LUCRAREA 5- Expresii LAMBDA.
Funcții de ordin superior. Repetiție prin mapare

1. SCOPUL LUCRĂRII

În această lucrare sunt prezentate modalitatea de a defini şi utiliza funcții anonime în Lisp prin intermediul expresiilor LAMBDA. De asemenea este prezentat modul de utilizare a funcțiilor de ordin superior, care aplică alte funcții legate ca valoare unor simboluri care pot fi transmise şi ca parametri. Este arătat si un iterator care aplică o funcție pe o listă de argumente.

2. CONSIDERAȚII TEORETICE

2.1. EXPRESII LAMBDA

În Lisp programele (funcțiile) şi datele au aceeaşi reprezentare internă: liste formate din celule CONS. Această omogenitate în reprezentare permite tratarea unitară a datelor şi a funcțiilor. Întrucât există date anonime (de exemplu 1., "abc" , (+ 2 3)), ar trebui să existe şi funcții anonime. Acestea sunt în Lisp LAMBDA-expresiile. Exemplu: (LAMBDA (x y) (+ x y)) specifică o funcție de două argumente care calculează suma lor. LAMBDA-expresiile nu sunt forme, ele nu se pot evalua direct, dar se pot aplica pe date analog funcțiilor obişnuite: pe prima poziție a unei liste apare o LAMBDA-expresie, nu neapărat un nume de funcție. Exemplu de aplicare LAMBDA-expresie:

EXECUTIE	REZULTAT
*((LAMBDA (x y) (+ x y)) 2 3)	5

Observație: Lista de parametri formali a LAMBDA-expresiilor este analaogă celei permise de DEFUN, adică pot apare parametri introduşi de cuvintele cheie "&optional", "&rest", "&aux".

Intern LAMBDA-expresiile sunt folosite şi pentru a specifica corpul funcțiilor definite de utilizator. Forma (SYMBOL-FUNCTION <simbol>) întoarce LAMBDA-expresia ce a fost ataşată unui simbol de către DEFUN. De fapt DEFUN este echivalent cu o setare indirectă prin intermediul SYMBOL-FUNCTION. Exemplu:

EXECUTIE	REZULTAT
*(DEFUN foo (x y) (+ x y))	foo
*(SYMBOL-FUNCTION 'foo)	(LAMBDA (x y) (+ x y))
(SETF (SYMBOL-FUNCTION 'foo) '(LAMBDA (x y) (x y)))	(LAMBDA (x y) (* x y))
*(foo 3 10)	30

2.2. ARGUMENTE FUNCȚIONALE

În general pentru a ataşa unui simbol o funcție se foloseşte DEFUN. Interpretorul nu "ştie" să aplice funcții ce sunt ataşate ca valoare unui simbol. În unele cazuri însă este nevoie să putem aplica funcții sau LAMBDA-expresii, ce sunt ataşate ca valoare unui simbol. Cel mai frecvent caz este atunci când dorim sa transmitem ca parametru unei funcții o altă funcție sau LAMBDA-expresie. În aceste cazuri trebuie folosite primitivele FUNCALL şi APPLY. Să amintim că în Common Lisp expresiile funcționale se citează cu "#'" care este o prescurtare pentru FUNCTION, nu cu "'" prescurtarea pentru QUOTE!

(FUNCALL <form> <arg1> <arg2> ... <argn>)

<form> trebuie să fie o formă Lisp care în urma evaluării să întoarcă o expresie funcțională (nume de funcție, LAMBDA-expresie, macrodefiniție, CLOSURE, etc). Se evaluează <form> şi se aplică funcția întoarsă în urma evaluării pe cele n argumente ce urmează.

(APPLY <form> <larg>)

<form> trebuie să fie o formă care în urma evaluării să întoarcă o expresie funcțională, iar <larg> trebuie să fie o listă ce conține argumente în numărul şi de tipul aşteptat de funcția întoarsă ca efect al evaluării <form>. Se aplică funcția întoarsă pe argumentele conținute în lista <larg>.

Formele FUNCALL şi APPLY se folosesc în general atunci când:

I. aplicăm funcții transmise ca parametri altor funcții

II. aplicăm funcții construite de alte funcții Lisp

În plus funcția APPLY mai este folosită atunci când dorim să construim prin program lista de argumente pe care se aplică o funcție. Exemple:

EXECUTIE	REZULTAT
*(DEFUN foo (x) (+ x 10))	foo
(SETQ foo #'(LAMBDA (x) (x 10)))	(LAMBDA (x) (* x 10))
*(foo 5)	15
*(FUNCALL foo 5)	50
*(APPLY foo '(5))	50
*(FUNCALL #'foo 5)	15
*(FUNCALL (LIST 'LAMBDA () 10))	10
(APPLY #'MAX `(,(+ 2 3) ,(2 3)))	6
*(DEFUN foo-2 (fun-param) (FUNCALL fun-param 'ALFA))	foo-2
*(foo-2 #'LIST)	(ALFA)

2.3. REPETIȚIE PRIN MAPARE

(MAPCAR <fun> <larg1> <larg2> ... <largn>)

- <fun> trebuie să fie o formă care în urma evaluării să întoarcă o expresie funcțională ce acceptă n argumente. Se aplică funcția pe n-tuplele construite din elementele aflate pe poziții corespondente în cele n liste şi se returnează lista ce conține rezultatele aplicărilor.Exemplu:

EXECUTIE	REZULTAT
*(MAPCAR #'(LAMBDA (x y z) (list x y z)) '(a1 a2 a3 a4) '(b1 b2 b3) '(c1 c2 c3))	((A1 B1 C1) (A2 B2 C2) (A3 B3 C3))

(MAPLIST <fun> <larg1> <larg2> ... <largn>)

- ca şi MAPCAR, dar argumentele funcției nu sunt n-tuple de elemente din liste, ci n-tuple de liste scurtate succesiv. Exemplu:

EXECUTIE	REZULTAT
*(MAPLIST #'(LAMBDA (x) x) '(1 2 3 4))	((1 2 3 4) (2 3 4) (3 4) (4))
*(MAPLIST #'LIST '(1 2) '(a b))	(((1 2) (a b)) ((2) (b)))

(MAPC <fun> <larg1> <larg2> ... <largn>)	**- ca şi MAPCAR, dar nu returnează lista rezultatelor, ci <larg1>. E folosită când e important doar efectul lateral al aplicării repetate a funcției întoarse de evaluarea formei <fun>. Exemplu:** **EXECUTIE** / **REZULTAT** ***(MAPC #'SET '(a b) '(3 u)) → (a b)** ***a → 3** ***b → u**
(MAPL <fun> <larg1> <larg2> ... <largn>)	**- ca şi MAPLIST, dar nu se întoarce lista rezultatelor, ci <larg1>.**
(MAPCAN <fun> <larg1> <larg2> ... <largn>)	**- ca şi MAPCAR, dar rezultatul întors nu este lista ce conține rezultatele aplicării iterative a funcției, ci lista ce rezultă prin concatenarea rezultatelor aplicării funcției. Este clar că pentru aceasta trebuie ca funcția aplicată să întoarcă liste. Atenție, concatenarea se face distructiv, prin modificarea ultimului pointer din fiecare listă (se foloseşte NCONC, nu APPEND)! Exemplu:** **EXECUTIE** / **REZULTAT** ***(MAPCAN #'(LAMBDA (x) (LIST x 0)) '(1 2 3)) → (1 0 2 0 3 0)**
(MAPCON <fun> <larg1> <larg2> ... <largn>)	**- ca şi MAPCAN, dar argumentele pentru funcție sunt liste scurtate succesiv. Exemplu:** **EXECUTIE** / **REZULTAT** ***(MAPCON #'(LAMBDA (x) (APPEND x nil)) '(1 2 3)) → (1 2 3 2 3 3)**

---

3. DESFĂSURAREA LUCRĂRII

1. Să se evalueze formele din exercițiile de mai jos.

```
((LAMBDA (x y) (+ x y)) 2. 3.)

((LAMBDA (x y)
        ((LAMBDA (x z) (+ x y z) ) (+ x y) y)) 4. 5. )

(SETQ a 'LIST)
(a 'a 'b 'c)
(FUNCALL a 'a 'b 'c)
(APPLY a '(a b c))
(FUNCALL 'LIST 'a 'b)
(APPLY 'LIST '(a b))

(SETQ CAR 'CDR)
(CAR '(a b) )
(FUNCALL CAR '(a b))
(APPLY CAR '((a b)) )
(FUNCALL 'CAR '(a b))
(APPLY 'CAR '((a b)) )
```

```
( (LAMBDA (fun larg)
        (APPLY      fun     (CONS 'arg0 larg)) )
          #'LIST    (APPEND '(1 2) '(3 4)) )
        (FUNCALL    (APPEND     '(LAMBDA (x y))
                                '((CONS x y)) )
                    'a '(b)
)
```

```
(MAPCAR #'+ '(1 2 3) '(4 5 6))
(MAPCAR #'LIST '(1 2 3))
(MAPCAR #'LIST '(1 2 3) '(a b c))
(SETQ a 1 b 2 c 3)
(MAPC #'SET '(a b c) '(c a b))
b
a
(MAPLIST    #'APPEND    '(1 2 3)  '(a b c) )
```

2. Se vor discuta şi executa funcțiile prezentate în cadrul surselor.

3. Implementați o versiune a funcției MEMBER-IF. MEMBER-IF acceptă ca parametrii un predicat unar şi o listă. Se întoarce sublista, dacă există, ce începe cu primul element ce satisface predicatul.

4. ÎNTREBĂRI SI PROBLEME

1. Scrieți o funcție care aplică o altă funcție pe atomii unei liste multinivel, întorcând o structură arborescentă izomorfă, dar în care frunzele (atomii) sunt înlocuiți cu rezultatul aplicării funcției asupra lor. Exemplu:

EXECUTIE	REZULTAT
*(map-leaf #'NUMBERP '(1 (2 x) (4)))	(T (T NIL) (T))

2. Implementați o versiune proprie pentru **MAPCAR.**

5. SURSE

```
;;; intoarce o copie a listei argument din care elimina toate elementele de pe primul nivel care satisfac testul

(DEFUN our-remove-if (test lis)
  (MAPCAN #'(LAMBDA (x)
   (IF (NOT (FUNCALL test x)) (LIST x)) )
        lis
))
```

```
;;; Operatii cu multimi
;;; multimea elementelor unei liste
;; varianta recursive

(DEFUN mkset0 (lis)
   (COND ((ENDP lis) NIL)
         ((MEMBER (CAR lis) (CDR lis))
              (mkset0 (CDR lis)))
         ((CONS         (CAR lis)
              (mkset0 (CDR lis)) ))
))
```

```
;;; aplica o functie succesiv pe elementele unei liste ca si MAPCAR pentru o functie de un argument, dar colecteaza doar rezultatele non-nil

(DEFUN mapcarn (fun lis)
(MAPCAN #'(LAMBDA (x)
                (IF (SETF x (FUNCALL fun x))
                        (LIST x) ))
        lis
))
```

```
;; iteratie cu "MAPLIST"

(DEFUN mkset1 (lis)
(APPLY  #'APPEND
          (MAPLIST    #'(LAMBDA (x)
                                (IF  (MEMBER  (CAR  x)
(CDR x))
                                     NIL
                                     (LIST (CAR x))))
                        lis)
))
```

```
;;; lungimea maxima a sublistelor unei liste
;; varianta recursiva

(DEFUN lgm1 (l)
(IF (ATOM l)
        0
        (MAX   (LENGTH l)
               (lgm1 (CAR l))
               (lgm1 (CDR l)))
))
```

```
;; iteratie cu "MAPCON"

(DEFUN mkset2 (lis)
(MAPCON #'(LAMBDA (x)
          (IF (MEMBER (CAR x) (CDR x))
            NIL
            (LIST (CAR x))))
          lis)
)
```

;; iteratie cu **"DO"**

```
(DEFUN lgm2 (l)
(DO (   (rez (LENGTH l))
        (ll l (REST ll)) )
        ((ENDP ll) rez)
        (IF (LISTP (FIRST ll))
          (SETF rez (max rez (lgm2 (FIRST ll))))
        )
))
```

;; **reuniunea a doua mulțimi** folosind **"MAPCON"**

```
(DEFUN reuniune1 (set1 set2)
(MAPCON #'(LAMBDA (x)
                (APPEND
                (IF (NOT (MEMBER (CAR x) set2))
                     (LIST (CAR x)))
                (IF (NULL (CDR x)) set2)))
                set1)
)
```

;; iteratie cu **"MAPCAR"**

```
(DEFUN lgm3 (l)
(IF (ATOM l)
        0
        (MAX (LENGTH l) (APPLY #'MAX
(MAPCAR #'lgm3 l)))
))
```

;; iteratie cu **"MAPC"**

```
(DEFUN reuniune2 (set1 set2)
(LET ((rez set2))
        (MAPC #'(LAMBDA (x)
        (IF (NOT (MEMBER x set2))
                (SETF rez (CONS x rez)) ) )
                set1)
        rez)
)
```

;;; **numãrul de apariții**, pe orice nivel, **ale unui atom intr-o listã**

```
(DEFUN aparitii (elem lista)
  (COND ((EQL elem lista) 1)
     ((ATOM lista) 0)
     ((APPLY #'+   (MAPCAR #'(LAMBDA (x)
                 (aparitii elem x) )
        lista)))
))
```

;; **intersectie a doua multimi** folosind **"MAPCARN"**

```
(DEFUN inters1 (set1 set2)
(mapcarn #'(LAMBDA (x)
                (IF (MEMBER x set2) x) )
         set1)
)
```

;; **iteratie cu "MAPCON"**

```
(DEFUN inters2 (set1 set2)
(MAPCON #'(LAMBDA (x)
  (IF (MEMBER (FIRST x) set2) (LIST (FIRST x))))
        set1)
)
```

;;; **numarul de atomi dintr-o lista**

```
(DEFUN nratoms (l)
(COND((NULL l) 0)
        ((ATOM l) 1)
        (T (APPLY #'+ (MAPCAR #'nratoms l)))
))
```

;;; **multimea partilor unei multimi**

```
(DEFUN m-parti (set)
(IF (NULL set)  '(())
        (extinde-cu     (FIRST set)
                        (m-parti (REST set)))
))
(DEFUN extinde-cu (elem set-seturi)
(APPEND set-seturi
    (MAPCAR  #'(LAMBDA (x) (CONS elem x) )
                set-seturi)
))
```

;;; **eliminarea parantezelor interioare unei liste**

```
(DEFUN strivire1 (x)
  (COND ( (NULL x) NIL)   ((ATOM x) (LIST x) )
   (T (APPLY #'APPEND
                (MAPCAR #'strivire1 x)))
  )
)
(DEFUN strivire2 (x)
  (COND ((NULL x) NIL) ((ATOM x) (LIST x) )
        (T (MAPCAN #'strivire2  x) )
  )
)
```

;; intoarce **multimea de liste** ce rezulta prin inserarea unui element in toate pozitiile unei liste

```
(DEFUN pune-peste-tot (elem lis)
(LET ((fata) (aux))
  (CONS (APPEND lis (LIST elem))
        (MAPLIST #'(LAMBDA (x)
         (SETQ aux (APPEND fata (LIST elem) x))
         (SETQ fata (APPEND fata (LIST (CAR x))))
                aux)
           lis)
        )
))
```

;;; **inversarea elementelor dintr-o lista** inclusiv a celor de pe nivelurile interioare

```
(DEFUN rev-all (lis)
    (REVERSE
     (MAPCAR #'(LAMBDA (x)
        (IF (ATOM x) x (rev-all x) ) )
                          lis)
))
```

;;; multimea **permutari**lor de elemente ale unei liste

```
(DEFUN perm (lis)
   (COND ((NULL lis) '(()) )
      (T (MAPCAN #'(LAMBDA (x) (pune-peste-tot (CAR lis) x))
                 (perm (CDR lis))))
))
```

;;; multimea **combinari**lor de "n" elem. dintr-o lista

```
(DEFUN comb (n lis)
 (COND ((= n (LENGTH lis)) (LIST lis))
     ((ZEROP n) '(NIL) )
     ((APPEND  (comb n (REST lis) )
      (MAPCAR #'(LAMBDA (x) (CONS (FIRST lis) x))
         (comb (- n 1) (REST lis)))
        ))
))
```

;;; multimea **aranjamente**lor de "n" elem.din lista

```
(DEFUN aranj (n lis)   (MAPCAN  #'perm  (comb n lis))
)
```