PROGRAMARE FUNCȚIONALĂ

Lucrarea 4

FORME FUNȚIONALE

Există cazuri în care numărul parametrilor unei funcţii trebuie stabilit dinamic. Aplicarea unei funcţii asupra unei mulţimi de parametri sintetizată eventual dinamic este posibilă cu ajutorul funcţiilor **APPLY** şi **FUNCALL**.

APPLY cheamă o funcţie asupra unei liste de argumente.

Sintaxă:

(apply functie (listaparametri))

Exemple:

```
> (apply 'cons  '(a b))
(A . B)
> (apply 'cons  '(a (1 2 3)))
(A 1 2 3)
> ( apply ’max  ’ ( 1 2 3 4 5 6 ) )
6
> ( apply ’+  ’ ( 1 2 3 4 ) )
10
```

FUNCALL - este o variantă a funcţiei apply care permite aplicarea unei funcţii la un număr fix de parametri.

Sintaxă:

(funcall functie arg1 arg2 ...)

Exemple:

```
> (funcall 'cons 'a 'b)
(A . B)
> (funcall 'cons 'a '(1 2 3))
(A 1 2 3)
 > ( funcall ’max  1 2 3 4 5 6 )
6
> (setq p 'car)
CAR
> ((eval p) '(a b c))
Error: bad function - (EVAL P)
Happened in: #<Subr-TOP-LEVEL-LOOP: #12a27d8>
```

Devine corect dacă utilizăm funcall sau apply:

```
> (funcall p '(a b c))
A
> (apply p '((a b c)))
A
```

Expresii **LAMBDA** apar:

- când o funcţie e folosită o singură dată şi e mult prea simplă ca să merite a fi definită;
- funcţia de aplicat trebuie sintetizată dinamic (fiind imposibil să fie definită cu DEFUN).

Lambda defineşte o funcţie anonimă şi se aseamană cu DEFUN doar că nu dă nume funcţiei definite, putând fi folosită doar în definiţii locale.

Sintaxă:

(lambda l f1 f2 f3 ... fn) sau ((lambda l f1 f2 f3 ... fn) par1 par2 ... parn) - defineşte o funcţie utilizată local;

- l reprezintă lista parametrilor; (poate fi dată explicit (număr fix de parametri) sau parametric (lista l având un număr variabil de parametri);

- f1 f2 ... fn corpul funcţiei; Argumentele sunt evaluate la apel

Exemple:

```
> ((lambda () 2 0))
0
> ( ( lambda ( x ) (1+ x ) ) 5 )
6
> ( ( lambda ( x y ) (+ x y ) ) 5 7 )
12
```

MAPCAR – se foloseşte atunci când dorim să aplicăm o funcţie la fiecare element al unei liste, şi returnează lista rezultatelor.

Sintaxă:

(Mapcar <nume-functie> >lista de argumente>)

Exemple:

```
> (mapcar 'car '((1 2 3) (3 4 5) (5 6 7)))
(1 3 5)
> (mapcar 'length '((1 2 3) (4 5) (a)))
(3 2 1)
```

MAPCAN – se comportă ca şi MAPCAR combinat cu APPEND

Sintaxă:

(Mapcan <nume functie> <lista de argumente>) ==(apply ‘append(mapcar <nume functie> <lista de argumente>))

Exemple:

```
> (mapcan 'cdr '((1 2 3) (3 4 5) (5 6 7)))
(2 3 4 5 6 7)
> (mapcan 'last '((a b) (c d ) (e f)))
(B D F)
```

REMOVE-IF-NOT – se comportă asemănător cu MAPCAR, dar păstrează în rezultatul final doar elementele pentru care un anumit predicat este adevărat.

Sintaxă:

(Remove-if-not <nume functie> <lista de argumente>)

Exemplu:

```
> (remove-if-not 'evenp '(2 3 4 5 6 8))
(2 4 6 8)
```

Exemple: -apeluri combinate

1. Fie o lista l > (setq l '(2 4 34 12 8))
   (2 4 34 12 8)

Definim funcţia DIV4 care returnează t dacă un număr x este divizibil cu 4 :

```
> (defun div4 (x)
(zerop (rem x 4)))
DIV4
```

Folosind MAPCAR cu DIV4 putem verifica elementele listei dacă sunt divizibile cu 4:

```
> (mapcar 'div4 l)
(NIL T NIL T T)
```

Dacă funcţia DIV4 nu o mai folosim în altă parte, putem să o definim chiar în locul în care apare:

```
> (mapcar (defun div4(x) (zerop (rem x 4))) l)
(NIL T NIL T T)
```

MAPCAR aplică asupra listei L aceeaşi funcţie DIV4, definită de această dată chiar la apel. Se poate concluziona că practic numele funcţiei nu este util dacă funcţia apare doar în acest loc. Deci, putem folosi o funcţie locală anonimă Lambda.

```
> (mapcar '(lambda (x)  (zerop (rem x 4)))  l)
(NIL T NIL T T)
```

2. Se calculează suma elementelor divizibile cu 4 ale unei liste:

```
>(setq lista '(12 36 11 7 56 81))
(12 36 11 7 56 81)
>(defun suma(l)
(apply '+ (remove-if-not '(lambda (x)  (zerop (rem x 4)))  l)))
SUMA
> (suma lista)
104
```

3. Considerăm o listă generalizată pentru care numărăm elementele divizibile cu un număr dat:

```
> (defun numar-div (l n)
(cond ((atom l) (cond ((zerop (rem l n)) 1) (t 0)))
(t (apply '+ (mapcar '(lambda(x) (numar-div x n)) l)))))
NUMAR-DIV
> (numar-div l 5)
4
```

Funcţii pentru tipărire

PRINT – primeşte un singur argument pe care îl evaluează şi îl tipăreşe pe o linie nouă punând un spaţiu la sfârşit:

```
> (setq l '6)          > (print l)
 6                     6
                       6
```

După afişare, PRINT returnează pe a doua linie valoarea argumentului. Câteodată, este util să avem simboluri ce conţin caractere special pe care urmează să le afişăm de mai multe ori într-o funcţie (Ex. spaţiu, paranteze, virgule, etc.). Acest lucru se obţine prin atribuirea cu setq unui simbol, caracterul dorit cuprins între bare verticale:

```
> (setq par '|{|)> (setq space '| |)        > (print par)
{                | |                       {
                                             {
```

PRIN1 - primeşte un singur argument pe care îl evaluează şi îl tipăreşe pe aceeaşi linie şi nu pune spaţiu la sfârşit. Returnează pe a doua linie, valoarea argumentului.

PRINC - primeşte un singur argument pe care îl evaluează şi îl tipăreşe pe aceeaşi linie şi nu pune spaţiu la sfârşit, nu afişează nici o bară verticală dacă există.

```
> (setq sym '|un simbol|)     > (print sym)     > (prin1 sym)       > (princ sym)
|un simbol|                   |un simbol|       |un simbol|          un simbol
                              |un simbol|       |un simbol|         |un simbol|
```

TERPRI - trece pe rând nou

```
> (setq pard '|{|)                    > (setq par '|}|)
{                              }
> (setq space '| |)
> (defun fct (s)
(princ pard)
(princ space)
(princ s)
(terpri)
(princ par))
FCT
> (fct 'a)
{ A
}
}
> (defun fct (s)
(princ pard)
(princ space)
(princ s)
(princ par))
FCT
> (fct 'a)
{ A}
}
```

Exemple:

1. Definim o funcţie care afişează o listă cu elemente atomi pe o linie:

```
> (setq pard '|{|)      > (setq par '|}|)      > (setq space '| |)
{                          }                  | |
> (defun print-lista(l)
(princ pard)
(do((lst l (cdr lst)))
((null lst) (princ par))
(princ (car lst))(princ space)))
PRINT-LISTA
> (print-lista '(a b c d e f))
{A B C D E F }
}
```

Funcţia afişează lista pe linie şi returnează valoarea argumentului ultimului PRINC.

2. Definim funcţia care afişează lista cu elementele pe linii diferite. Se doreşte ca după paranteză deschisă să nu se treacă pe rândul următor, iar după ultimul element din listă să se afişeze imediat paranteza închisă:

```
> (defun print1-lista(l)
(princ pard)
(do((lst l (cdr lst)))
((null (cdr lst))(princ (car lst))(princ par))
(princ (car lst))(terpri)))
PRINT1-LISTA
> (print1-lista '(a b c d e f))
{A
```

B
C
D
E
F}
}

LISTE DE ASOCIAŢII

Elementele unei liste de asociaţie sunt celule cons în care părţile aflate în CAR se numesc chei şi cele aflate în CDR se numesc date. Pentru a introduce şi pentru a extrage noi elemente, de regulă, operăm asupra unui capăt al listei, comportamentul este analog stivelor. Într-o astfel de structură, introducerea unei noi perechi cheie-dată cu o cheie identică uneia deja existentă are semnificaţia "umbririi" asociaţiei vechi, după cum eliminarea ei poate să însemne revenirea la asociaţia anterioară.

Pe acest comportament se bazează, de exemplu, "legarea" variabilelor la valori.

O listă de asociaţii este deci o listă formată din subliste de forma (<cheie> <valoare>).

Următoarea expresie crează o listă de asociaţii ce descrie proprietăţile unui anume obiect:

> (setq telefon '((marca samsung) (culoare alb) (tip galaxyS7)))
((MARCA SAMSUNG) (CULOARE ALB) (TIP GALAXYS7))

ASSOC – returnează valoarea unei chei dintr-o listă de asociaţii.Are două argumente: o cheie şi o listă de asociaţii. ASSOC va căuta asociaţia cu cheia dată şi va returna întreaga asociaţie (<cheie> <valoare>), iar dacă nu găseşte cheia, va returna NIL.

> (assoc 'marca telefon)
(MARCA SAMSUNG)
> (assoc 'culoare telefon)
(CULOARE ALB)

O listă de asociaţii poate conţine două elemente cu aceeaşi cheie, dar ASSOC va returna doar prima asociaţie validă găsită.

> (setq telefon '((marca samsung) (culoare alb) (culoare negru) (tip galazyS7)))
((MARCA SAMSUNG) (CULOARE ALB) (CULOARE NEGRU) (TIP GALAZYS7))
> (assoc 'culoare telefon)
(CULOARE ALB)

LISTE DE PROPRIETĂȚI

Unui simbol i se poate asocia o listă de perechi proprietate- valoare, în care proprietăţile sunt simboluri, iar valorile sunt date Lisp. În această listă o proprietate poate să apară o singură dată. O listă de proprietăţi are asemănări cu o listă de asociaţie (astfel numele de proprietate corespunde cheii, iar valoarea proprietăţii corespunde datei), dar există şi diferenţe între ele (în lista de proprietăţi o singură valoare poate fi atribuită unei proprietăţi, pe când in lista de asociaţii, mai multe).

SETF – dă valori unei proprietăţi;

GET - cercetază valoarea unei proprietăţi, a unui simbol.

Exemplu:

> (setf (get 's 'p1) 'prop1)
PROP1

> (setf (get 's 'p2) 'prop2)
PROP2
> (get 's 'p1)
PROP1
> (get 's 'p2)
PROP2
> (setf (get 'program 'limbaj) '(lisp prolog))
(LISP PROLOG)
> (get 'program 'limbaj)
(LISP PROLOG)

Putem șterge o proprietate astfel: (setf (get <simbol> <proprietate>) nil)

FUNCȚII DISTRUCTIVE

Funcţiile distructive (chirurgicale) îşi justifică numele prin faptul că realizează modificări asupra argumentele. Ele sunt aşadar funcţii în care efectul lateral este cel care primează.

Funcția **NCONC** modifică toate argumentele (fiecare de tip listă) cu excepţia ultimului, realizând o listă din toate elementele listelor componente.

NCONC face exact ceea ce am fost tentaţi să credem că face APPEND. Concatenează două liste modificând ultima celulă a primei liste şi nu prin copiere. Funcția poate avea oricâte argumente,alterând sfârşitul fiecărei liste în afară de ultima.

> (setq x '(a b c))
(A B C)
> (setq y '(1 2 3))
(1 2 3)
> (setq z '(u v))
(U V)
> (nconc x y z)
(A B C 1 2 3 U V)
> X
(A B C 1 2 3 U V)
> y
(1 2 3 U V)
> z
(U V)

Funcţia **RPLACA** modifică car-ul celulei cons obţinută din evaluarea primului argument la valoarea celui de al doilea argument şi întoarce celula cons modificată.

Funcția are 2 argumente din care primul trebuie să fie o listă. Alterează această lista înlocuind conţinutul primei celule cu al doilea argument.

> (setq x ‘(a b c))
(A B C)
> (rplaca (cdr x) ‘d)
(D C)
> x
(A D C)

RPLACD este complementar, alterează restul listei (pointerul către celula următoare al primei celule).

> (setq x '(a b c))

(A B C)
> (rplacd x '(d))
(A D)
> x
(A D)

DELETE elimină aparițiile primului argument în primul nivel al celui de al doilea argument, dar fizic nu alterează lista inițială.

> (setq x'(a b c d))
(A B C D)
> (delete 'a x)
(B C D)
> x
(A B C D)

EQL compară două liste. Pentru EQL două liste sunt identice doar dacă sunt reprezentate în aceleași celule de memorie. Copiile nu sunt considerate identice de către EQL, dar sunt considerate egale de către EQUAL.

MACROU-uri

Un **macro** este în esență o funcție care definește o funcție. Evaluarea apelurilor de macro-uri este un proces în doi pași: în primul rând o expresie specificată în definiție este construită, apoi această expresie este evaluată. Primul pas – cel al construirii macro-expresiei se numește **macroexpandare**.

O definiție de macro, analog unei definiții de funcție, conține trei elemente: un simbol care dă numele macro-ului, o listă de parametri și corpul:

(**defmacro** simbol (lista parametri) (corp))

Într-un apel de funcție definită de utilizator, parametrii actuali sînt evaluați înainte ca parametrii formali din definiție să se lege la aceștia. Rezultă că o evaluare diferențiată a parametrilor nu poate fi realizată printr-o definiție de funcție. Toate formele Lisp-ului în care parametrii se evaluează diferențiat sînt realizate intern ca macro-uri.

În evaluarea unui macro: pasul macroexpandării operează cu expresii, cel al evaluării – cu valorile lor.

Exemplu:

Dacă ar fi să realizăm o funcție care să aibă comportamentul unui if, de exemplu, utilizând însăși forma if pentru aceasta, o definiție precum următoarea:

> (defun my-if(test expr-da expr-nu)
  (if test expr-da expr-nu))

nu satisface, pentru că la intrarea în funcție toți cei trei parametri sunt evaluați.

Definiția corectă ar fi:

> (defmacro my-if(test expr-da expr-nu)
`(if ,test ,expr-da ,expr-nu))
MY-IF
> (my-if t (setq x 'da) (setq x 'nu))
DA
> (my-if (evenp 7) t nil)
NIL

Apostroful-stânga (*backquote*):

Un apostrof-stânga (`) construieşte o formă Lisp conform modelului (*template*) care urmează după el. Sintactic, el prefixează o listă. La evaluare orice formă a listei va fi copiată, cu excepţia formelor prefixate de virgulă (,), care sunt evaluate.

```
> (setq b 'beta d 'gamma)
GAMMA
> b
BETA
> `(a ,b c ,d)
(A BETA C GAMMA)
```

Restricţii privind folosirea virgulei:

- virgula poate să apară numai în interiorul unei expresii prefixate cu apostrof-stânga.

Exemplul anterior ar putea fi scris şi astfel:

```
> (defmacro my-if (test expr-da expr-nu)
  (subst test 'test
  (subst expr-da 'expr-da
  (subst expr-nu 'expr-nu
  `(if test expr-da expr-nu)))))
MY-IF
> (my-if (> 8 7) (- 8 7) (+ 8 7))
1
```

VECTORI ŞI MATRICI

MAKE-ARRAY- crează un vector unidimensional sau o matrice:

```
> (setq v (make-array 5))
#(NIL NIL NIL NIL NIL)
Dimensiunea vectorului este un întreg.
> (setq m (make-array '(3 3)))
#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
Dimensiunea matricii este o listă de 2 elemente: nr. linii, nr. coloane
```

AREF - întoarce valoarea unui element de pe o anumită poziţie:

```
> (aref v 1)          > (aref m 1 2)
NIL                   NIL
```

Pentru a da valori elementelor unui vector sau matrice, identificăm locaţia elementului cu ajutorul lui AREF, iar conţinutul îl modificăm cu SETF.

```
> (setf (aref v 0) '9)      > (setf (aref m 0 0) 1)
9                           1
> (setf (aref v 1) '1)      > (setf (aref m 1 2) 9)
1                           9
> v                         > m
#(9 1 NIL NIL NIL)          #2A((1 NIL NIL) (NIL NIL 9) (NIL NIL NIL))
```

În LISP locaţiile unei matrici pot conţine nu doar numere ci expresii arbitrare. Două locaţii diferite ale unei matrici pot conţine expresii de tipuri diferite.

ARRAY-DIMENSION - returnează domeniul unui indice.

```
> (array-dimension v 0)
```

5
> (array-dimension m 0) > (array-dimension m 1)
3 3
Funcţia **ARRAY-DIMENSIONS** returnează domeniile tuturor indicilor într-o listă:
> (array-dimensions v) > (array-dimensions m)
(5) (3 3)

O altă modalitate de a reprezenta o matrice, este ca o listă de liste, astfel:
((e1 e2 e3) (e4 e5 e6) (e7 e8 e9)) - în acest caz fiecare sublistă conţine elementele de pe o linie a matricii, deci dimensiunea matricii este de 3x3.
Sau
($(I_0\ j_0\ e1)$ $(i_0\ j_1\ e1)$ … $(i_n\ j_m\ ek)$) - în acest caz fiecare sublistă are 3 elemente, iar elementele sunt: I – indicele liniei, j –indicele coloanei, e – elementul corespunzător locaţiei respective.

STRUCTURI

Definirea unei structuri se face cu macroul **DEFSTRUCT.**
Exemplu: -definim o structura pentru un punct de coordonate X Y:
> (defstruct punct x y)
PUNCT

În mod automat, implicit au fost definite funcţiile MAKE-PUNCT, PUNCT-P, COPY-PUNCT, PUNCT-X şi PUNCT-Y. Pentru a crea un punct nou, apelăm funcţia MAKE-PUNCT. Pentru a da valori câmpurilor, utilizăm ca argumente cuvintele cheie date de denumirea câmpurilor structurii:
> (setf p (make-punct :x 1 :y 2))
#S(PUNCT X 1 Y 2)
> (punct-x p) > (punct-y p)
1 2
> (setf (punct-y p) 3) > p
3 #S(PUNCT X 1 Y 3)

Notaţia **#S** este modalitatea standard de afişare a structurilor. Când definim o structură, este definit şi un tip cu acelaşi nume:
> (punct-p p)
T
> (typep p 'punct)
T

PROBLEME

1.Implementaţi o funcţie care adună toate elementele unei liste, aflate pe poziţii pare. Poziţiile se consideră începând cu 0.

2.Implementaţi recursiv reuniunea,intersecţia şi diferenţa dintre 2 mulţimi folosind remove-if-not.

3.Definiţi o funcţie care numără apariţiile unui număr dat într-o listă generalizată, folosind mapcar şi lambda.

4.Definiţi o funcţie care afişează o listă generalizată sub forma precizată pentru exemplul următor:

(a b (c d) (e f g) h) →

```
( a
  b
   ( c
     d  )
       ( e
         f
         g )
  h )
```

5.Scrieţi o funcţie FETCH care primeşte o cheie şi o listă de asociaţii, iar dacă găseşte cheia va returna doar valoarea ei. Dacă nu o găseşte va returna un mesaj corespunzător.

```
> (fetch 'temperature '((temperature 100)
        (pressure (120 60))
        (pulse 72)))
100
```

6.Scrieţi o funcţie ce returnează o listă cu toate cheile dintr-o listă de asociere.

7.Presupunând că avem simboluri ce au proprietatea **tata**. Definiţi procedura bunic ce primeşte un simbol şi returnează bunicul din partea tatălui dacă este cunoscut, alftel nil.

8.Definiţi funcţia recursivă ADAM care primeşte un simbol şi returnează cel mai îndepărtat strămoş pe linie paternă.

9.Definiţi funcţia recursivă STRAMOSI care returnează o listă formată din persoana împreuna cu toţi strămoşii cunoscuţi, mergând pe două proprietatăti: tata şi mama.

10.Definiţi un macro define care defineşte o funcţie având sintaxa:
(define <nume functie> (<param 1> ... <param n>) <corp>)

11.Definiţi un macro dotimes cu sintaxa:
(dotimes (<var> <count> <rezultat> <corp>)
Prima dată se evaluează <count> care trebuie să fie un întreg. Apoi <var> este succesiv legata la întregi de la 0 la valoarea lui <count> minus 1. Corpul este evaluat de fiecare dată, iar <rezultat> este returnat la sfârşit.

12.Scrieţi o funcţie PRINT-ARRAY care afişează o matrice de două dimensiuni primită ca argument. Fiecare rând al matricii va fi afişat pe o linie nouă. Folosiţi ARRAY-DIMENSION pentru a afla domeniul indicilor.

13.Scrieţi câte o funcţie care transformă o matrice reprezentată ca listă de liste, în ambele moduri prezentate, într-o matrice standard LISP. Determinați apoi numărul elementelor pare ale matricii.

14.Definiţi o structură de tip nod cu elementele: rădăcina stga dr;
Implementaţi apoi o funcţie care adaugă un nou nod, după regula: în stânga pentru valorile <= decât rădăcina, sau în dreapta pentru valori mai mari decât rădăcina. Creaţi un arbore .