

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

Câteva exemple:

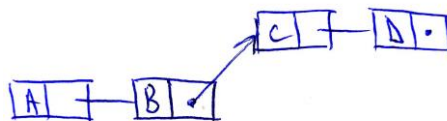
- (LIST 'A 'B) = (A B)
- (LIST '(A B) 'C) = ((A B) C)
- (LIST 'A) = (A) ; echivalent cu (CONS A NIL)
- (LIST '(A B C) NIL) = ((A B C) NIL)

(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.
- 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)



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

(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 semnificația de **fals**, și T, cu semnificația de **adevărat**. Ca și în alte limbaje, 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 evaluează 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 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 și 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 aceeași 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 și NIL în caz contrar.

(NUMBERP e) : T, NIL

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

1.2 Operații 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 funcția 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 1 (car 1)) este echivalentă cu următoarea structură alternativă

Dacă 1=Ø atunci

returnează Ø

altfel

returnează (car 1)

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 1) (car 1)) este echivalentă cu următoarea structură alternativă

Dacă (cdr 1) ≠ Ø atunci

returnează (cdr 1)

altfel

returnează (car l)
SfDacă

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

1.3 Operații aritmetice

(+ n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: $n1+n2+\dots$

(- n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: $n1-n2-\dots$

(* n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: $n1*n2*\dots$

(/ n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: $n1/n2/\dots$

(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 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

În descrierea de mai sus **f₁, f₂, ..., f_n** sunt liste nevide de lungime arbitrară (**f₁ f₂ ... f_n**) 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 f₂, f₃, ... 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 formali 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 evaluează nici un argument.

Apelul unei funcții definite prin

```
(DEFUN fnume (p1 p2 ... pn)
  ...
)
```

este o formă

```
(fnume arg1 arg2 ... argn)
```

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

- (a) se evaluează 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 formali aveau deja valori, acestea sunt salvate în vederea restaurării ulterioare;
- (c) se evaluează în ordine fiecare formă aflată în corpul funcției, valoarea ultimei forme fiind întoarsă ca valoare a apelului funcției;
- (d) se restaurează valorile parametrilor formali, 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 redefini 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 redefinește 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 redefinită 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)
  )
)

```

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](#))

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

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:


```

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

```

Complexitatea timp e data de recurența

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

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:

L	Col
(1, 2, 3)	\emptyset
(2, 3)	(1)
(3)	(2, 1)
\emptyset	(3, 2, 1)

Modelele recursive

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

$$invers(l_1 l_2 \dots l_n) = invers_aux(l_1 l_2 \dots l_n, \emptyset)$$

```

(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 funcții ce trebuie apelată cu (INVERS L). De aceea, funcția INVERS se va defini:

```

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

```

Funcția 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

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

```

(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))
```

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.

(dublare '(1 b 2 (c (3 h 4)) (d 6)))) → (2 b 8 (c (6 h 8)) (d 12)))

Varianta 1

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

```

(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

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

```

(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)) → ???

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) → ???
```