# CURS 8

# Funcții LISP. Exemple

## Cuprins



## 1. Funcții Lisp (cont.)

**(LIST e1 e2…): l**

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

Câteva exemple:

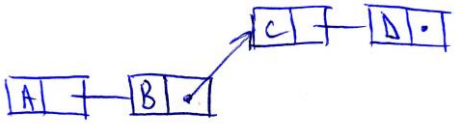
- (LIST 'A 'B) = (A B)
- (LIST '(A B) 'C) = ((A B) C)
- (LIST 'A) = (A) ; echivalent cu (CONS A NIL)
- (LIST '(A B C) NIL) = ((A B C) NIL)

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

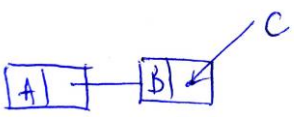
- se evaluează argumentele și apoi se trece la evaluarea funcției
- copiază structura fiecărui argument, înlocuind CDR-ul fiecărui ultim CONS cu argumentul din dreapta. Întoarce lista rezultată.
- toţi parametrii atomi, eventual cu excepţia ultimului parametru, sunt ignoraţi.
- Dacă toate S-expresiile argument sunt liste, atunci efectul este concatenarea listelor
- funcţia 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)



- (APPEND '(A B C) '()) = (A B C)
- (APPEND 'A ' (C D) 'E 'F) = (C D . F)
- (APPEND '(A . B) ' C) = (A . C )

**(LENGTH l): n**

- se evaluează 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 semnificaţia de fals, şi T, cu semnificaţia de adevărat. Ca şi în alte limbaje, funcţiile care returnează o valoare logică vor returna exclusiv NIL sau T. Totuşi, se acceptă că orice valoare diferită de NIL are semnificaţia de adevărat.

**(ATOM e) : T, NIL**

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

Câteva exemple:

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

**(LISTP e) : T, NIL**

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

Câteva exemple:

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

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

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

De exemplu,

| $e_1$ | $e_2$ | 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 evaluează argumentul și apoi se trece la evaluarea funcției
- Întoarce T dacă argumentul se evaluează la o listă vidă sau atom nul şi NIL în caz contrar.

**(NUMBERP e) : T, NIL**

- se evaluează argumentul și apoi se trece la evaluarea funcției
- Verifică daca argumentul este număr sau nu.

## 1.2 Operaţii logice

**(NOT e) : T, NIL**

- se evaluează 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 funcţia NULL.

**(AND e1 e2 … ) : e**

- Evaluarea argumentelor face parte din procesul de evaluare al funcției
- Se evaluează 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=Ø **atunci**
    returnează Ø
**altfel**
    returnează (car l)
**SfDacă**

**(OR e1 e2 … ) : e**

- Evaluarea argumentelor face parte din procesul de evaluare al funcției
- Se evaluează 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** (cdr l) ≠ Ø **atunci**
    returnează (cdr l)

**altfel**
returnează (car l)
**SfDacă**

### 1.3 Operaţii aritmetice

**(+ n1 n2 … ) : n**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1+n2+…

**(- n1 n2 … ) : n**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1-n2-…

**(* n1 n2 … ) : n**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1*n2*…

**(/ n1 n2 … ) : n**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1/n2/…

**(MAX n1 n2 … ) : n**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: maximul valorilor argumentelor

**(MIN n1 n2 … ) : n**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: minimul valorilor argumentelor

Pentru numere întregi se pot folosi MOD și DIV.

### 1.4 Operatori relaționali

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

## 2. Ramificarea prelucrărilor. Funcția COND

Funcţia 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 **l1**, **l2**,…**ln** sunt liste nevide de lungime arbitrară **($f_1$ $f_2$ ... $f_n$)** numite *clauze*. COND admite oricâte clauze ca argumente şi oricâte forme într-o clauză. Iată modul de funcţionare a funcţiei COND:

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

Următoarea secvenţă

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

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

## 3. Definirea funcțiilor utilizator. Funcția DEFUN

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

Funcţia DEFUN crează o nouă funcţie având ca nume primul argument (simbolul **s**), iar ca parametri formali elementele simboluri ale listei ce constituie al doilea argument; corpul funcţiei create este alcătuit din una sau mai multe forme aflate, ca argumente, pe poziţiile a treia şi eventual următoarele (evaluarea acestor forme la execuţie reprezintă efectul secundar). Se întoarce numele funcţiei create. Funcţia DEFUN nu-şi evalueaza nici un argument.

Apelul unei funcţii definită prin
(DEFUN **fnume** ($p_1$ $p_2$ ... $p_n$)
        ...
)

este o formă

(**fnume** $arg_1$ $arg_2$ ... $arg_n$)

unde **fnume** este un simbol, iar **$arg_i$** sunt forme; evaluarea apelului decurge astfel:

(a) se evaluează argumentele $arg_1$, $arg_2$, ... $arg_n$; fie $v_1$, $v_2$, ... $v_n$ valorile lor;
(b) fiecare parametru formal din definiţia funcţiei 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 parametri formali aveau deja valori, acestea sunt salvate în vederea restaurării ulterioare;
(c) se evaluează în ordine fiecare formă aflată în corpul funcţiei, valoarea ultimei forme fiind întoarsă ca valoare a apelului funcţiei;

(d) se restaurează valorile parametrilor formali, adică $p_1$, $p_2$ ... pn se "dezleagă" de valorile $v_1$, $v_2$, ... şi se leagă din nou la valorile corespunzătoare salvate (dacă este cazul).

**Observaţie.** DEFUN poate redefini si funcţiile sistem standard. Spre exemplu dacă se redefineşte funcţia 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 şi 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 si nu produce mesaj de eroare.

```
(DEFUN XCAR (X)
        (COND
                ((ATOM X) NIL)
                (T (CAR X))
        )
)
```

**Observaţii.**

(1). DEFUN poate redefini si funcţiile sistem standard. Spre exemplu dacă se redefineşte funcţia 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 redefnită 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).

## 4. Exemple

**EXEMPLU 4.1** Să se construiască lista obținută prin adăugarea unui element la sfârşitul unei liste.

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

Model recursiv

; construirea listei $(l_1, l_2, \ldots, l_n, e)$

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

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

**EXEMPLU 4.2. 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

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

O definiţie posibilă pentru funcţia INVERS este următoarea:

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

Problema este că o astfel de definiţie consumă multă memorie. Eficienţa aplicării funcţiilor 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) şi să subliniem de asemenea că funcţia APPEND acţionează prin copierea primului argument, care este apoi "lipit" de cel de-al doilea argument. Astfel, funcţia 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 + ... + (N-1) = N (N-1) / 2 folosiri de CONS-uri. Deci, complexitatea timp este $\boldsymbol{\theta(n^2)}$, *n* fiind numărul de elemente din listă.

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

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

| L | Col |
|---|---|
| (1, 2, 3) | ∅ |
| (2, 3) | (1) |
| (3) | (2, 1) |
| ∅ | (3, 2, 1) |

Modelele recursive

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

$$invers(l_1 l_2 \dots l_n) = invers\_aux(l_1 l_2 \dots l_n, \varnothing)$$

```
(DEFUN INVERS_AUX (L Col)
        (COND
                ((NULL L) Col)
                (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 funcţii ce trebuie apelată cu (INVERS L). De aceea, funcţia INVERS se va defini:

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

Functia INVERS_AUX are rolul de functie auxiliară, ea punând în evidenţă rolul argumentului **Col** ca **variabilă colectoare** (variabilă ce colectează rezultatele parţiale până la obţinerea rezultatului final). Se vor efectua atâtea CONS-uri cât este lungimea listei sursă L, deci complexitatea timp este **θ(*n*)**, *n* fiind numărul de elemente din listă.

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

Să observăm deci că colectarea rezultatelor parţiale într-o variabilă separată poate contribui la micşorarea complexităţii 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 4.3.** Să se definească o funcţie care să determine lista perechilor dintre un elelement dat si elementele unei liste.

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

Modelul recursiv este

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

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

**EXEMPLU 4.4.** Să se definească o funcţie 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))

Vom folosi o funcție auxiliară care returnează lista perechilor cu element 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))

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

```
(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)))
           )
        )
    )
)
```

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

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

**EXEMPLU 4.5** Se dă o listă neliniară. Se cere să se dubleze valorile numerice de la orice nivel al listei, păstrând structura ierarhică a acesteia.
(dublare '(1 b 2 (c (3 h 4)) (d 6))) → (2 b 8 (c (6 h 8)) (d 12)))

**Varianta 1**

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

```
(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**

$$dublare(l) = \begin{cases} 2l & \text{dacă } l \text{ numar} \\ l & \text{dacă } l \text{ atom} \\ lista(l_1) \oplus lista(l_2 \dots l_n) & altfel, l = (l_1 l_2 \dots l_n) e\ lista \end{cases}$$

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

**EXEMPLU 4.6** ***Folosirea unei variabile colectoare poate crește complexitatea.*** Se dă o listă liniară. Care este efectul următoarei evaluări:

(lista '(1 a 2 b 3 c)) → ???

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?

**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?

**EXEMPLU 4.7** 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 nil)
)
```

(parcurg '(1 2 3 4 5) 3) → ((3 2 1) (4 5))