinește funcția CAR astfel: (DEFUN CAR(L) (CDR L)), atunci (CAR '(1 2 3)) se va evalua la (2 3).

Următorul exemplu întoarce argumentul dacă acesta este atom, NIL dacă acesta este listă vidă și primul element dacă argumentul e listă.

```
(DEFUN PRIM (X)
  (COND
    ((ATOM X) X)
    ((NULL X) NIL) ;inutil
    (T (CAR X)))
  )
)
```

Următorul exemplu întoarce maximul valorilor celor două argumente.

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

Următorul exemplu întoarce ultimul element al unei liste, la nivel superficial.

```
(DEFUN ULTIM (X)
  (COND
    ((ATOM X) X)
    ((NULL (CDR X)) (CAR X))
    (T (ULTIM (CDR X)))
  )
)
```

Următorul exemplu rescrie CAR pentru a întoarce NIL dacă argumentul este atom și nu produce mesaj de eroare.

```
(DEFUN XCAR (X)
  (COND
```

```
((ATOM X NIL)
 (T (CAR X))
 )
)
```

**Observații.**

- (1). DEFUN poate redefini și funcțiile sistem standard. Spre exemplu dacă se redifineste funcția CAR astfel: (DEFUN CAR(L) (CDR L)), atunci (CAR '(1 2 3)) se va evalua la (2 3).  
(2). În cazul în care o funcție este redifinită cu o altă aritate (număr de argumente), este valabilă ultimă definiție a acesteia. De exemplu, fie următoarele definiții (în ordinea indicată)

```
(defun f(l)
      (car l)
)
(defun f(l1 l2)
      (cdr l2)
)
```

Evaluarea formei (f '(1 2 3)) produce eroare, pe când evaluarea (f '(1 2) '(3 4)) produce (4).

Fie următoarea definiție de funcție

```
(DEFUN F (X Y)
  (COND
    ((< X Y) X)
    (T Y)
  )
  Y
)
```

Care este efectul evaluării (F 2 5)?

## 4. Exemple

**EXEMPLU 4.1** Să se calculeze suma atomilor numerici de la orice nivel dintr-o listă neliniară.

### Modele recursive

**Varianta 1** (discutată în Cursul 7)

$$\text{suma}(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + \text{suma}(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ \text{suma}(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom} \\ \text{suma}(l_1) + \text{suma}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

## Functii LISP (cont). Exemple

Page 8 of 13

### Varianta 2

$$suma(l) = \begin{cases} l & \text{dacă } l \text{ atom numeric} \\ 0 & \text{dacă } l \text{ atom} \\ suma(l_1) \oplus suma(l_2 \dots l_n) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

; să se calculeze suma atomilor numerici dintr-o listă neliniară  
; (la toate nivelurile) (suma '(1 (2 a (3 4) b 5) c 1)) → 16

**EXEMPLU 4.2** Să se construiască lista obținută prin adăugarea unui element la sfârșitul unei liste.

(adaug '3 '(1 2)) → (1 2 3)  
(adaug '(3) '(1 2)) → (1 2 (3))  
(adaug '3 '()) → (3)

### Model recursiv

; se returnează lista  $(l_1, l_2, \dots, l_n, e)$

$$adaug(e, l_1 l_2 \dots l_n) = \begin{cases} (e) & \text{daca } l \text{ e vida} \\ l_1 \oplus adaug(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun adaug(e l)
  (cond
    ((null l) (list e)) ; (list e) sau (cons e nil)
    (t (cons (car l) (adaug e (cdr l)))))
  )
)
```

**EXEMPLU 4.3.** Metoda variabilei colectoare. Să se definească o funcție care inversează o listă liniară.

(invers '(1 2 3)) va produce (3 2 1).

Modelul recursiv este

$$invers(l_1 l_2 \dots l_n) = \begin{cases} \phi & \text{daca } l \text{ e vida} \\ invers(l_2 \dots l_n) \oplus l_1 & \text{altfel} \end{cases}$$

O definiție posibilă pentru funcția INVERS este următoarea:

## Functii LISP (cont). Exemple

Page 9 of 13

```
(DEFUN INVERS (L)
  (COND
    ((ATOM L) L)
    (T (APPEND (INVERS (CDR L)) (LIST (CAR L)))))
  )
)
```

Complexitatea timp e data de recurență

$$T(n) = \begin{cases} 1 & \text{daca } n = 0 \\ T(n - 1) + n & \text{altfel} \end{cases}$$

Problema este că o astfel de definiție consumă multă memorie. Eficiența aplicării funcțiilor Lisp (exprimată prin consumul de memorie) se măsoară prin numărul de CONS-uri pe care le efectuează. Să ne reamintim că (LIST arg) este echivalent cu (CONS arg NIL) și să subliniem de asemenea că funcția APPEND acționează prin copierea primului argument, care este apoi "lipit" de cel de-al doilea argument. Astfel, funcția REVERSE definită mai sus va realiza copierea fiecărui (REVERSE (CDR L)) înainte de "lipirea" sa la al doilea argument. De exemplu pentru lista (A B C D E) se vor copia listele NIL, (E), (E D), (E D C), (E D C B), deci pentru o listă de dimensiune N vom avea  $1 + 2 + \dots + (N-1) = N(N-1)/2$  folosiri de CONS-uri. Deci, complexitatea timp este  $\Theta(n^2)$ ,  $n$  fiind numărul de elemente din listă.

O soluție pentru a reduce complexitatea timp a operației de inversare este **folosirea metodei variabilei colectoare**: scrierea unei funcții auxiliare care utilizează doi parametri ( $Col$  = lista destinație și  $L$  = lista sursă), scopul ei fiind trecerea pe rând a câte unui element din  $L$  spre  $Col$ :

<b>L</b>	<b>Col</b>
(1, 2, 3)	∅
(2, 3)	(1)
(3)	(2, 1)
∅	(3, 2, 1)

Modelele recursive

$$\text{invers\_aux}(l_1 l_2 \dots l_n, Col) = \begin{cases} Col & \text{daca } l \neq \text{vida} \\ \text{invers\_aux}(l_2 \dots l_n, l_1 \oplus Col) & \text{altfel} \end{cases}$$

$$\text{invers}(l_1 l_2 \dots l_n) = \text{invers\_aux}(l_1 l_2 \dots l_n, \emptyset)$$

```
(DEFUN INVERS_AUX (L Col)
  (COND
    ((NULL L) Col)
```

## Functii LISP (cont). Exemple

Page 10 of 13

```
(T (INVERS_AUX (CDR L) (CONS (CAR L) Col)))
)
)
```

Ceea ce dorim noi se realizează prin apelul (INVERS\_AUX L()), dar să nu uităm ca am pornit de la necesitatea definirii unei funcții ce trebuie apelată cu (INVERS L). De aceea, funcția INVERS se va defini:

```
(DEFUN INVERS (L)
  (INVERS_AUX L())
)
```

Functia INVERS\_AUX are rolul de funcție auxiliară, ea punând în evidență rolul argumentului **Col** ca **variabilă colectoare** (variabilă ce colectează rezultatele parțiale până la obținerea rezultatului final). Se vor efectua atâtea CONS-uri cât este lungimea listei sursă L, deci complexitatea timp este  $\Theta(n)$ ,  $n$  fiind numărul de elemente din listă.

$$T(n) = \begin{cases} 1 & \text{daca } n = 0 \\ T(n - 1) + 1 & \text{altfel} \end{cases}$$

Să observăm deci că colectarea rezultatelor parțiale într-o variabilă separată poate contribui la micșorarea complexității algoritmului. Dar acest lucru nu este valabil în toate cazurile: sunt situații în care folosirea unei variabile colectoare crește complexitatea prelucrării, în cazul în care elementele se adaugă la finalul colectoarei (nu la începutul ei, ca în exemplul indicat).

**EXEMPLU 4.4.** Să se definească o funcție care să determine lista perechilor dintre un element dat și elementele unei liste.

(LISTA 'A '(B C D)) = ((A B) (A C) (A D))

Modelul recursiv este

$$\text{lista}(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ (e, l_1) \oplus \text{lista}(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN LISTA (E L)
  (COND
    ((NULL L) NIL)
    (T (CONS (LIST E (CAR L)) (LISTA E (CDR L))))
  )
)
```

**EXEMPLU 4.5.** Să se definească o funcție care să determine lista perechilor cu elemente în ordine strict crescătoare care se pot forma cu elementele unei liste numerice (se va păstra ordinea elementelor din listă).

(perechi '(3 1 5 0 4)) = ((3 5) (3 4) (1 5) (1 4))

## Functii LISP (cont). Exemple

Page 11 of 13

Vom folosi o funcție auxiliară care returnează lista perechilor cu elemente în ordine strict crescătoare, care se pot forma între un element și elementele unei liste.

$$(per \ '2 \ '(3 \ 1 \ 5 \ 0 \ 4)) = ((2 \ 3) \ (2 \ 5) \ (2 \ 4))$$

$$per(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ (e, l_1) \oplus per(e, l_2 \dots l_n) & \text{dacă } e < l_1 \\ per(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun per (e l)
  (cond
    ((null l) nil)
    (t (cond
          ((< e (car l)) (cons (list e (car l))(per e (cdr l))))
          (t (per e (cdr l)))
          )
        )
      )
    ))
```

$$perechi(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ per(l_1, l_2 \dots l_n) \oplus perechi(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun perechi (l)
  (cond
    ((null l) nil)
    (t (append (per (car l) (cdr l)) (perechi (cdr l)))))
    )
  ))
```

**EXEMPLU 4.6** Se dă o listă neliniară. Se cere să se dubleze valorile numerice de la orice nivel al listei, păstrând structura ierarhică a acesteia.

$$(\text{dublare} \ '(1 \ b \ 2 \ (c \ (3 \ h \ 4)) \ (d \ 6))) \rightarrow (2 \ b \ 8 \ (c \ (6 \ h \ 8)) \ (d \ 12))$$

### Varianta 1

$$dublare(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ 2l_1 \oplus dublare(l_2 \dots l_n) & \text{dacă } l_1 \text{ numeric} \\ l_1 \oplus dublare(l_2 \dots l_n) & \text{dacă } l_1 \text{ atom} \\ dublare(l_1) \oplus dublare(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun dublare(l)
  (cond
    ((null l) nil)
    ((numberp (car l)) (cons (* 2 (car l)) (dublare (cdr l))))
    ((atom (car l)) (cons (car l) (dublare (cdr l))))
    (t (cons (dublare (car l)) (dublare (cdr l)))))
  )
)
```

### Varianta 2

$$dublare(l) = \begin{cases} l & \text{dacă } l \text{ numar} \\ dublare(l_1) \oplus dublare(l_2 \dots l_n) & \text{dacă } l \text{ atom} \\ \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```
(defun dublare(l)
  (cond
    ((numberp l) (* 2 l))
    ((atom l) l)
    (t (cons (dublare (car l)) (dublare (cdr l)))))
  )
)
```

**EXEMPLU 4.7** Folosirea unei variabile colectoare poate crește complexitatea. Se dă o listă liniară. Care este efectul următoarei evaluări:

(lista '(1 a 2 b 3 c))  $\rightarrow$  ???

Scrieți modelele matematice pentru fiecare funcție.

### Varianta 1 – direct recursiv (fără variabilă colectoare)

```
(defun lista (l)
  (cond
    ((null l) nil)
    ((numberp (car l)) (cons (car l) (lista (cdr l))))
    (t (lista (cdr l)))
  )
)
```

Care este complexitatea timp în caz defavorabil?

**Varianta 2 – cu variabilă colectoare**

```
(defun lista_aux (l col)
  (cond
    ((null l) col)
    ((numberp (car l)) (lista_aux (cdr l) (append col (list (car l)))))
    (t (lista_aux (cdr l) col))
  )
)

(defun lista (l)
  (lista_aux l nil)
)
```

Care este complexitatea timp în caz defavorabil?

**EXEMPLU 4.8** Se dă o listă liniară. Care este efectul funcției PARCURG?

```
(defun parcurg_aux(L k col)
  (cond
    ((null L) nil)
    ((= k 0) (list col L))
    (t (parcurg_aux (cdr L) (- k 1) (cons (car l) col)))
  )
)

(defun parcurg (L k)
  (parcurg_aux L k nil)
)
```

(parcurg '(1 2 3 4 5) 3) → ???

## CURS 9

### Funcțiile SET, SETQ, SETF. Arbori binari. Exemple

#### Cuprins

1.	Arbori binari.....	1
2.	Exemple .....	2
3.	SET, SETQ, SETF .....	9

#### 1. Arbori binari

Un arbore binar se poate memora sub forma unei liste, în următoarele două moduri:

**V1** Un arbore având

- rădăcină,
- subarborii subarbore-stâng și subarbore-drept

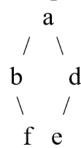
se va reprezenta sub forma unei liste neliniare de forma

(rădăcină lista-subarbore-stâng lista-subarbore-drept)

unde

- lista-subarbore-stâng reprezintă lista asociată (în memorarea sub forma V1) a subarborelui stâng al nodului rădăcină
- lista-subarbore-drept reprezintă lista asociată (în memorarea sub forma V1) a subarborelui drept al nodului rădăcină

De exemplu arboarele



se reprezintă, în varianta **V1** sub forma listei (a (b () (f)) (d (e)))

Reprezentarea V1 este cea mai potrivită pentru reprezentarea sub formă de listă a unui arbore cu rădăcină, fiind adecvată definiției recursive a unui arbore (binar).

**V2** Un arbore având

- rădăcină,
- nr-arborei subarbori
- subarborii subarbore-stâng și subarbore-drept

se va reprezenta sub forma unei liste liniare de forma

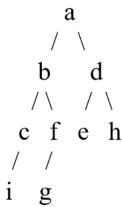
(rădăcină nr-subarbori lista-subarbore-stâng lista-subarbore-drept)

unde

- lista-subarbore-stâng reprezintă lista asociată (în memorarea sub forma V2) subarborelui stâng al nodului rădăcină
- lista-subarbore-drept reprezintă lista asociată (în memorarea sub forma V2) subarborelui drept al nodului rădăcină

Dezavantajul reprezentării **V2** este dat faptul că nu este potrivită pentru un arbore ordonat (de ex., în cazul arborelui binar nu se face distincție între subarborele stâng și cel drept).

De exemplu arborele



se reprezintă, în varianta **V2** sub forma listei (a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)

**Observatie.** Reprezentările V1 și V2 pot fi generalizate pentru arbori  $n$ -ari, varianta V1 fiind cea mai potrivită.

## 2. Exemple

**EXEMPLU 2.1** Se dă un arbore binar reprezentat în V1 (a se vedea Secțiunea 1). Să se determine lista liniară a nodurilor obținute prin parcurgerea arborelui în inordine.

(inordine '(a (b () (f)) (d (e)))) = (b f a e d)

În reprezentarea unui AB în varianta V1, sub forma unei liste  $l$  de forma (rădăcină lista-subarbore-stâng lista-subarbore-drept), observăm următoarele

- (car l) – primul element al listei este rădăcina arbrelui
- (cadr l) – al doilea element al listei, la nivel superficial, este subarborele stâng
- (caddr l) – al treilea element al listei, la nivel superficial, este subarborele drept

Model recursiv

$$\text{inordine}(l_1 l_2 l_3) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ \text{inordine}(l_2) \oplus l_1 \oplus \text{inordine}(l_3) & \text{altfel} \end{cases}$$

## Functii LISP (cont). Exemple. Arboi binari

Page 3 of 11

```
(defun inordine(l)
  (cond
    ((null l) nil)
    (t (append (inordine (cadr l)) (list (car l)) (inordine (caddr l))))
      ;(t (append (inordine (cadr l)) (cons (car l) (inordine (caddr l)))))
```

**EXEMPLU 2.2** Se dă un arbore binar reprezentat în V2 (a se vedea Secțiunea 2). Să se determine lista nodurilor obținute prin parcurgerea arborelui în inordine.

În reprezentarea unui AB în V2, față de reprezentarea V1, nu este clar identificat subarborele stâng și subarborele drept, pentru ca AB să poată fi prelucrat recursiv.

**Idee:** Dacă am reuși să extragem lista corespunzătoare (reprezentării în V2) subarborelor stâng și drept, problema s-ar reduce la cea prezentată în EXEMPLU 3.3.

Vom folosi o funcție auxiliară pentru a determina subarborele stâng al unui AB reprezentat în V2 (presupunem reprezentarea corectă).

(stang '(a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (b 2 c 1 i 0 f 1 g 0)

Vom folosi o funcție auxiliară, **parcurg\_st**, care va parcurge lista începând cu al 3-lea element și care va returna subarborele stâng. **Idee:** se colectează elementele, începând cu al 3-lea element al listei, ....**până când?**

(parcurg\_st '(b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (b 2 c 1 i 0 f 1 g 0)

*stang(l<sub>1</sub> l<sub>2</sub> ... l<sub>n</sub>) = parcurg\_st(l<sub>3</sub> ... l<sub>n</sub>, 0, 0)*

nv – număr vârfuri

nm – număr muchii

$$\begin{aligned} \text{parcurg\_st}(l_1 l_2 \dots l_k, nv, nm) \\ = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ \emptyset & \text{daca } nv = 1 + nm \\ l_1 \oplus l_2 \oplus \text{parcurg\_st}(l_3 \dots l_k, nv + 1, nm + l_2) & \text{altfel} \end{cases} \end{aligned}$$

```
(defun parcurg_st(arb nv nm)
  (cond
    ((null arb) nil)
    ((= nv (+ 1 nm)) nil)
    (t (cons (car arb) (cons (cadr arb) (parcurg_st (caddr arb) (+ 1 nv) (+ (cadr arb) nm)))))))
  )
)

(defun stang(arb)
```

```
(parcurg_st (cddr arb) 0 0)
)
```

**Temă.** Scriptă funcția pentru determinarea subarborelui drept.

- 1) Se poate folosi aceeași idee ca la parcurgerea pentru subarborele stâng, doar că în momentul în care  $nv=1+nm$ , se va returna lista rămasă
- 2) Pentru a reduce complexitatea timp, se poate defini o sigură funcție care determină atât subarborele stâng, cât și cel drept.

```
(drept '(a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (d 2 e 0 h 0)
```

**Observație.** Se poate folosi o funcție care să returneze atât subarborele stâng, cât și subarborele drept.

Având funcțiile anterioare care determină subarborele stâng și cel drept, lista nodurilor obținute prin parcurgerea arborelui în inordine se va face similar modelului recursiv descris în **EXEMPLU 2.1**.

$$\begin{aligned} & \text{inordine}(l_1 l_2 \dots l_n) \\ &= \begin{cases} \emptyset & \text{dacă } l \text{ e vida} \\ \text{inordine}(\text{stang}(l_1 l_2 \dots l_n)) \oplus l_1 \oplus \text{inordine}(\text{drept}(l_1 l_2 \dots l_n)) & \text{altfel} \end{cases} \end{aligned}$$

**EXEMPLU 2.3** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se determine lista (mulțimea) submulțimilor listei.

```
(subm '(1 2)) → ((0) (2) (1) (1 2))
```

? Cum obținem lista submulțimilor (1 2) dacă știm să generăm lista submulțimilor listei (2)? ((0) (2))

**Idee:** Dacă lista e vidă, submulțimea sa e lista vidă. Pentru determinarea submulțimilor unei liste [E|L], care are capul E și coada L, vom proceda în felul următor:

- i. determină o submulțime a listei L
- ii. plasează elementul E pe prima poziție într-o submulțime a listei L

Vom folosi o funcție auxiliară **insPrimaPoz(e, l)** care are ca parametri un element e și o listă l de liste liniare și returnează o copie a listei l în care listă se inserează e pe prima poziție.

```
(insPrimaPoz '3 '((0) (2) (1) (1 2)) → ((3) (3 2) (3 1) (3 1 2))
```

; l este o listă de liste liniare

## Functii LISP (cont). Exemple. Arbori binari

Page 5 of 11

; se inserează  $e$  pe prima poziție, în fiecare listă din  $l$ , și se returnează rezultatul

$$\text{insPrimaPoz}(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca lista e vida} \\ (e, l_1) \oplus \text{insPrimaPoz}(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun insPrimaPoz(e l)
  (cond
    ((null l) nil)
    (t (cons (cons e (car l)) (insPrimaPoz e (cdr l)))))
  )
)
```

;  $l$  este o mulțime reprezentată sub forma unei liste liniare

$$\text{subm}(l_1 l_2 \dots l_n) = \begin{cases} (\emptyset) & \text{daca lista e vida} \\ \text{subm}(l_2 \dots l_n) \oplus \text{insPrimaPoz}(l_1, \text{subm}(l_2 \dots l_n)) & \text{altfel} \end{cases}$$

```
(defun subm(l)
  (cond
    ((null l) (list nil))
    (t (append (subm (cdr l)) (insPrimaPoz (car l) (subm (cdr l)))))))
)
```

În codul Lisp scris anterior observăm faptul că apelul recursiv `(subm (cdr l))` din cea de-a doua clauză COND se repetă, ceea ce evident, nu este eficient din perspectiva complexității timp. O soluție pentru evitarea acestui apel repetat va fi folosire unei funcții anonime LAMBDA (se va discuta în [Cursul 10](#)).

De asemenea, în [Cursul 11](#) vom discuta despre simplificarea implementării funcției `subm`, renunțând la utilizarea funcției auxiliare, folosind o funcție MAP.

**EXEMPLU 2.4** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se determine lista (mulțimea) permutărilor listei inițiale.

(PERMUTĂRI '(1 2 3)) → ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

? Cum obținem lista permutărilor  $(1 \ 2 \ 3)$  dacă știm să generăm lista permutărilor listei  $(2 \ 3)$ ?  $(2 \ 3) \ (3 \ 2)$

**Idee:** Dacă lista e vidă, lista permutărilor sale este lista vidă. Pentru determinarea permutărilor unei liste  $[E|L]$ , care are capul  $E$  și coada  $L$ , vom proceda în felul următor:

1. determină o permutare  $L_1$  a listei  $L$ ;

## Funcții LISP (cont). Exemple. Arbori binari

Page 6 of 11

2. plasează elementul E pe toate pozițiile listei L1 și produce în acest fel lista X care va fi o permutare a listei inițiale [E|L].

Vom folosi câteva funcții auxiliare:

- 1) o funcție care returnează lista obținută prin inserarea unui element E pe o anumită poziție N într-o listă L ( $1 \leq N \leq$  lungimea listei L+1).

$$\begin{aligned} (\text{INS } '1 2 '(2 3)) &= (2 1 3) \\ (\text{INS } '1 3 '(2 3)) &= (2 3 1) \end{aligned}$$

Modelul recursiv

$$\text{ins}(e, n, l_1 l_2 \dots l_k) = \begin{cases} (e \ l_1 l_2 \dots l_k) & \text{daca } n = 1 \\ l_1 \oplus \text{ins}(e, n - 1, l_2 \dots l_k) & \text{altfel} \end{cases}$$

```
(DEFUN INS (E N L)
  (COND
    ((= N 1) (CONS E L))
    (T (CONS (CAR L) (INS E (- N 1) (CDR L)))))
  )
)
```

- 2) o funcție care să returneze mulțimea formată din liste obținute prin inserarea unui element E pe pozițiile 1, 2, ..., lungimea listei L+1 într-o listă L.

$$(\text{INSERARE } '1 '(2 3)) = ((2 3 1) (2 1 3) (1 2 3))$$

**Observație:** Se va folosi o funcție auxiliară (INSERT E N L) care returnează mulțimea formată cu liste obținute prin inserarea unui element pe pozițiile N, N-1, N-2, ..., 1 în lista L.

$$(\text{INSERT } '1 2 '(2 3)) = ((2 1 3) (1 2 3))$$

$$\text{insert}(e, n, l_1 l_2 \dots l_k) = \begin{cases} \emptyset & \text{daca } n = 0 \\ \text{ins}(e, n, l_1 l_2 \dots l_k) \oplus \text{insert}(e, n - 1, l_1 l_2 \dots l_k) & \text{altfel} \end{cases}$$

```
(DEFUN INSERT (E N L)
  (COND
    ((= N 0) NIL)
    (T (CONS (INS E N L) (INSERT E (- N 1) L))))
  )
)
```

## Functii LISP (cont). Exemple. Arborescență

Page 7 of 11

$$\text{inserare}(e, l_1 l_2 \dots l_n) = \text{insert}(e, n + 1, l_1 l_2 \dots l_n)$$

```
(DEFUN INSERARE (E L)
  (INSERT E (+ (LENGTH L) 1) L)
  )
```

Temă Continuați implementarea pentru funcția PERMUTĂRI.

**EXEMPLU 2.5.** Se o listă liniară numerică. Să se determine lista sortată, folosind sortarea arborescentă (un ABC intermediu).

$$(\text{sortare } '(5\ 1\ 4\ 6\ 3\ 2)) = (1\ 2\ 3\ 4\ 5\ 6)$$

Pentru sortarea arborescentă a unei liste, vom proceda astfel:

1. Construim un ABC intermediu cu elementele listei inițiale
  - pentru memorarea ABC vom face folosi o listă liniară – varianta V1 (Secțiunea 1)
  - pentru construirea ABC intermediu vom avea nevoie de inserarea unui element în arbore
2. Parcurgerea în ordine a arborelui construit anterior ne va furniza lista inițială ordonată.

Model recursiv – inserarea unui element în ABC

$$\text{inserare}(e, l_1 l_2 l_3) = \begin{cases} (e) & \text{daca } l \neq \emptyset \\ l_1 \oplus \text{inserare}(e, l_2) \oplus l_3 & \text{daca } e \leq l_1 \\ l_1 \oplus l_2 \oplus \text{inserare}(e, l_3) & \text{altfel} \end{cases}$$

```
(defun inserare(e arb)
  (cond
    ((null arb) (list e))
    ((<= e (car arb)) (list (car arb) (inserare e (cadr arb)) (caddr arb)))
    (t (list (car arb) (cadr arb) (inserare e (caddr arb)))))
  ))
```

Model recursiv – construirea ABC din listă

$$\text{construire}(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \neq \emptyset \\ l_1 \oplus \text{construire}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

## Functii LISP (cont). Exemple. Arboi binari

Page 8 of 11

```
(defun construire(l)
  (cond
    ((null l) nil)
    (t (inserare (car l) (construire (cdr l)))))
  )
)

; lista nodurilor unui ABC parcurs în inordine
(defun inordine (arb)
  (cond
    ((null arb) nil)
    (t (append (inordine (cadr arb)) (list (car arb)) (inordine (caddr arb)))))
  )
)

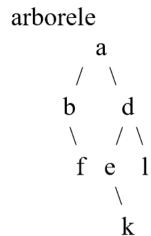
; sortarea arborescentă a listei
(defun sortare(l)
  (inordine (construire l))
)
```

Care este complexitatea timp a sortării arborescente?

Temă Se dă un arbore binar, ale căruia noduri sunt distincte, reprezentat sub forma unei liste neliniare de forma (rădăcină lista-subarbore-stâng lista-subarbore-drept). Să se determine o listă liniară reprezentând calea de la rădăcină către un nod *e* dat.

De exemplu :

- (setq arb '(a (b () (f)) (d (e () (k)) (l))))



(cale 'm arb) → NIL
(cale 'f arb) → (a b f)
(cale 'k arb) → (a d e k)

$$cale(e, l_1 l_2 l_3) = \begin{cases} \emptyset & daca l \neq e \\ (e) & daca e = l_1 \\ l_1 \oplus cale(e, l_2) & daca e \in l_2 \\ l_1 \oplus cale(e, l_3) & daca e \in l_3 \\ \emptyset & altfel \end{cases}$$

### 3. SET, SETQ, SETF

Pentru a da valori simbolurilor în Lisp se folosesc funcțiile cu **efect secundar** SET și SETQ.

Acțiunea prin care o funcție, pe lângă calculul valorii sale, realizează și modificări ale structurilor de date din memorie se numește **efect secundar**.

#### (SET s<sub>1</sub> f<sub>1</sub> ... s<sub>n</sub> f<sub>n</sub>): e

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- efect:
  - valorile argumentelor de rang par (f<sub>i</sub>) devin valorile argumentelor de rang impar corespunzătoare evaluate în prealabil la simboluri (s<sub>i</sub>)
- rezultatul întors valoarea ultimei forme evaluate (f<sub>n</sub>)

Iată câteva exemple

- (SET 'X 'A) = A X se evaluatează la A
- (SET X 'B) = B A se evaluatează la B
- (SET 'X (CONS (SET 'Y X) '(B))) = (A B) X := (A B) și Y := A
- (SET 'X 'A 'L (CDR L)) X := A și L := (CDR L)

#### (SETQ s<sub>1</sub> f<sub>1</sub> ... s<sub>n</sub> f<sub>n</sub>): e

- se evaluatează doar formele f<sub>1</sub>, ..., f<sub>n</sub>
- efect:
  - valorile argumentelor de rang par (f<sub>i</sub>) devin valorile argumentelor de rang impar corespunzătoare neevaluate (s<sub>i</sub>)
- rezultatul întors valoarea ultimei forme evaluate (f<sub>n</sub>)

Iată câteva exemple

- (SETQ X 'A) = A
- (SETQ A '(B C)) = (B C)
- (CDR A) = (C)
- (SETQ X (CONS X A)) = (A B C) X se evaluatează la (A B C)

Lista vidă este singurul caz de listă care are un nume simbolic, NIL, iar NIL este singurul atom care se tratează ca și lista, () și NIL reprezentând același obiect (elementul NIL are în campul CDR un pointer spre el însuși, iar în CAR are NIL).

- (CDR NIL) = NIL
- (CAR NIL) = NIL

**Observație.** Atenție la diferența dintre () = NIL, lista vidă, și () = (NIL), listă ce are ca singur element pe NIL.

- (CONS NIL NIL) = ()

**(SETF f<sub>1</sub> f<sub>2</sub> ... f<sub>n</sub> f'<sub>1</sub> ... f'<sub>n</sub>): e**

- este un macro, are efect destrukturiv
- f<sub>1</sub>, ..., f<sub>n</sub> sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp
- f'<sub>1</sub>, ..., f'<sub>n</sub> sunt forme ale căror valori vor fi legate de locațiile desemnate de parametrii f<sub>1</sub>, ..., f<sub>n</sub> corespunzători.
- efect:
  - se evaluatează formele de rang par (f<sub>i</sub>) și valorile acestora se leagă de locațiile desemnate de formele f<sub>i</sub> corespunzătoare
- rezultatul întors valoarea ultimei forme evaluate (f<sub>n</sub>)

De remarcat că Lisp oferă pe lângă funcțiile SET și SETQ macro-ul SETF. Parametrii p<sub>1</sub>, ..., p<sub>n</sub> sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp, iar e<sub>1</sub>, ..., e<sub>n</sub> sunt forme ale căror valori vor fi legate de locațiile desemnate de parametrii p<sub>1</sub>, ..., p<sub>n</sub> corespunzători. Rezultatul întors de SETF este valoarea ultimei expresii evaluate, e<sub>n</sub>.

Pentru a ne lămuri în legătură cu modul de operare a lui SETF, să vedem următorul exemplu:

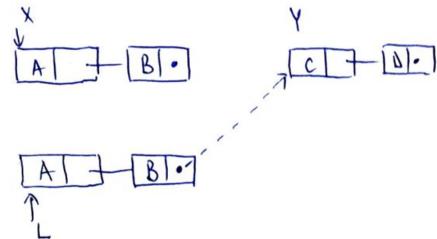
- (SETQ A '(B C D))
  - în acest moment A se evaluatează la (B C D)
- (SETF (CADR A) 'X)
  - efectul este înlocuirea lui (CADR A) cu X
- în acest moment A se evaluatează la (B X D)
  - (SET (CADR A) 'Y)
- efectul este evaluarea lui (CADR A) la X și inițializarea simbolului X la valoarea Y
- în acest moment A se evaluatează la (B X D)
- în acest moment X se evaluatează la Y
  
- (SETQ X '(A B))                    X se evaluatează la (A B)
- (SETQ Y '(A B))                    Y se evaluatează la (A B)
- (SETF (CAR X) 'C)                X se evaluatează la (C B), Y se va evalua tot la (A B)

!!! Atenție la funcția APPEND

- (SETQ X '(A B))                    X se evaluatează la (A B)
- (SETQ Y '(C D))                    Y se evaluatează la (C D)
- (SETQ L (APPEND X Y))            L se evaluatează la (A B C D)

## Funcții LISP (cont). Exemple. Arboi binari

Page 11 of 11



- (SETF (CAR X) 'E)                    X se evaluatează la (E B)  
    ??
- L                                        ??
- (SETF (CADR Y) 'F)                    Y se evaluatează la (C F)  
    ??
- L                                        ??

## CURS 10

### **Expresii LAMBDA. Mecanisme definiționale evolute. Funcții MAP**

#### **Cuprins**

1.	Definirea funcțiilor anonime. Expresii LAMBDA .....	1
1.1	Forma LABELS .....	2
1.2	Utilizarea expresiilor LAMBDA pentru evitarea apelurilor repetitive .....	3
2.	Mecanisme definiționale evolute .....	5
Forme funcționale. Funcțiile APPLY și FUNCALL .....	8	
3.	Functii MAP.....	11

#### **1. Definirea funcțiilor anonime. Expresii LAMBDA**

În situațiile în care

- funcția folosită o singură dată este mult prea simplă ca să merite a fi definită
- funcția de aplicat trebuie sintetizată dinamic (nu este, deci, posibil să fie definită static prin DEFUN)

se poate utiliza o formă funcțională particulară numită **expresie lambda**.

O expresie lambda este o listă de forma

**(LAMBDA l f<sub>1</sub> f<sub>2</sub> ... f<sub>m</sub>)**

ce definește o funcție anonimă utilizabilă doar local, o funcție ce are definiția și apelul concentrate în același punct al programului ce le utilizează, l fiind lista parametrilor iar f<sub>1</sub>...f<sub>m</sub> reprezentând corpul funcției.

Argumentele unei expresii lambda sunt evaluate la apel. Dacă se dorește ca argumentele să nu fie evaluate la apel trebuie folosită forma **QLAMBDA**.

O astfel de formă LAMBDA se folosește în modul ușual:

**((LAMBDA l f<sub>1</sub> f<sub>2</sub> ... f<sub>m</sub>) par<sub>1</sub> par<sub>2</sub> ... par<sub>n</sub>)**

#### **Exemple**

Iată câteva exemple de utilizare a funcțiilor anonime

- ((lambda (l) (cons (car l) (cdr l))) '(1 2 3)) = (1 2 3)
- ((lambda (l1 l2) (append l1 l2)) '(1 2) '(3 4)) = (1 2 3 4)

- Să se definească o funcție care primește ca parametru o listă neliniară și returnează NIL dacă lista are cel puțin un atom numeric la nivel superficial și T, în caz contrar.

```
(defun f(l)
  (cond
    ((null l) t)
    (((lambda (v)
      (cond
        ((numberp v) t)
        (t nil)
        )
      )
      (car l)
      ) nil)
    (t (f (cdr l)))
    )
  )
)
```

## 1.1 Forma LABELS

O formă specială pentru legarea locală a funcțiilor este forma LABELS.

### Exemplu

#### **Ex1.** evaluarea

```
(labels ((fct(l)
            (cdr l)
            )
           )
         (fct '(1 2))
      )
```

va produce (2).

#### **Ex2.**

```
(labels ((temp (n)
            (cond
              ((= n 0) 0)
              (t (+ 2 (temp (- n 1))))))
            )
           )
         (temp 3)
      )
```

va produce 6

**Ex3.** Să se scrie o funcție care primește ca parametru o listă de liste formate din atomi și înțoarce T dacă toate listele conțin atomi numerici și NIL în caz contrar.

```
(test '((1 2) (3 4))) = T
(test '((1 2) (a 4))) = NIL
(test '((1 (2)) (a 4))) = NIL
```

### Solutie

```
(DEFUN TEST (L)
  (COND
    ((NULL L) T)
    ((LABELS ((TEST1 (L)
      (COND
        ((NULL L) T)
        ((NUMBERP (CAR L)) (TEST1 (CDR L)))
        (T NIL)
      )
      )
      )
      (TEST1 (CAR L))
      )
      (TEST (CDR L)))
      (T NIL)
    )
  )
)
```

## 1.2 Utilizarea expresiilor LAMBDA pentru evitarea apelurilor repetitive

**Ex1.** Fie următoarea definiție de funcție

```
(defun g(l)
  (cond
    ((null l) nil)
    (t (cons (car (f l)) (cadr (f l))))
  )
)
```

Soluția pentru a evita apelul **(f l)** este folosirea unei funcții anonte utilizată local, care să poată fi apelată cu parametrul actual **(f l)**.

### Varianta 1

```
(defun g(l)
  (cond
    ((null l) nil)
    (t ((lambda (v)
```

```
(cons (car v) (cadr v))
)
(fl)
)
)
)
```

**Varianta 2**

```
(defun g(l)
  ((lambda (v)
    (cond
      ((null l) nil)
      (t (cons (car v) (cadr v)))
    )
  )
  (fl)
)
)
```

**Ex2.** Să considerăm definiția funcției care generează lista submulțimilor unei mulțimi reprezentate sub formă de listă (a se vedea [Cursul 9, Exemplul 2.3](#)).

```
(defun subm(l)
  (cond
    ((null l) (list nil))
    (t (append (subm (cdr l)) (insPrimaPoz (car l) (subm (cdr l))))))
  )
)
```

După cum se observă, apelul `(subm (cdr l))` este repetat. Pentru a evita apelul repetat, se va folosi o expresie LAMBDA.

O posibilă soluție este următoarea:

```
(defun subm(l)
  (cond
    ((null l) (list nil))
    (t ((lambda (s)
      (append s (insPrimaPoz (car l) s)))
    )
    (subm (cdr l))
    )
  )
)
```

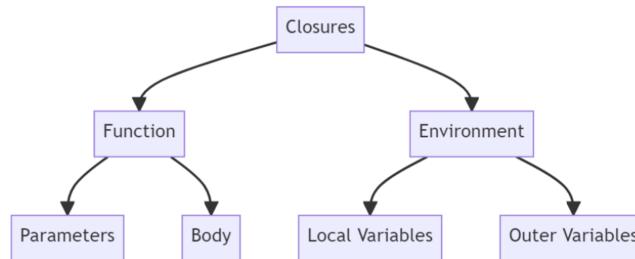
## 2. Mecanisme definiționale evolute

### Closure (închidere)

- *Lexical closure*
- <https://dept-info.labri.fr/~strandh/Teaching/MTP/Common/David-Lamkins/chapter15.html>
- [closure](#)

A closure is a function together with the environment in which it was created. The principal idea behind closures is that in Common Lisp, functions are “first-class objects,” that is, you can write code that takes a function as an argument, and you can write code that returns a function as one of its values.

- combinație între o funcție și mediul lexical în care aceasta e definită
  - o închidere dă acces la domeniul de vizibilitate al unei funcții exterioare dintr-o funcție interioară



- o **închidere** este o funcție care acces la variabilele care sunt în afara domeniului lor de vizibilitate lexical, chiar după ce aceasta și-a încheiat execuția
  - astfel funcția își “memorează” mediul în care a fost creată
- expresiile Lambda în Lisp reprezintă **închideri**
- conceptul provine din programarea funcțională, dar apare și în programarea imperativă
  - [C++ 11](#) - funcțiile lambda construiesc o închidere
  - Javascript, Python

### Exemple Python

1)

```

def greet():
    # variable defined outside the inner function
    name = "John"

    # return a nested anonymous function
    return lambda: "Hi " + name

# call the outer function
message = greet()

# call the inner function
print(message())
  
```

```
2)
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply
|
# Usage
double = multiplier(2)
triple = multiplier(3)

print(double(5)) # Output: 10
print(triple(5)) # Output: 15
```

**Clojure**

- <https://clojure.org/>
- dialect al limbajului Lisp pe platforme Java
- multi-paradigmă, orientat agent, concurrent, funcțional, logic

**Closure Common Lisp (CCL)**

- implementare Common Lisp
  - fire de execuție, compilare rapidă, mecanism pentru apelul funcțiilor externe (mecanism de *callback* – funcții Lisp apelate din cod extern)

Fie următoarele definiții și evaluări

```
> (defun f() 10)
F
> (setq f '11)
11
> (f)
10
> f
11
> (function f)
#<FUNCTION F NIL (DECLARE (SYSTEM::IN-DEFUN F)) (BLOCK F 10)>
> (setq g 7)
7
> (function g)
undefined function G
> (quote g) //echivalent cu 'g
G
> (function car)
#<SYSTEM-FUNCTION CAR>
> (function (lambda (l) (cdr l)))
#<FUNCTION LAMBDA (L) (CDR L)>
```

**De remarcat faptul că AND și OR nu sunt considerate funcții, ci operatori speciali.**

```
> (function not)
#<SYSTEM-FUNCTION NOT>
> (function and)
undefined function AND
> (function or)
undefined function OR
```

!!! Din punct de vedere semantic, standardul CommonLisp impune să se indice dacă e vorba de o funcție sau un simbol.

Argument		
Funcție f	#'f	(function f)
Simbol x	'x	(quote x)

	Funcție f
Standard	#'f
CLisp	'f
GCLisp, Emacs Lisp, alte dialecte	'f

	Expresie Lambda
Standard	#'(lambda ....)
CLisp	(lambda ...)
GCLisp, Emacs Lisp, alte dialecte	'(lambda ....)

## Forma EVAL

Aplicarea formei EVAL este echivalentă cu apelul evaluatorului Lisp. Sintaxa funcției este

**(EVAL f) : e**

Efectul constă în evaluarea formei și returnarea rezultatului evaluării. Forma este evaluată de fapt în două etape: mai întâi este evaluată ca argument al lui EVAL iar apoi rezultatul acestei evaluări este din nou evaluat ca efect al aplicării funcției EVAL. De exemplu:

- (SETQ X '((CAR Y) (CDR Y) (CADR Y)))
- (SETQ Y '(A B C))
- (CAR X) se evaluatează la (CAR Y)
- (EVAL (CAR X)) va produce A

Mecanismul este asemănător cu ceea ce înseamnă indirectarea prin intermediul pointerilor din cadrul limbajelor imperitive.

- (SETQ L '(1 2 3))
- (SETQ P '(CAR L))
- P se evaluatează la (CAR L)
- (EVAL P) va produce 1

- (SETQ B 'X)
- (SETQ A 'B)
- (EVAL A) se evaluează la X
- (SETQ L '(+ 1 2 3))
- L se evaluează la (+ 1 2 3)
- (EVAL L) se evaluează la 6

**Observație.** Lisp nu evaluează primul element dintr-o formă, ci numai îl aplică.

Ex: Asocierea (SETQ Q 'CAR) nu permite apelul sub forma (Q '(A B C)), un astfel de apel semnalând eroare în sensul că evaluatorul LISP nu găsește nici o funcție cu numele Q. Pe de altă parte, nici numai cu ajutorul functiei EVAL nu putem rezolva problema:

- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (EVAL P) va produce CAR
- ((EVAL P) '(A B C)) va produce mesaj de eroare: “Bad function when ...”

Mesajul de eroare de mai sus apare deoarece Lisp nu-și evaluează primul argument dintr-o formă.

**!!! O listă este totdeauna evaluată dacă acest lucru nu este opriți explicit (prin QUOTE), în schimb primul argument al oricărei liste nu este niciodată evaluat!**

**Exemplu** Fie lista L care pe prima poziție are un operator binar (+, -, \*, /), iar pe următoarele 2 poziții are 2 operanzi (numerici). De exemplu, L = (\* 2 3 - 4 1 ...). Se cere să returneze rezultatul aplicării operatorului (1-ul element al listei) asupra celor doi operanzi care urmează în listă. În exemplu nostru, ar trebui să se returneze valoarea 6.

Soluția este simplă: (EVAL (LIST (CAR L) (CADR L) (CADDR L))) = 6

## Forme funcționale. Funcțiile APPLY și FUNCALL

Există situații în care forma funcției nu se cunoaște, expresia ei trebuind să fie determinată dinamic. Ar trebui să avem ceva de genul

**(EXPR\_FUNC p<sub>1</sub> ... p<sub>n</sub>)**

Deoarece EXPR\_FUNC trebuie să genereze în cele din urmă o funcție ea este o așa-numită **formă funcțională**. Forma EXPR\_FUNC este evaluată până ce se obține o funcție sau, în general, o expresie ce poate fi aplicată parametrilor.

Într-o astfel de situație, evaluarea parametrilor este amânată până în momentul reducerii formei funcționale EXPR\_FUNC la funcția propriu-zisă F. Parametrii vor fi evaluați doar dacă F își evaluează, în prealabil, parametrii. Deci evaluarea formei de mai sus parcurge etapele:

- (i) reducerea formei EXPR\_FUNC la F (eventual o expresie LAMBDA sau macrodefiniție) și substituția lui EXPR\_FUNC prin F în forma de evaluat;
- (ii) evaluarea formei (F p<sub>1</sub> ... p<sub>n</sub>).

Există însă situații în care și numărul parametrilor trebuie stabilit dinamic, deci funcția determinată dinamic trebuie să accepte un număr variabil de parametri. Este nevoie deci de o modalitate de a permite aplicarea unei funcții asupra unei mulțimi de parametri sintetizată eventual dinamic. Acest lucru este oferit de funcțiile APPLY și FUNCALL.

#### **(APPLY ff lp):e**

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- Funcția APPLY permite aplicarea unei funcții asupra unor parametri furnizați sub formă de listă. În descrierea de mai sus, **ff** este o formă funcțională și **lp** este o formă reductibilă prin evaluare la o listă de parametri efectivi (p<sub>1</sub> p<sub>2</sub> ... p<sub>n</sub>).

#### **EXAMPLE**

- (APPLY #'CONS '(A B)) va produce (A . B)
- (APPLY (FUNCTION CONS) '(A B)) va produce (A . B)
- (APPLY #'MAX '(1 2 3)) va produce 3
- (APPLY #'+ '(1 2 3)) va produce 6
- (DEFUN F(L) (CDR L))
- (APPLY #'F '((1 2 3))) va produce (2 3)
- (APPLY #'(LAMBDA (L) (CAR L)) '((A B C))) va produce A
- (SETQ P 'CAR)
- (APPLY P '((A B C))) va produce A
- (APPLY #'P '((A B C))) va produce eroare **undefined function P**
- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (APPLY (EVAL P) '((1 2 3))) va produce 1

#### **(FUNCALL ff l1 l2 ...ln):e**

- se evaluatează argumentul și apoi se trece la evaluarea funcției
- FUNCALL este o variantă a funcției APPLY care permite aplicarea unei funcții (sau expresii) rezultate prin evaluarea unei forme funcționale **ff** asupra unui număr fix de parametri.

#### **EXAMPLE**

- (FUNCALL #'CONS 'A 'B) va produce (A . B)

- (FUNCALL (FUNCTION CONS) 'A 'B) va produce (A . B)
- (FUNCALL #'MAX '1 '2 '3)) va produce 3
- (FUNCALL #'+ '1 '2 '3)) va produce 6
- (DEFUN F(L) (CDR L))
- (FUNCALL #'F '(1 2 3)) va produce (2 3)
- (FUNCALL #'(LAMBDA (L) (CAR L)) '(A B C)) va produce A
- (SETQ P 'CAR)
- (FUNCALL P '(A B C)) va produce A
- (FUNCALL #'P '(A B C)) va produce eroare undefined function P
- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (FUNCALL (EVAL P) '(1 2 3)) va produce 1

!!! Atenție la folosirea AND și OR, deoarece nu sunt funcții.

- (APPLY #'AND '((T NIL))) va produce eroare undefined function AND
- (APPLY #'OR '((T NIL))) va produce eroare undefined function OR
- (FUNCALL #'AND '(T NIL)) va produce eroare undefined function AND
- (FUNCALL #'OR '(T NIL)) va produce eroare undefined function OR

Soluția este definirea unei funcții al cărei efect să fie aplicarea AND/OR pe elementele unei liste cu valori logice T, NIL.

$$\text{SI}(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } l_1 \text{ e fals} \\ \text{SI}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun SI(l)
  (cond
    ((null l) t)
    ;(t (and (car l) (SI (cdr l))))
    ((not (car l)) nil)
    (t (SI (cdr l)))
  )
)
```

- (FUNCALL #'SI '(T NIL T)) va produce NIL
- (SI '(T T T)) va produce T

**Exemple****Ex 1**

- (DEFUN F()
 #'(LAMBDA (x) (CAR x))
 )
- (FUNCALL (F) '(1 2 3)) → ???
- (APPLY (F) '((1 2 3))) → ???

**Ex 2**

- (FUNCALL (FUNCTION (LAMBDA (x) (cadr x))) '(1 2 3)) → ???

**Exemplificare Closure Common Lisp****Ex1**

```
(defun increment (x)
  #'(lambda (y)
    (+ x y)
  )
)

(setq inc5 (increment 5))
; returnează o nouă funcție (închidere) care adaugă 5 la argumentul său

(print (funcall inc5 3)) ; va afișa 8
```

**Ex2**

```
(defun two-funs (x)
  (list
    (function (lambda () x))
    (function (lambda (y) (setq x y)))
  )
)

(setq F (two-funs 6))

(funcall (car F)) => 6
(funcall (cadr F) 43) => 43
(funcall (car F)) => 43

(setq G (two-funs 5))
(funcall (car G)) => 5
(funcall (cadr G) 13) => 13
(funcall (car G)) => 13

(funcall (car F)) => 43
```

**3. Funcții MAP**

Rolul funcțiilor MAP este de a aplica o funcție în mod repetat asupra elementelor (sau sublistelor succesive) listelor date ca argumente.

**(MAPCAR f l<sub>1</sub> ... l<sub>n</sub>) : 1**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
  - CAR-ului listelor => e<sub>1</sub>
  - CADR-ului listelor => e<sub>2</sub>
  - ....până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

Iată câteva exemple de aplicare ale funcției MAPCAR:

- (MAPCAR #'CAR '((A B C) (X Y Z))) se evaluatează la (A X)
- (MAPCAR #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluatează la (NIL T T)
- (MAPCAR #'LIST '(A B C)) se evaluatează la ((A) (B) (C))
- (MAPCAR #'LIST '(A B C) '(1 2)) se evaluatează la ((A 1) (B 2))
- (MAPCAR #'+'(1 2 3) '(4 5 6)) se evaluatează la (5 7 9)

Aplicarea funcției LIST este posibilă indiferent de numărul listelor argument deoarece LIST este o funcție cu număr variabil de argumente.

**Observatie.** Dacă F este o funcție unară, care se aplică unei liste L=(l<sub>1</sub> l<sub>2</sub> ... l<sub>n</sub>), atunci (MAPCAR #'F L) va produce lista (F(l<sub>1</sub>), F(l<sub>2</sub>), ..., F(l<sub>n</sub>))

**Exemple**

1. Să se definească o funcție MODIF care să modifice o listă dată ca parametru astfel: atomii nenumerici rămân nemodificați iar cei numerici își dublează valoarea; modificarea trebuie făcută la toate nivelurile.

(MODIF '(1 (b (4) c) (d (3 (5 f))))) va produce (2 (b (8) c) (d (6 (10 f))))

**Model recursiv**

$$\text{MODIF}(l) = \begin{cases} 2l & \text{dacă } l \text{ număr} \\ l & \text{dacă } l \text{ atom} \\ \bigcup_{i=1}^n \text{MODIF}(l_i) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```
(DEFUN MODIF (L)
  (COND
    ((NUMBERP L) (* 2 L)) ; determină operația asupra atomilor numerici
```

((ATOM L) L) ; determină operația asupra atomilor nenumerici  
 (T (MAPCAR #'MODIF L)) ; reprezintă strategia de parcurgere

)  
 )

2. Să se construiască o funcție LGM ce determină lungimea (calculată în număr de elemente la nivel superficial) celei mai lungi subliste dintr-o listă dată L (dacă lista este formată numai din atomi atunci lungimea cerută este chiar cea a listei L).

(LGM '(1 (2 (3 4) (5 (6)) (7)))) va produce 4

Descrierea algoritmului se poate exprima astfel:

- a). valoarea LGM este maximul dintre lungimea listei L și maximul valorilor de aceeași natură calculate prin aplicarea lui LGM pentru fiecare element al listei L în parte;  
 b). LGM(atom) = 0.

#### Model recursiv

$$\text{LGM}(l) = \begin{cases} 0 & \text{dacă } l \text{ e atom} \\ \max(n, \max(\text{LGM}(l_1), \text{LGM}(l_2), \dots, \text{LGM}(l_n))) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

(DEFUN LGM(L)  
 (COND  
 ((ATOM L) 0)  
 (T (MAX (LENGTH L) (APPLY #'MAX (MAPCAR #'LGM L)) ))  
 )  
 )

Aplicarea funcțiilor ATOM și LENGTH calculează de fapt lungimile, (MAPCAR #'LGM L) realizând de fapt parcurgerea integrală a listei. Apelul (MAPCAR 'LGM L) furnizează o listă de lungimi. Deoarece trebuie să obținem maximul acestora, va trebui să aplicăm funcția MAX pe elementele acestei liste. Pentru aceasta folosim APPLY.

#### **(MAPCAN f l<sub>1</sub> ... l<sub>n</sub>) : 1**

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară f este aplicată pe rând asupra:
  - CAR-ului listelor => e<sub>1</sub>
  - CADR-ului listelor => e<sub>2</sub>
  - CADDR-ului listelor => e<sub>3</sub>
  - ....până când una din liste ajunge vidă
- Rezultatele sunt grupate cu NCONC într-o listă ce este returnată rezultat

**(NCONC l1 l2 ... ln) : l**

Relativ la modificarea sau nu a structurii listelor implicate, concatenarea de liste se poate efectua în două maniere: cu modificarea listelor (folosind funcția NCONC) și fară (folosind funcția APPEND)

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- efect: **NCONC** realizează concatenarea efectivă (fizică) prin modificarea ultimului pointer (cu valoarea NIL) al primelor n-1 argumente și întoarce primul argument, care le va îngloba la ieșire pe toate celelalte.
- (SETQ L1 '(A B C) L2 '(D E)) se evaluatează la (D E)
- (APPEND L1 L2) se evaluatează la (A B C D E)
- L1 se evaluatează la (A B C)
- L2 se evaluatează la (D E)
- (SETQ L3 (NCONC L1 L2)) se evaluatează la (A B C D E)
- L1 se evaluatează la (A B C D E)
- (SETQ L1 '(A) L2 '(B) L3 '(C)) se evaluatează la (C)
- L1 se evaluatează la (A)
- L2 se evaluatează la (B)
- L3 se evaluatează la (C)
- (NCONC L1 L2 L3) se evaluatează la (A B C)
- L1 se evaluatează la (A B C)
- L2 se evaluatează la (B C)
- L3 se evaluatează la (C)

**Exemple MAPCAN**

- (MAPCAN #'CAR '((A B C) (X Y Z))) se evaluatează la NIL, deoarece NCONC cere liste, și ca atare (NCONC 'A 'X) este NIL
- (MAPCAN #'LIST '(A B C) '(1 2)) se evaluatează la (A 1 B 2)
- (MAPCAN #'LIST '(A B C)) se evaluatează la ( A B C )
- (MAPCAN #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluatează la NIL
- (MAPCAN #'+ '(1 2 3) '(4 5 6)) se evaluatează la NIL

**Observatie (MAPCAN vs. MAPCAR)**

Fie următoarele definiții

```
(defun F(L)
  (cdr L)
)

(setq L '((1 2 3) (4 5 6) (7 8)))
  → ((1 2 3) (4 5 6) (7 8))

(mapcar #'F L)
  → ((2 3) (5 6) (8 9))
```

(mapcan #'F L) → (2 3 5 6 8 9)

⇒ (apply #'append (mapcar #'F L)) ≡ (mapcan #'F L)

!!! Din cauza efectului destructiv al lui NCONC aplicarea MAPCAN poate avea efecte secundare și e de recomandat să se folosească varianta echivalentă cu MAPCAR

#### (MAPLIST f l<sub>1</sub> ... l<sub>n</sub>) : l

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară f este aplicată pe rând asupra:
  - listelor => e<sub>1</sub>
  - CDR-ului listelor => e<sub>2</sub>
  - CDDR-ului listelor => e<sub>3</sub>
  - ....până când una din liste ajunge vidă
- Rezultatele sunt grupate cu LIST într-o listă ce e returnată rezultat

#### Exemple MAPLIST

- (MAPLIST #'APPEND '(A B C) '(1 2 3)) furnizează ((A B C 1 2 3) (B C 2 3) (C 3))
- (MAPLIST #'(LAMBDA (X) X) '(A B C)) furnizează ((A B C) (B C) (C))
- (SETF TEMP '(1 2 7 4 6 5)) urmat de  
 (MAPLIST #'(LAMBDA (XL YL) (<(CAR XL)(CAR YL)))  
 TEMP (CDR TEMP))  
 ) va furniza lista (T T NIL T NIL)

Comparativ cu exemplele date la MAPCAR, aici vom obține:

- (MAPLIST #'CAR '((A B C) (X Y Z))) se evaluatează la ((A B C) (X Y Z))
- (MAPLIST #'LIST '(A B C) '(1 2)) se evaluatează la ( ((A B C) (1 2)) ((B C) (2)) )
- (MAPLIST #'LIST '(A B C)) se evaluatează la ( ((A B C)) ((B C)) ((C)) )
- (MAPLIST #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluatează la (NIL NIL NIL)
- (MAPLIST #'+ '(1 2 3) '(4 5 6)) va produce mesajul de eroare: “argument to + should be a number: (1 2 3)”.

#### (MAPCON f l<sub>1</sub> ... l<sub>n</sub>) : l

- se evaluatează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară f este aplicată pe rând asupra:
  - listelor => e<sub>1</sub>
  - CDR-ului listelor => e<sub>2</sub>
  - CDDR-ului listelor => e<sub>3</sub>
  - ....până când una din liste ajunge vidă
- Rezultatele sunt grupate cu NCONC într-o listă ce e returnată rezultat

Exemple MAPCON

- (MAPCON #'CAR '((A B C) (X Y Z))) furnizează (A B C X Y Z)
  - (MAPCON #'LIST '(A B C) '(1 2)) furnizează ((A B C) (1 2) (B C) (2))
  - (MAPCON #'LIST '(A B C)) furnizează ((A B C) (B C) (C))
  - (MAPCON #'EQUAL '(A (B C) D) '(Q (B C) D X)) furnizează NIL
  - (MAPCON #'+ '(1 2 3) '(4 5 6)) furnizează mesaj de eroare: : “argument to + should be a number: (1 2 3)”
  - (DEFUN G(L)  
    (MAPCON #'LIST L)  
)
- (G '(1 2 3)) = ((1 2 3) (2 3) (3))
- (MAPCON #'(LAMBDA (L) (MAPCON #'LIST L)) '(1 2 3)) furnizează ((1 2 3) (2 3) (3) (2 3) (3) (3))

## CURS 11

### Exemple funcții MAP (cont).

**EXEMPLU 2.1** Fie următoarele definiții

```
(defun f(L e)
      (list e L)
)
(setq L '(1 2 3))
(setq e 4)

(mapcar #'f L e) se valuează la NIL
(mapcar #'(lambda (L) (f L e)) L) se valuează la ((4 1) (4 2) (4 3))
```

**EXEMPLU 2.2** Fie următoarea definiție de funcție

```
(defun f(L)
      (list L)
)
(mapcar #'f '(1 2 3)) se valuează la ((1) (2) (3))
(mapcan #'f '(1 2 3)) se valuează la (1 2 3)
```

De remarcat echivalența între

(mapcan #'f L) și (apply #'append (mapcar #'f L))

**EXEMPLU 2.3** Să se definească o funcție care să returneze lungimea unei liste neliniare (în număr de atomi la orice nivel)

$(LG'(1 (2 (a ) c d) (3))) = 6$

$$LG(L) = \begin{cases} 1 & \text{daca } L \text{ e atom} \\ \sum_{i=1}^n LG(L_i) & \text{daca } L \text{ e lista } (L_1 \dots L_n) \end{cases}$$

```
(DEFUN LG (L)
  (COND
    ((ATOM L) 1)
    (T (APPLY #'+ (MAPCAR #'LG L)))
  )
)
```

**EXEMPLU 2.4** Să se definească o funcție care având ca parametru o listă neliniară să returneze numărul de subliste (inclusiv lista) având lungime număr par (la nivel superficial).

$$(nr '(1 (2 (3 (4 5) 6)) (7 (8 9)))) = 4$$

Vom folosi o funcție auxiliară care returnează T dacă lista argument are număr par de elemente la nivel superficial, NIL în caz contrar.

$$nr(L) = \begin{cases} 0 & \text{daca } L \text{ e atom} \\ 1 + \sum_{i=1}^n lg(L_i) & \text{daca } L \text{ e lista } (L_1 \dots L_n) \text{ si } n \text{ e par} \\ \sum_{i=1}^n lg(L_i) & \text{altfel} \end{cases}$$

```
(DEFUN PAR (L)
  (COND
    ((= 0 (MOD (LENGTH L) 2)) T)
    (T NIL)
  )
)

(DEFUN nr (L)
  (COND
    ((ATOM L) 0)
    ((PAR L) (+ 1 (APPLY #'+ (MAPCAR #'nr L))))
    (T (APPLY #'+ (MAPCAR #'nr L)))
  )
)
```

**EXEMPLU 2.5** Să se definească o funcție care având ca parametru o listă neliniară să returneze lista atomilor (de la orice nivel) din listă.

$$(atomi '(1 (2 (3 (4 5) 6)) (7 (8 9)))) = (1 2 3 4 5 6 7 8 9)$$

$$atomi(L) = \bigcup_{i=1}^n atomi(L_i) \quad \text{daca } L \text{ e lista } (L_1 \dots L_n)$$

```
(DEFUN atomi (L)
  (COND
    ((ATOM L) (LIST L))
    (T (MAPCAN #'atomi L)))
  )
)
```

**Observație:** Varianta echivalentă folosind funcția MAPCAR

```
(DEFUN atomi (L)
  (COND
    ((ATOM L) (LIST L))
    (T (APPLY #'APPEND (MAPCAR #'atom i L)))
  )
)
```

**EXEMPLU 2.6** Să se definească o funcție care determină numărul de apariții, de la orice nivel, ale unui element într-o listă neliniară.

```
(nrap 'a '(1 (a (3 (4 a) a)) (7 (a 9)))) = 4
```

$$nrap(e, l) = \begin{cases} 1 & \text{dacă } l = e \\ 0 & \text{dacă } l \text{ e atom} \\ \sum_{i=1}^n nrap(e, l_i) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```
(defun nrap(e L)
  (cond
    ((equal L e) 1)
    ((atom L) 0)
    (t (apply #'+ (mapcar #'(lambda(L)
      (nrap e L)
    )
    L
  )
  )
  )
  )
)
```

**EXEMPLU 2.7** Se dă o listă neliniară. Se cere să se returneze lista din care au fost șterși atomii numerici negativi. Se va folosi o funcție MAP.

Ex: (stergere '(a 2 (b -4 (c -6)) -1)) → (a 2 (b (c)))

$$ster g(l) = \begin{cases} \emptyset & \text{dacă } l \text{ numeric negativ} \\ l & \text{dacă } l \text{ e atom} \\ \bigcup_{i=1}^n ster g(l_i) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```
(defun sterg(L)
  (cond
    ((and (numberp L) (minusp L)) nil)
    ((atom L) (list L))
    (t (list (apply #'append
                     (mapcar #'sterg L)
                     )
                     )
                     )
    )
  )

(defun stergere(L)
  (car (sterg L))
)
```

; (sterg '((a) (c -2))) → ((a) (c))  
 ; (sterg '(a)) → (a)  
 ; (sterg '(c -2)) → (c)  
 ; (append '(a) '(c)) → (a c)  
 ; (append '((a)) '((c))) → ((a) (c))

**EXEMPLU 2.8** Se dă un arbore n-ar nevid, reprezentat sub forma unei liste neliniare de forma (rădăcina lista\_sub1.....lista\_sub\_n) (V1 de reprezentare a arborilor binari, Curs 9). Se cere să se determine numărul de noduri din arbore.

$(nrNoduri '(a (b (c) (d (e))) (f (g))))$  va produce 7

#### Model recursiv

$$nrNoduri(l_1 l_2 \dots l_n) = \begin{cases} 1 & \text{daca } n = 1 \\ 1 + \sum_{i=2}^n nrNoduri(l_i) & \text{altfel} \end{cases}$$

```
(defun nrNoduri(L)
  (cond
    ((null (cdr L)) 1)
    (t (+ 1 (apply #'+
                     (mapcar #'nrNoduri (cdr L))
                     )
                     )
    )
  )
)
```

**EXEMPLU 2.9** Se dă un arbore n-ar nevid, reprezentat sub forma unei liste neliniare de forma (rădăcina lista\_sub1.....lista\_sub\_n) (V1 de reprezentare a arborilor binari, Curs 9). Se cere să se determine adâncimea arborelui. Observație. Nivelul rădăcinii este 0.

$(adancime '(a (b (c) (d (e))) (f (g))))$  va produce 3

Model recursiv

$$adancime(l_1 l_2 \dots l_n)$$

$$= \begin{cases} 0 & \text{daca } n = 1 \\ 1 + \max(adancime(l_2), adancime(l_3), \dots, adancime(l_n)) & \text{altfel} \end{cases}$$

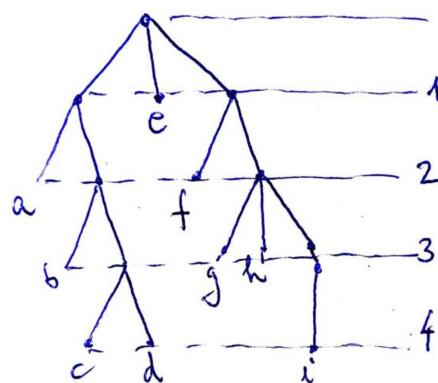
```
(defun adancime(L)
  (cond
    ((null (cdr L)) 0)
    (t (+ 1 (apply #'max
                    (mapcar #'adancime (cdr L)))))))
```

**EXEMPLU 2.10** Să se determine lista atomilor de adâncime  $n$  dintr-o listă neliniară (nivelul superficial al listei se consideră 1).

(lista '((a (b (c d))) e (f (g h (i)))) 3) va produce (b g h)

(lista '((a (b (c d))) e (f (g h (i)))) 4) va produce (c d i)

(lista '((a (b (c d))) e (f (g h (i)))) 5) va produce NIL



## Functii MAP

Page 6 of 10

### Model recursiv

$$lista(l, n) = \begin{cases} (l) & dacă n = 0 și l atom \\ \emptyset & dacă n = 0 \\ \emptyset & dacă l atom \\ \bigcup_{i=1}^k lista(l_i, n - 1) & altfel, l = (l_1 l_2 \dots l_k) și lista \end{cases}$$

```
(defun lista(L n)
  (cond
    ((and (= n 0) (atom L)) (list L))
    ((= n 0) nil)
    ((atom L) nil)
    (t (mapcan #'(lambda(L)
                    (lista L (- n 1)))
                  L)
        )
      )
    )
  )
```

!!! Clauza 4 din COND poate fi rescrisa cu MAPCAR

```
(t (apply #'append (mapcar #'(lambda(L)
                                (lista L (- n 1)))
                               L)
                  )
    )
  )
```

**EXEMPLU 2.11** Se dă o listă de liste. Se cere să se determine ....

```
(defun m (L)
  (cond
    ((numberp L) L)
    ((atom L) most-negative-fixnum)
    (t (apply #'max
              (mapcar #'m L)
              )
      )
    )
  )
(defun lista (L)
  (mapcan #'(lambda (L)
               ((lambda (v)
                   (cond
                     ((= 0 (mod v 2)) (list v))
```

```
(t nil)
      )
    ) (m L)
      )
    )
  L
  )
)
```

(lista '((5 a (2 b (8))) (7 a (9)) (c d (10)))) → ???

**EXEMPLU 2.12** Se dă o listă liniară. Se cere....

```
(defun p (L)
  (mapcan #'(lambda (e1)
    (mapcar #'(lambda (e2)
      (list e1 e2)
      )
    L
    )
  L
  )
)
```

(p '(1 2 3)) → ((1 1) (1 2) (1 3) (2 1) (2 2) (2 3) (3 1) (3 2) (3 3))

**EXEMPLU 2.13** O matrice se poate reprezinta sub forma unei liste formate din liste conținând elementele de pe linii, în ordine de la prima la ultima linie. De exemplu, lista ((1 2) (3 4)) corespunde matricei

```
1 2
3 4
```

Fie L o listă asociată unei matrici cu elemente numerice. Care este rezultatul returnat de funcția *fct*?

```
(defun fct (L)
  (cond
    ((null (car L)) nil)
    (t (cons
      (mapcar #'car L)
      (fct (mapcar #'cdr L))
      )
    )
  )
)
```

$(fct '((1 2) (4 5) (7 8))) \rightarrow ((1 4 7) (2 5 8))$

**EXEMPLU 2.14** Să se definească o funcție care având ca parametru o listă neliniară returnează lista liniară a atomilor care apar pe orice nivel, dar în ordine inversă.

$(INVERSARE '(A (B C (D (E))) (F G))) = (G F E D C B A)$

```
(DEFUN INVERSARE (L)
  (COND
    ((ATOM L) (LIST L))
    (T (APPLY #'APPEND (MAPCAR #'INVERSARE (INVERS L)))))
  )
)
```

Unde INVERS este funcția care returnează o listă inversată la nivel superficial (Curs 8, Exemplu 4.3) – echivalent cu apelul funcției REVERSE (funcție predefinită).

**EXEMPLU 2.15** O matrice se poate reprezenta în Lisp sub forma unei liste ale cărei elemente sunt liste reprezentând liniile matricei.

$(\text{linia1}) (\text{linia2}) \dots$

Să se definească o funcție care având ca parametri două matrice de ordin **n** returnează (sub formă de matrice) produsul acestora.

$(PRODUS '((1 2) (3 4)) '((2 -1) (3 1))) = ((8 1) (18 1))$

**Observație:** Vom folosi două funcții ajutătoare: o funcție (COLOANE L) care returnează lista coloanelor matricei parametru L și o funcție (PR L1 L2) care returnează sub formă de matrice rezultatul înmulțirii matricei L1 (listă de linii) cu lista L2 (o listă de coloane ale unei matrice).

```
(DEFUN COLOANE (L)
  (COND
    ((NULL (CAR L)) NIL)
    (T (CONS (MAPCAR #'CAR L) (COLOANE (MAPCAR #'CDR L))))
  )
)
```

```
(DEFUN PR (L1 L2)
  (COND
    ((NULL (CAR L1)) NIL)
    (T (CONS (MAPCAR #'(LAMBDA (L)
                                (APPLY #'+ (MAPCAR #'* (CAR L1) L)))
                           )
             )
      ))
```

```

        )
      )
    )
  )
)
(DEFUN PRODUS (L1 L2)
  (PR L1 (COLOANE L2))
)

```

**EXEMPLU 2.16** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se genereze lista submulțimilor multimii. Se va folosi o funcție MAP.

Ex:  $(subm\ '(1\ 2)) \rightarrow (\text{nil}\ (1)\ (2)\ (1\ 2))$

## Observatie

(setq e 1)

(mapcar #'(lambda(L) (cons e L)) '((2 3) (3 2))) va produce ((1 2 3) (1 3 2))

```
(defun subm (L)
  (cond
    ((null L) (list nil))
    (t ((lambda (s)
           (append s (mapcar #'(lambda (sb)
                                 (cons (car L) sb)
                               )
                           s
                         )
                     )
           )
            (subm (cdr L))
          )
         )
      )
    )
  )
}
```

**EXEMPLU 2.17** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se genereze lista permutărilor multimiei. Se va folosi o funcție MAP.

Ex:  $(permutari '(1 2 3)) \rightarrow ((1\ 2\ 3)\ (1\ 3\ 2)\ (2\ 1\ 3)\ (2\ 3\ 1)\ (3\ 1\ 2)\ (3\ 2\ 1))$

```
(defun permutari (L)
  (cond
    ((null (cdr L)) (list L))
    (t (mapcan #'(lambda (e)
```

```
(mapcar #'(lambda (p)
  (cons e p)
  )
  (permutari (remove e L))
)
)
)
)
)
)
```

unde REMOVE este funcția care returnează lista din care s-a șters un element de la nivel superficial.

## Alte aspecte ale programării funcționale

### Cuprins

1.	Parametrii Lisp în context “Dynamic Scoping” .....	1
2.	Universul expresiilor și universul instrucțiunilor .....	3
3.	Independența ordinii de evaluare .....	4

### 1. Parametrii Lisp în context “Dynamic Scoping”

Așa cum am văzut până acum, toate aparițiile parametrilor formali sunt înlocuite cu valorile argumentelor corespunzătoare. Deci, avem o variantă de apel prin nume, apelul prin text, deoarece în Lisp avem de-a face cu determinarea dinamică a domeniului de vizibilitate (dynamic scoping) simultan cu înlocuirea textuală a valorilor evaluate.

Deși parametrul se leagă de argument (în sensul apelului prin referință din limbajele imperative, adică numele argumentului și cel al parametrului devin sinonime), nu este apel prin referință, deoarece modificările valorilor parametrilor formali nu afectează valoarea argumentelor (ceea ce amintește de apelul prin valoare).

De exemplu:

- (DEFUN FCT(X) (SETQ X 1) Y) se evaluează la FCT
- (SETQ Y 0) se evaluează la 0
- (FCT Y) se evaluează la 0

In secvența de mai sus Y se evaluează la 0, iar X se leaga la 0. Apelul FCT este astfel echivalent cu (FCT 0). X e parametru formal, Y e parametru actual, și ca atare modificarea lui X nu afectează pe Y, deci NU avem de-a face cu apel prin referință.

Aceasta demonstrează că transmiterea de parametri în LISP nu se face prin referință. Cum justificăm însă că nu este nici apel prin valoare? Există totuși situații în care valoarea parametrului actual poate fi modificată, de exemplu prin acțiunea funcțiilor cu efect distructiv RPLACA și RPLACD.

Să mai remarcăm faptul că în ciuda caracteristicii “Dynamic scoping”, variabila Y nu se leagă la execuție de parametrul formal tocmai înlocuit cu Y, adică Y-ul “intern” funcției, ci își păstrează caracteristica de variabilă globală pentru FCT.

Corpul unei funcții LISP este domeniul de vizibilitate a parametrilor săi formali, care se numesc variabile legate în raport cu funcția respectivă. Celelalte variabile ce apar în definitia funcției se numesc variabile libere.

Contextul curent al execuției este alcătuit din toate variabilele din program împreună cu valorile la care sunt legate acestea în acel moment. Acest concept este necesar pentru stabilirea valorii variabilelor libere care intervin în evaluarea unei funcții, evaluarea în LISP făcându-se într-

un context dinamic (dynamic scoping), prin ordinea de apel a funcțiilor, și nu static, adică relativ la locul de definire.

Să reamintim în acest scop diferența între determinarea statică și respectiv cea dinamica a domeniului de vizibilitate în cadrul limbajelor imperative. Fie programul Pascal:

```

var
    a:integer;

procedure P;
begin
    writeln(a);
end;

procedure Q;
var
    a: integer;
begin
    a := 7;
    P;
end;

begin
    a := 5;
    Q;
end.

```

**Determinarea statică a domeniului de vizibilitate** (DSDV, static scoping) presupune că procedura P acționează în mediul de definire și ca urmare ea va “vedea” întotdeauna variabila a globală. Este și cazul limbajului Pascal, situație în care P apelat în Q va tipări 5 și nu 7.

Pe de alta parte, dacă am avea de-a face cu **determinarea dinamică a domeniului de vizibilitate** (DDDV, dynamic scoping), procedura P va tipări întotdeauna ultima (în ordinea apelurilor) variabilă *a* definită (sau legată, în terminologie Lisp). În acest caz se va tipări 7, deoarece ultima legare a lui *a* este relativ la valoarea 7.

Limbajul Lisp dispune de DDDV, această abordare fiind mai naturală decât cea statică pentru cazul sistemelor bazate pe interpretoare. O variantă Lisp a programului de mai sus care pune în evidență DDDV este:

```

(DEFUN P () A)
(DEFUN Q ()
  (SETQ A 7)
  (P)
  )
  (SETQ A 5)
  (SETQ Y (LIST (P) (Q)))
  (PRINT Y)

```

Se va tipări lista (5 7), valoarea întoarsă de P fiind de fiecare dată ultimul A legat.

Avantajul legării dinamice este adaptabilitatea, adică o flexibilitate sporită. În cazul argumentelor funcționale însă pot să apară interacțiuni nedorite. Pentru a ilustra genul de probleme care pot apărea să luăm exemplul formei funcționale twice, care aplică de două ori argumentul funcție unei valori:

```
(DEFUN twice (func val) (funcall func (funcall func val)) )
```

Exemple de aplicare:

- (twice 'add1 5) returnează 7
- (twice '(lambda (x) (\* 2 x)) 3) returnează 12

Dacă se întâmplă însă să folosim identificatorul val și în cadrul lui func, ținând cont de DDDV, apare o coliziune de nume, cu următorul efect:

- (setq val 2) returnează 2
- (twice '(lambda (x) (\* val x)) 3) returnează 27, nu 12

(cum probabil am dori să obținem). Aceasta deoarece ultima legare (dinamică deci) a lui val s-a efectuat ca parametru formal al funcției twice, deci val a devenit 3.

Această problemă a fost denumită problema FUNARG (funcțional argument). Este o problemă de conflict între DSDV și DDDV. Rezolvarea ei nu a constat în renunțarea la DDDV pentru LISP ci în introducerea unei forme speciale numite **function**, care realizează legarea unei lambda expresii de mediul ei de definire (deci tratarea aceluia caz în mod static). Astfel,

- (twice (function (lambda (x) (\* val x))) 3) se evaluează la 12

Concluzia este deci că LISP are două reguli de DDV: DDDV implicit și DSDV prin intermediul construcției function.

## 2. Universul expresiilor și universul instrucțiunilor

Orice limbaj de programare poate fi împărțit în două așa-numite universuri (domenii):

1. universul expresiilor;
2. universul instrucțiunilor.

**Universul expresiilor** include toate construcțiile limbajului de programare al căror scop este producerea unei valori prin intermediul procesului de evaluare.

**Universul instrucțiunilor** include instrucțiunile unui limbaj de programare. Acestea sunt de două feluri:

- (i) instrucțiuni ce influențează fluxul de control al programului:
  - instrucțiuni conditionale;
  - instrucțiuni de salt;
  - instrucțiuni de ciclare;
  - apeluri de proceduri și funcții.

(ii) instrucțiuni ce alterează starea memoriei:

- atribuirea (alterează memoria internă, primară);
- instrucțiuni de intrare/ieșire (alterează memoria externă, secundară)

Ca asemănare a acestor două universuri, putem spune că ambele alterează ceva. Există însă deosebiri importante. În universul instrucțiunilor ordinea în care se execută instrucțiunile este de obicei esențială. Adică instrucțiunile

$i := i + 1; a := a * i;$

au efect diferit fata de instrucțiunile

$a := a * i; i := i + 1;$

Fie

$z := (2 * a * y + b) * (2 * a * y + c);$

Multe compilatoare elimină evaluarea redundantă a subexpresiei comune  $2 * a * y$  prin următoarea substituție:

$t := 2 * a * y; z := (t + b) * (t + c);$

Această înlocuire s-a putut efectua în universul expresiilor deoarece în cadrul unei expresii o subexpresie va avea întotdeauna aceeași valoare.

În cadrul universului instrucțiunilor situația se schimbă, datorită posibilelor efecte secundare. De exemplu, pentru secvența

$y := 2 * a * y + b; z := 2 * a * y + c;$

factorizarea subexpresiei comune furnizează secvență neechivalentă

$t := 2 * a * y; y := t + b; z := t + c;$

Deși un compilator poate analiza un program pentru a determina pentru fiecare subexpresie în parte dacă poate fi factorizată sau nu, acest lucru cere tehnici sofisticate de analiză globală a fluxului (global flow analysis). Astfel de analize sunt costisitoare și dificil de implementat. Concluzionăm deci că universul expresiilor prezintă un avantaj asupra universului instrucțiunilor, cel puțin referitor la acest aspect al eliminării subexpresiilor comune (există, oricum, și alte avantaje care vor fi evidențiate în continuare).

Scopul programării funcționale este extinderea la nivelul întregului limbaj de programare a avantajelor pe care le promoveaza universul expresiilor față de universul instrucțiunilor.

### 3. Independența ordinii de evaluare

A evalua o expresie înseamnă a-i extrage valoarea. Evaluarea expresiei  $6 * 2 + 2$  furnizează valoarea 14. Evaluarea expresiei  $E = (2ax + b)(2ax + c)$  nu se poate face până nu precizăm valorile numelor  $a$ ,  $b$ ,  $c$  și  $x$ . Deci, valoarea acestei expresii este dependentă de contextul evaluării. Odată acest lucru precizat, mai este important să subliniem că valoarea expresiei  $E$  nu depinde de ordinea de evaluare (adică de înlocuire a numelor, mai întâi primul factor sau cel de-al doilea, etc). Este posibilă chiar evaluarea paralelă. Aceasta deoarece în cadrul expresiilor pure (așa cum sunt cele matematice) evaluarea unei subexpresii nu poate afecta valoarea nici unei alte subexpresii.

**Definiție.** O expresie pură este o expresie liberă de efecte secundare, adică nu conține atribuirii nici explicit (gen C) și nici implicit (apeluri de funcții).

Această proprietate a expresiilor pure, și anume independența ordinii de evaluare, se numește proprietatea Church-Rosser. Ea permite construirea de compilatoare ce aleg ordini de evaluare care fac uz într-un mod cât mai eficient de resursele masinii. Să subliniem din nou potențialul de paralelism etalat de această proprietate.

Proprietatea de transparență referențială presupune că într-un context fix înlocuirea subexpresiei cu valoarea sa este complet independentă de expresia înconjuratoare. Deci, odata evaluată, o subexpresie nu va mai fi evaluată din nou pentru că valoarea sa nu se va mai schimba. Transparența referențială rezultă din faptul că operatorii aritmetici nu au memorie, astfel orice apel al unui operator cu aceleași intrări va produce același rezultat.

Una dintre caracteristicile noției matematice este interfața manifestă, adică, conexiunile de intrare ieșire între o subexpresie și expresia înconjuratoare sunt vizual evidente (de exemplu în expresia  $3 + 8$  nu există intrări “ascunse”) și ieșirile depind numai de intrări. Producătorii de efecte secundare, deci și funcțiile în general, nu au interfață manifestă ci o interfață nemanifestă (hidden interface).

Să trecem în revistă proprietățile expresiilor pure:

- valoarea este independentă de ordinea de evaluare;
- expresiile pot fi evaluate în paralel;
- transparența referențială;
- lipsa efectelor secundare;
- intrările și efectele unei operații sunt evidente.

Scopul programării funcționale este extinderea tuturor acestor proprietăți la întreg procesul de programare.

Programarea aplicativă are o singură construcție sintactică fundamentală și anume aplicarea unei funcții argumentelor sale. Modurile de definire a funcțiilor sunt următoarele:

1. **Definire enumerativă.** Este posibilă doar când funcția are un domeniu finit de dimensiune redusă.
2. **Definire prin compunere de funcții deja definite.** Dacă e cazul unei definiri în termeni de un număr infinit sau neprecizat de compunerii, o astfel de metodă este utilă doar dacă se poate extrage un principiu de regularitate, principiu care să ne permită generarea cazurilor încă neenumerate pe baza celor specificate. Dacă un astfel de principiu există suntem în cazul unei definiții recursive. În programarea funcțională recursivitatea este metoda de bază pentru a descrie un proces iterativ.

O altă distincție ce trebuie făcută relativ la definirea de funcții se referă la definiții explicite și implicate. O definiție explicită ne spune ce este un lucru. O definiție implicită statuează anumite proprietăți pe care le are un anumit obiect.

Într-o definiție explicită variabila definită apare doar în membrul stâng al ecuației (de exemplu  $y = 2ax$ ). Definițiile explicite au avantajul că pot fi interpretate drept reguli de rezcriere (reguli care specifică faptul că o clasă de expresii poate fi înlocuită de alta).

O variabilă este definită implicit dacă ea este definită de o ecuație în care ea apare în ambele membri (de exemplu  $2a = a + 3$ ). Pentru a-i găsi valoarea trebuie rezolvată ecuația.

Aplicarea repetată a regulilor de rezcriere se termină întotdeauna. Este posibil însă ca anumite definiții implicate să nu ducă la terminare (adică ele nu definesc nimic). De exemplu, ecuația fără soluție  $a = a + 1$  duce la un proces infinit de substituție.

Un avantaj al programării funcționale este că, la fel ca și în cazul algebrei elementare, ea simplifică transformarea de definiții implicate în definiții explicite. Acest lucru este foarte important deoarece specificațiile formale ale sistemelor soft au de obicei forma unor definiții implicate, însă conversia specificațiilor în programe are loc mai ușor în cazul definițiilor explicite.

Să mai subliniem că definițiile recursive sunt prin natura lor implicate. Totuși, datorită formei lor regulate de specificare (adică membrul stâng e format numai din numele și argumentele funcției) ele pot fi folosite ca reguli de rezcriere. Condiția de oprire asigură terminarea procesului de substituții.

Față de limbajele conventionale, limbajele funcționale diferă și în ceea ce privește modelele operaționale utilizate pentru descrierea execuției programelor. Nu este proprie de exemplu pentru limbajele funcționale urmărirea execuției pas cu pas, deoarece nu conțează ordinea de evaluare a subexpresiilor. Deci nu avem nevoie de un depanator în adevăratul înțeles al cuvântului.

Totuși, limbajul Lisp permite urmărirea execuției unei forme Lisp. Pentru aceasta există macrodefiniția TRACE care primește ca argument numele funcției dorite, rezultatul întors de TRACE fiind T dacă funcția respectivă există și NIL în caz contrar.

## CURS 12

### Generatori. Argumente optionale. OBLIST și ALIST. Macrodefiniții. Apostroful invers

#### Cuprins

1.	Generatori .....	1
2.	Argumente optionale.....	3
3.	OBLIST și ALIST.....	5
4.	Macrodefiniții .....	7
5.	Apostroful invers (backquote) .....	8

#### 1. Generatori

Ca exemplu sugestiv pentru această problematică să consideram exemplul unei funcții **VERIF** care primește o listă de liste și întoarce T dacă toate sublistele de la nivel superficial sunt liniare și NIL în caz contrar.

(VERIF '((1 2) (a (b)))) va produce NIL  
 (VERIF '((1 2) (a b))) va produce T

Vom folosi două funcții

- Funcția **LIN**, care verifică dacă o listă este liniară

Model recursiv

$$\text{LIN}(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } l_1 \text{ nu e atom} \\ \text{LIN}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN LIN (L)
  (COND
    ((NULL L) T)
    (T (AND (ATOM (CAR L)) (LIN (CDR L)))))
  )
)
```

- Funcția **VERIF**, care verifică dacă o listă de liste are toatele sublistele de la nivel superficial liste liniare.

Model recursiv

$$\text{VERIF}(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } \neg \text{LIN}(l_1) \\ \text{VERIF}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN VERIF (L)
  (COND
    ((NULL L) T)
    (T (AND (LIN (CAR L)) (VERIF (CDR L)))))
  )
)
```

Să observăm că funcțiile LIN și VERIF de mai sus au aceeași structură, conformă şablonului

```
(DEFUN F (L)
  (COND
    ((NULL L) T)
    (T (AND (F1 (CAR L)) (F (CDR L))))
  )
)
```

cu  $F1 = \text{LIN}$  în cazul  $F = \text{VERIF}$  și  $F1 = \text{ATOM}$  în cazul  $F = \text{LIN}$ . O astfel de structură este des folosită, “rețeta” ei de lucru fiind: elementele unei liste L sunt transmise pe rând (în ordinea apariției lor în listă) unei funcții F (în cazul de mai sus F1) care le va prelucra. Dacă F întoarce NIL acțiunea se încheie, rezultatul final fiind NIL. Altfel, parcurgerea continuă până la epuizarea tuturor elementelor, caz în care rezultatul final este T. Cum această rețetă se poate aplica pentru orice funcție F ce realizează prelucrarea într-un anume fel a tuturor elementelor unei liste, apare ideea de a scrie o funcție **generică** (şablon) GEN ce respectă “rețeta”, având doi parametri: funcția ce va realiza prelucrarea și lista asupra căreia se va acționa.

Terminologia adoptată pentru astfel de funcții (sau similare) este cea de **generatori**. Nu este necesar ca un generator să întoarcă T sau NIL. În funcție de implementare se pot concepe generatori care să întoarcă, de exemplu, lista tuturor rezultatelor non-NIL ale lui F culese până în momentul închiderei acțiunii generatorului (L este vidă sau F întoarce NIL).

#### Model recursiv

$$\text{GEN}(F, l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } \neg F(l_1) \\ \text{GEN}(F, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN GEN (F L)
  (COND
    ((NULL L) T)
```

```
(T (AND (FUNCALL F (CAR L)) (GEN F (CDR L))))
  )
)
```

După ce a fost definit generatorul, se poate defini funcția VERIF (L) astfel:

```
(DEFUN VERIF (L)
  (GEN #'(LAMBDA (L)
    (GEN #'ATOM L)
    )
  L
  )
)
```

## 2. Argumente optionale

În lista parametrilor formali ai unei funcții putem folosi următoarele variabile: &OPTIONAL și &REST.

- &OPTIONAL - dacă apare în lista parametrilor formali atunci următorul parametru formal va lua ca valoare parametrul corespunzător din lista parametrilor actuali, respectiv NIL dacă parametrul actual nu există;
- &REST - dacă apare în lista parametrilor formali atunci următorul parametru formal va lua ca valoare lista parametrilor actuali rămași neatribuiți, respectiv NIL dacă nu mai există parametri actuali neatribuiți.

### Exemple

#### Exemplu 1

```
(defun f (x &optional y)
  (cond
    ((null y) x)
    (t (+ x y))
  )
)
```

- (f 1 2) → 3
- (f 3) → 3

#### Exemplu 2

```
(defun g (x &rest y)
```

```
(+ x (apply #'+ y))
)
```

- (g 1 2 3) → 6
- (g 4 5) → 9
- (g 3) → 3

### Exemplu 3

```
(DEFUN F (L1 &REST L2)
  (COND
    ((NULL (CAR L2)) NIL)
    (T (CONS (MAPCAR #'* L1 (CAR L2)) (F L1 (CADR L2))))
  )
)
```

- (F '(1 2) '(3 4) '(5 6)) se evaluează la ((3 8) (5 12))

- **Exemplu 4**

Să presupunem că dorim să construim o funcție care să adune 1 la valoarea unei expresii:

```
(DEFUN INC (NUMAR)
  (+ NUMAR 1)
)
```

Această funcție se va utiliza în felul următor:

- (INC 10) se evalua la 11

Sigur că, în măsura în care dorim, putem adăuga funcții asemănătoare și pentru alte valori de incrementare. Alternativ, putem să rescriem funcția INC astfel încât să accepte un alt doilea parametru:

```
(DEFUN INC (NUMAR INCREMENT)
  (+ NUMAR INCREMENT)
)
```

In marea majoritate a cazurilor, totuși, vom avea nevoie de incrementarea cu 1 a valorii expresiei corespunzătoare lui NUMAR. In această situație putem folosi facilitatea argumentelor optionale. Iată în continuare definiția funcției INC cu un argument optional:

```
(DEFUN INC (NUMAR &OPTIONAL INCREMENT)
  (COND
    ((NULL INCREMENT) (+ NUMAR 1))
    (T (+ NUMAR INCREMENT))
  )
)
```

Caracterul & din &OPTIONAL semnalează faptul că &OPTIONAL este un separator de parametri, nu un parametru propriu-zis. Parametrii care urmează după &OPTIONAL sunt legați la intrarea în procedură la fel ca și ceilalți parametri. Dacă nu există parametru actual corespunzător, valoarea unui parametru formal optional este considerată NIL. Iată câteva exemple de utilizare a funcției INC pe care tocmai am construit-o:

- (INC 10) se evaluează la 11
- (INC 10 3) se evaluează la 13
- (INC (\* 10 2) 5) se evaluează la 25

Putem, de asemenea, să luăm în considerare o valoare implicită pentru parametrii optionali. Iată o definiție a funcției INC în care parametrul optional are valoarea implicită 1:

```
(DEFUN INC (NUMAR &OPTIONAL (INCREMENT 1))
  (+ NUMAR INCREMENT)
  )
```

Observați că de această dată în locul parametrului optional este precizată o listă formată din două elemente: primul element este numele parametrului optional, iar al doilea element este valoarea cu care se inițializează acesta atunci când argumentul actual corespunzător nu este prezent.

Să notăm că putem avea oricâte argumente optionale. Toate aceste argumente vor fi trecute în lista parametrilor după simbolul &OPTIONAL, care va apărea o singură dată. De asemenea, putem avea un singur argument optional semnalat prin &REST, a cărui valoare devine lista tuturor argumentelor date cu excepția celor care corespund parametrilor obligatorii și optionali. Fie o nouă versiune a funcției INC:

```
(DEFUN INC (NUMAR &REST INCREMENT)
  (COND
    ((NULL INCREMENT) (+ NUMAR 1))
    (T (+ NUMAR (APPLY #'+ INCREMENT)))
    )
  )
```

Iată câteva exemple de aplicare a acestei funcții:

- (INC 10) se evaluează la 11
- (INC 10 1 2 3) se evaluează la 16

### 3. OBLIST și ALIST

Un obiect Lisp este o structură alcătuită din:

- numele simbolului;
- valoarea sa;
- lista de proprietăți asociată simbolului.

Gestiunea simbolurilor folosite într-un program Lisp este realizată de sistem cu ajutorul unei tabele speciale numită lista obiectelor (**OBLIST**). Orice simbol este un obiect unic în sistem. Apariția unui nou simbol determină adăugarea lui la OBLIST. La fiecare întâlnire a unui atom, el este căutat în OBLIST. Dacă se găsește, se întoarce adresa lui.

Pentru implementarea mecanismului de apel, sistemul Lisp folosește o altă listă, numită lista argumentelor (**ALIST**). Fiecare element din **ALIST** este o pereche cu punct formată dintr-un parametru formal și argumentul asociat sau valoarea acestuia.

In general, o funcție definită cu  $n$  parametri  $P_1, P_2, \dots, P_n$  și apelată prin  $(F\ A_1\ A_2 \dots A_n)$  va adăuga la **ALIST**  $n$  perechi de forma  $(P_i . A_i)$  dacă funcția își evaluează argumentele sau  $(P_i . A'_i)$ , unde  $A'_i$  este valoarea argumentului  $A_i$ , dacă mediul Lisp evaluează argumentele. De exemplu, pentru funcția

```
(DEFUN F (A B)
  (COND
    ((ATOM B) A)
    (T (CONS A B)))
  )
)
```

apelată cu  $(F\ 'X\ '(1.1))$ , vom avea pentru **ALIST** structura  $( \dots (A . X) (B . (1 . 1)))$ .

*Dacă la momentul apelului simbolurile reprezentând parametrii formali aveau deja valori, acestea sunt salvate în vederea restaurării ulterioare.*

Simbolurile sunt reprezentate în structură prin adresa lor din **OBLIST**, iar atomii numerici și cei sir de caractere prin valoarea lor direct în celula care îi conține.

Inițial, la începutul programului, **ALIST** este vidă. Pe măsura evaluării de noi funcții, se adaugă perechi la **ALIST**, iar la terminarea evaluării funcției, perechile create la apel se sterg. În corpul funcției în curs de evaluare valoarea unui simbol s este căutată întâi în perechile din **ALIST** începând dinspre vîrf spre bază (această acțiune este cea care stabilește dinamic domeniul de vizibilitate). Dacă valoarea nu este găsită în **ALIST** se continuă căutarea în **OBLIST**. Așadar, este clar că **ALIST** și **OBLIST** formează contextul curent al execuției unui program.

### Exemplu

- $(SETQ X 1)$  inițializează simbolul  $X$  cu valoarea 1;
- $(SETQ Y 10)$  inițializează simbolul  $Y$  cu valoarea 10;
- $(DEFUN DEC (X) (SETQ X (- X 1)))$  definește o funcție de decrementare a valorii parametrului;
- $(DEC X)$  se evaluează la 0;
- $X$  se evaluează tot la 1;
- $(DEC Y)$  se evaluează la 9;
- $Y$  se evaluează tot la 10.

Să observăm deci că modificările operate asupra valorilor simbolurilor reprezentând argumentele formale se vor pierde după ieșirea din corpul funcției și revenirea în contextul apelator.

#### 4. Macrodefiniții

Din punct de vedere sintactic, macrodefinițiile se construiesc în același mod ca funcțiile, cu diferența că în loc să se folosească funcția DEFUN se va folosi funcția DEFMACRO:

**(DEFMACRO s l f1 ... fn): s**

Funcția DEFMACRO creează o macrodefiniție având ca nume primul argument (simbolul s), iar ca parametri formalii elementele simboluri ale listei ce constituie al doilea argument; corpul macrodefiniției create este alcătuit din una sau mai multe forme aflate, ca argumente, pe a treia poziție și eventual pe următoarele. Valoarea întoarsă ca rezultat este numele macrocomenzii create. Funcția DEFMACRO nu-și evaluează niciun argument.

- evaluarea argumentelor face parte din procesul de evaluare al macrodefiniției

Modul de lucru al macrodefinițiilor create cu DEFMACRO este diferit de cel al funcțiilor create cu DEFUN:

- parametrii macrodefiniției nu sunt evaluati;
- se evaluează corpul macrodefiniției și se va produce o S-expresie intermedieră;
- această S-expresie va fi evaluată, acesta fiind momentul în care sunt evaluati parametrii.

Să vedem mai întâi un exemplu comparativ. Fie următoarele definiții și evaluări:

<pre>&gt;(DEFUN F (N)   (PRINT N)   ) &gt;(SETQ N 10) &gt;(F N) 10 10</pre> <p><b>Notă</b> Parametrul N este evaluat de la bun început, PRINT va tipări valoarea acestuia, și o va întoarce ca valoare a apelului funcției.</p>	<pre>&gt;(DEFMACRO G (N)   (PRINT N)   ) &gt;(SETQ N 10) &gt;(G N) N 10</pre> <p><b>Notă</b> În primul moment nu se face o încercare de a se evalua parametrul N. Ca atare, PRINT va tipări N, după care îl va întoarce pe N ca valoare a formei intermediere a macrodefiniției. Apoi această valoare intermedieră este evaluată și se va produce 10.</p>
---	---

Ca un alt exemplu, dorim să producem o macrodefiniție DEC care să decrementeze cu 1 valoarea parametrului ei, ca în exemplul următor:

- (SETF X 10) se evaluează la 10
- (DEC X) va produce 9
- X se evaluează la 9

Aceasta înseamnă că forma intermediară va trebui să fie

(SETQ parametru (- parametru 1))

Iată o posibilitate:

```
(DEFMACRO DEC (N)
  (LIST 'SETQ N (LIST '- N 1))
)
```

## 5. Apostroful invers (backquote)

**Apostroful invers** (`) ușurează foarte mult scrierea macrocomenzilor. Oferă o modalitate de a crea expresii în care cea mai mare parte este fixă, și în care doar câteva detalii variabile trebuie completeate. Efectul apostrofului invers este asemănător cu al apostrofului normal ('), în sensul că blochează evaluarea S-expresiei care urmează, cu excepția că orice **virgulă** (,) care apare va produce *evaluarea* expresiei care urmează. Iată un exemplu:

- (SETQ V 'EXEMPLU)
- `(ACESTA ESTE UN ,V) se evaluează la (ACESTA ESTE UN EXEMPLU)

De asemenea, apostroful invers acceptă construcția ,@. Această combinație produce și ea evaluarea expresiei care urmează, dar cu diferența că valoarea rezultată trebuie să fie o listă. Elemente acestei liste sunt dizolvate în lista în care apare combinația ,@. Iată un exemplu:

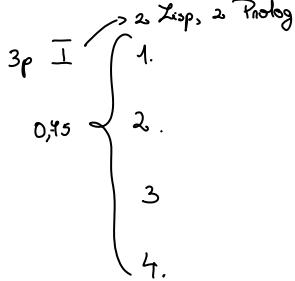
- (SETQ V `'(ALT EXEMPLU))
- `(ACESTA ESTE UN ,V) se evaluează la  
(ACESTA ESTE UN (ALT EXEMPLU))
- `(ACESTA ESTE UN ,@V) se evaluează la  
(ACESTA ESTE UN ALT EXEMPLU)

Folosind apostroful invers, macrodefiniția DEC se va putea scrie mai simplu astfel:

```
(DEFMACRO DEC (N)
  `'(SETQ ,N (- ,N 1))
)
```

## Examen PLT

→ 1 ană și 30 de min  
+ model matematic, modul de flux



3p II backtracking Prolog (! fără brute-force): direct recursiv sau cu & colectare

3p III funcții MAP Lisp  
( liste melanjante + anbori m-ani)

exemplu:

Lisp:

① ( defun f(l)  
 ( car l)  
 )

( defun g(l)  
 ( cdn l)  
 )

\* ( setq h #'f) → f

\* ( set h #'*g*)

\* f

↳ closure    cdn l ⇒ stim că e funcție ⇔ (f '(1 2 3))

\* ( funcall #'f '(1 2 3)) ⇒ 1

\* ( funcall f '(1 2 3)) ⇒ (2 3)

( defun  $f(e)$   
 #'( lambda (l)  
     ( cons e l)  
 ) )

\* ( $f 2$ )

< closure ... (cons e l) >

\* (apply ( $f 2$ ) '(3 4)) → (2 3 4)

\* (funcall ( $f 2$ ) '(3 4)) → (2 3 4)