Programare Funcționala

Lucrarea 3

TEHNICI DE PROGRAMARE

Definiții de funcții

În Lisp definiția de funcții se face cu **DEFUN**:

(DEFUN nume_fct (param_1 param_2 …param_n) (corpul_fct)) → nume_fct

Rezultatul unei definiții de funcții este rezultatul unei legări între un nume, recunoscut global – *nume_fct* , o listă de variabile, considerate variabile formale ale funcției (param_1 param_2...param_n), şi un corp al funcției compus dintr-o secvență de instrucțiuni. Funcția astfel definită va returna numele funcției.

DEFUN nu îşi evaluează argumentele, ci doar defineşte o funcție care va putea fi mai târziu utilizată. Numele funcției trebuie să fie un simbol. Lista de variabile de după numele procedurii se numeşte listă de parametrii. Parametrii sunt de asemenea simboluri, ce reprezintă atomi sau liste. Când o funcție va fi apelată împreună cu argumentele sale într-o expresie simbolică, valoarea fiecărui parametru din procedură va fi determinat de valoarea agrumentului corespunzator din expresie.

Exemplu:

```
> (defun prim_elem(lista)
 (car lista))
PRIM_ELEM
> (prim_elem  '(a b c d e f))
A
Sau
> (setq L '(1 2 3 4))
(1 2 3 4)
> (prim_elem L)
1
```

Cu DEFUN am definit funcţia *prim_elem*. Apoi pentru a vedea dacă această funcție a fost definită corect, apelăm funcția într-o expresie simbolică, prin numele ei şi lista de parametri, care în acest caz este formată dintr-un singur parametru, parametru care este o listă. DEFUN va returna numele funcţiei ce tocmai a fost definită, dar partea utilă se realizează prin efecte laterale. La apel valoarea argumentului devine valoarea temporară a parametrului procedurii.

Procesul de pregătire a spaţiului de memorie pentru valorile parametrilor se numeşte **LEGARE**. Parametrii sunt legaţi la apelul funcțiilor.

Procesul de stabilire a unei valori se numeşte **ATRIBUIRE**. LISP întotdeauna atribuie valori parametrilor imediat după legare.

Exemple:

1. Să se definească o funcție ce returnează aria unui dreptunghi.

Rezolvare:

```
 (defun aria (lung lat)    (* lung lat))
```

ARIA
(aria '4 '5)
20
Sau
> (setq lu '5)
5
> (setq la '3)
3
> (aria lu la)
15

2. Să se definească funcția care returnează ultima cifră a unui număr întreg.

Rezolvare:

```
> (defun cifra (numar) (rem numar 10))
CIFRA
> (cifra '45621)
```

3. Să se definească funcția rotate-left, care roteşte la stânga cu o poziție elementele unei liste. Lista care va fi returnată este lista în care primul element al listei inițiale este mutat la sfârşitul listei.

Rezolvare:

```
> (defun rotate-left (lista)
    (append (cdr lista) (list (car lista))))
ROTATE-LEFT
> (rotate-left '(a b c d e))
(B C D E A)
```

4. Sa se definească o funcție care determină elementul minim dintre trei elemente numerice.

Rezolvare:

```
> (defun minim (a b c)
(cond ((< a b c) a) ((< b a c) b) ((< c a b) c)
    ((< a c b) a) ((< b c a) b) ((< c b a) c)))
MINIM
> (minim 9 3 7)
3
```

5. Definiți predicatul not-real care are ca parametri coeficienții ecuației de gradul 2 şi returnează t dacă delta este pozitiv şi nil dacă este negativ.

Rezolvare:

```
> (defun not-real (a b c)
(if (>= (- (* b b) (* 4 a c)) 0) t))
NOT-REAL
> (not-real 2 4 1)
T
> (not-real 5 1 2)
NIL
```

Forme pentru controlul evaluarii

Cele mai utilizate forme LISP pentru controlul evaluării sunt: **IF** şi **COND**.
IF poate avea doi sau trei parametri. În varianta cu trei parametri:
(if e_1 e_2 e_3)

unde dacă e_1 este diferit de nil atunci rezultatul este e_2, altfel rezultatul este e_3

În varianta cu doi parametri:
(if e_1 e_2)
unde dacă e_1 este diferit de nil atunci rezultatul este e_2, altfel nil.

Exemple:

```
 (setq a '8)                  > (setq b '4)
8                             4
> (setq c (rem a b))          > (setq  x  '9)
0                             9
> (setq y '2)
2
> (if (zerop c) "numere divizibile" (floor (/ a b)))
"numere divizibile"
> (if (zerop (rem x y)) "numere divizibile" (floor (/ x y)))
4
0.5
> (if (not (= a 0)) a)                > (if (= a 0) a)
8                                     NIL
```

COND este cea mai utilizată formă de control a evaluării. Sintactic, ea este formată dintr-un număr oarecare de liste , fiecare conținând un test şi o expresie de evaluat. Fiecare listă se numeşte clauză. Elementele de pe prima poziție a clauzelor se evaluează în secvență până la primul care e diferit de nil.

În acest moment celelalte elemente ale clauzei se evaluează şi cond întoarce valoarea ultimului element din clauză. Dacă toate elementele de pe prima poziție din clauze se evaluează la nil, atunci cond însuşi întoarce nil. Dacă clauza conține doar testul, atunci se returnează valoarea testului.

Sintaxă:
(cond (test1 rezultat1) (test2 rezultat2)...(testn rezultatn))

Exemple:

```
> (setq a  8)                 > (setq b 4)
8                             4
> (cond  ((> a b) a)          > (cond ((> a b) b)
   ((< a b) b)                     ((< a b) a)
   (t "egalitate"))                (t "egalitate"))
8                             4
> (setq u '5)                  > (setq v '5)   > (setq lista ‘(a b c))
5                             5                 (A B C)

> (cond ((> u v) u)           > (cond ((not lista) "eroare")
    ((< u v) v)                     (t (caddr lista)))
     (t "egalitate"))         C
"egalitate"
```

```
> (cond (lista (caddr lista))
    (t "eroare"))
C
```

Cu toată simplitatea ei, forma **if** are un dezavantaj, şi anume faptul că nu permite evaluarea a mai mult decât o singură expresie înainte de ieşire, față de **cond**, care evaluează n expresii.

Legarea variabilelor

În LISP argumentele unei funcții sunt transmise prin valoare la fel ca în limbajul C. La intrarea în procedură, parametri sunt legați la argumente devenind **variabile legate**, iar la ieşirea din procedură, variabilele legate primesc vechile valori. Variabilele folosite într-o funcție, dar care nu sunt parametri se numesc **variabile libere** în raport cu acea procedură.

Spre deosebire de SETQ, **LET** leagă variabile şi le dă valori, pe când SETQ dă numai valori.

Sintaxă:

(**let** ((<variabila 1> <expresie 1>)
    (<variabila 2> <expresie 2>)
    …
    (<variabila n> <expresie n>))
    <corpul lui let>)

O expresie **LET** are două părți: în prima parte este o listă de instrunțiuni pentru crearea variabilelor, de forma: (variabilă expresie) în care fiecare variabilă va fi inițializată cu valoarea corespunzătoare expresiei.

Legarea este valabilă doar în interiorul corpului lui LET. În a doua parte, se află corpul lui LET ce poate conține mai multe expresii ce urmează a fi evaluate în ordine. Valoarea ultimei expresii este returnată ca valoare a lui LET.

Exemple:

```
> (let ((x 1) (y 2))  (+ x y))
3
```

În acest caz nu avem decât o singură expresie în corpul lui LET. Se crează cele două variabile x şi y inițializate cu valorile corespunzătoare 1 respectiv 2. Apoi se evaluează expresia din corp şi se returnează rezultatul.

```
>  (defun comanda ()
(format t "Introduceti un numar: ")
(let ((val (read)))
 (if (numberp val) val "eroare")))
COMANDA
>  (comanda)
Introduceti un numar: u
"eroare"
> (comanda)
Introduceti un numar: 45
45
```

Această funcție afişează un mesaj, setează variabila *val* la valoarea care urmează a fi citită, iar în corp verifică dacă *val* este un număr sau nu. Dacă este număr returnează numărul citit, dacă nu, returnează un şir de caractere ce reprezintă un mesaj de eroare.

Recursivitate

Atunci când o funcție se apelează pe sine direct sau indirect, spunem că avem recursivitate. O strategie, care nu dă greş, de definire a funcțiilor recursive ar fi:

- începe prin a scrie condiția de oprire;
- scrie apoi apelul recursiv.

Exemple:

1. Funcția factorial

```
(defun factorial (n)  (if (zerop n) 1 (* n (factorial (- n 1)))))
FACTORIAL
> (factorial 3)
6
```

Sau

```
> (defun factorial (n)  (cond ((zerop n) 1)
          (t (* n (factorial (- n 1))))))
FACTORIAL
> (factorial 4)
24
```

2. Funcția exponențială:

```
> (defun expt (m n) (cond ((zerop n) 1)
          (t (* m (expt m (- n 1))))))
EXPT
> (expt 2 4)
16
```

3. Funcția reverse:

```
> (defun my-reverse (lista) (cond ((null lista) lista)
          (t (append (my-reverse (cdr lista)) (list (car lista))))))
MY-REVERSE
> (my-reverse '(a b c d e f))
(F E D C B A)
```

4. Funcția de adunare a elementelor unei liste:

```
> (defun suma1 (lst ac)
        (if (null lst) ac (suma1 (cdr lst) (+ ac (car lst)))))
SUMA1
> (suma1 '(1 2 3 4 5) 0)
15
```

ac – este o variabilă în care calculăm sumele intermediare.

Forme de iterare

Forma **DO** reprezintă maniera cea mai generală de a organiza o iterație în Lisp. Când dorim să executăm ceva în mod repetat, câteodată este mult mai natural să folosim iterația decât recursivitatea. Ea permite utilizarea unui număr oarecare de variabile şi controlarea valorilor lor de la un pas al iterației la următorul.

Forma cea mai complexă a unui apel **DO** este:

(DO ((<param 1> <valoare initala 1> <actualizare 1>)

(<param 2> <valoare initala 2> <actualizare 2>)

(<param n> <valoare initala n> <actualizare n>))

(<test sfarsit> <rezultat>)

(<corpul lui DO>))

Primele n elemente ale unui DO sunt câte o listă ce definesc variabilele de control ale buclei, valorile lor inițiale şi actualizarea. Astfel, fiecărei variabile îi corespunde un nume, valoarea ei inițială și o formă de incrementare a acesteia.

Dacă expresia de inițializare e omisă, ea va fi implicit considerată NIL. Dacă expresia de incrementare este omisă, variabila nu va fi schimbată între paşii consecutivi ai iterației (deşi corpul lui DO poate modifica valorile variabilei prin setq). Înainte de prima iteraţie, toate formele de iniţializare sunt evaluate şi fiecare variabilă este legată la valoarea de iniţializare corespunzătoare (acestea sunt legări iar nu asignări, astfel încât după ieşirea din iteraţie, variabilele revin la valorile la care erau legate înainte de intrarea în iteraţie).

La începutul fiecărei iteraţii, după procesarea variabilelor, o expresie de test (test sfârşit) este evaluată. Dacă este nil, execuţia continuă cu evaluarea formelor din corpul DO-ului. Dacă testul este diferit de NIL se returnează rezultatul.

La începutul oricărei iteraţii, cu excepţia primei, variabilele sunt actualizate astfel: toate formele de incrementare sunt evaluate de la stânga la dreapta şi rezultatele reprezintă valorile la care sunt legate variabilele în paralel.

Exemple:

1. Funcţia Reverse:

```
> (defun list-reverse(lst)
(do((l lst (cdr l)) (x '() (cons (car l) x)))
((null l) x)))
LIST-REVERSE
```

Variabila de ciclu l se leagă la liste din ce în ce mai scurte din cea iniţială, lst, în timp ce variabila x pleacă de la lista vidă şi adaugă la fiecare iteraţie primul element al listei l. Când l ajunge la lista vidă do se termină întorcând valoarea acumulată în x.

Sau

```
> (defun list-reverse(lst)
(do((l lst (cdr l)) (x '() ))
((null l) x)
(setq x (cons (car l) x))))
LIST-REVERSE
```

În acest caz, legarea variabilei x cu noua valoare se face în corpul lui DO numai prin SETQ.

```
> (list-reverse '(a b c d e f))
(F E D C B A)
```

Corpul lui DO poate lipsi dacă reprezintă o formă de incrementare a unei variabile iniţializate în DO.

2. Funcţia exponenţială:

```
> (defun expt(m n)
(do((rez 1 (* m rez))
(exp n (- exp 1)))
((zerop exp) rez)))
EXPT


> (expt 3 5)
243
```

Forma **DOLIST** iterează aceleaşi evaluări asupra tuturor elementelor unei liste.

Exemplu:

```
> (dolist (x '(a b c d) 'Gata)
 (prin1 x)
 (princ " ")
 )
A B C D
GATA
```

Prelucrările care se execuța pentru fiecare element al listei sunt: tipărirea elementului şi a unui spațiu, la final se returnează mesajul GATA.

Forma **DOTIMES** iterează aceleaşi evaluări de un număr anumit de ori.

Exemplu:

```
> (dotimes (n 11)
   (print n) (prin1 (* n n))
)


0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
      NIL
```

Exemple:

1.Funcţia Reverse:

```
> (defun list-reverse(lst)
(do((l lst (cdr l)) (x '() (cons (car l) x)))
((null l) x)))
LIST-REVERSE
```

Sau
```
> (defun list-reverse(lst)
(do((l lst (cdr l)) (x '() ))
((null l) x)
(setq x (cons (car l) x))))
LIST-REVERSE
```

În acest caz, legarea variabilei x cu noua valoare se face în corpul lui DO numai prin SETQ.

```
> (list-reverse '(a b c d e f))
(F E D C B A)
```

Corpul lui DO poate lipsi dacă reprezintă o formă de incrementare a unei variabile inițializate în DO.

2. Funcţia exponenţială:

```
> (defun expt(m n)
(do((rez 1 (* m rez))
(exp n (- exp 1)))
((zerop exp) rez)))
EXPT
> (expt 3 5)
243
```

PROBLEME

1. Exprimați cu ajutorul lui **if** modulul unui număr şi operatorul **not**:
(abs x) (not y)

2. Exprimați cu ajutorul lui cond : and, or între 3 argumente:
(or x y z) (and u v w)

3. Determinați cu ajutorul lui cond elementul din mijloc dintre trei elemente numerice.

4. Definiți funcțiile *my-third, my-last,*care să returneze al trei-lea element al unei liste, respectiv ultimul element al unei liste.

5. Definiți o funcție *my-append* care primeşte ca argumente doi parametri de tip listă şi returnează o listă de lungime dublă obținută prin concatenarea celor două liste inițiale. Apoi definiți predicatul *palindrom* care determină dacă lista astfel obținută este sau nu palindrom.

6. Definiți o funcție care primeşte ca parametri coeficienții unei ecuații de gradul 2 şi care dacă delta este pozitiv returnează o listă cu soluțiile ecuației, dacă delta este egal cu 0, returnează soluția unică a ecuației de gradul 2, iar dacă delta este negativ, returnează mesajul “solutii complexe”.

7. Definiți o funcție care primeşte un singur argument x şi returnează valoarea unei funcții defintе astfel:

$$f(x)=\begin{cases} x^2; x \le -2 \\ x+2; -2 < x \le 1 \\ \sqrt{x^2+1}; x > 1 \end{cases}$$

8. Definiți o funcție care primeşte ca argumente un atom şi o listă, şi returnează o listă în care se adaugă la coada listei inițiale atomul, numai în cazul în care atomul nu se regăseşte în listă. In caz contrar, se returnează fragmentul de listă incepând cu elementul găsit.

9. Definiți o funcție care afişează un meniu interactiv, citeşte câte un caracter şi execută o operație conform caracterului citit, iar în cazul în care caracterul nu este nici unul dintre caracterele citite, returnează un mesaj de eroare.

1 – Adunare
2 – Scadere
3 – Inmulțire
4 – Restul împărțirii

Se considera trei variabile inițializate cu valori citite de la tastatură , primele două reprezintă două numere intregi, iar ultima, numărul corespunzător operației ce va fi executată. În corpul lui LET se execută operația corespunzătoare caracterului citit pentru cea de-a treia variabilă.

10. Să se implementeze recursiv o funcţie COUNTATOMS care primeşte ca argument o listă ce conţine şi liste imbricate şi returnează numărul total de elemente ale listei.

11. Să se implementeze funcțiile recursive *Rot-left* şi *Rot-right* care rotesc la stânga, respectiv la dreapta n elemente ale unei liste. Rotire la stânga înseamnă mutarea primului element al listei la coada listei. Rotire la dreapta înseamnă mutarea ultimului element al listei în fața listei.

12. Scrieţi o funcţie *fără _dubluri (lista)* care returnează lista fără dubluri. Elementele să fie în aceeaşi ordine în lista returnată ca şi în lista iniţială, păstrându se ultima apariţie a elementelor duplicat.
Exemplu: > (f a r a _d u b l u r i ' (1 2 2 3 4 5 6 4 4)) → (1 2 3 5 6 4)

13. Implementați recursiv predicatul *presentp* care determină dacă un atom apare oriunde în interiorul unei liste, chiar şi o listă ce conține liste imbricate.

14. Implementați recursiv o funcție care primeşte ca argument un număr întreg şi returnează lista cu cifrele numărului.

15. Implementați iterativ funcția *pozpar*, care primeşte ca argument o listă şi returnează o listă cu elmentele de pe pozițiile impare ale listei inițiale.

16. Implementați iterativ o funcție, care determină cel mai mare divizor comun a două numere întregi. Evidenţiaţi parametri actuali şi valoarea rezultată a funcţiei apelate, în ordinea apelurilor, respectiv terminărilor funcţiei pentru un exemplu concret.