

## CURS 12

### Generatori. Argumente opționale. OBLIST și ALIST. Macrodefiniții. Apostroful invers. Forme iterative

#### Cuprins

1. Generatori .....	1
2. Argumente opționale.....	3
3. OBLIST și ALIST.....	6
4. Macrodefiniții .....	7
5. Apostroful invers (backquote) .....	8
6. Forme iterative.....	9
7. Parametrii Lisp în context “Dynamic Scoping”.....	11
8. Universul expresiilor și universul instrucțiunilor .....	13
9. Independența ordinii de evaluare .....	15

#### 1. Generatori

Ca exemplu sugestiv pentru această problemă să considerăm exemplul unei funcții **VERIF** care primește o listă de liste și întoarce T dacă toate sublistele de la nivel superficial sunt liniare și NIL în caz contrar.

(VERIF '((1 2) (a (b)))) va produce NIL

(VERIF '((1 2) (a b))) va produce T

Vom folosi două funcții

- Funcția **LIN**, care verifică dacă o listă este liniară

Model recursiv

$$\text{LIN}(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{daca } l_1 \text{ nu e atom} \\ \text{LIN}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN LIN (L)
  (COND
    ((NULL L) T)
    (T (AND (ATOM (CAR L)) (LIN (CDR L) )))
  )
)
```

- Funcția **VERIF**, care verifică dacă o listă de liste are toate sublistele de la nivel superficial liste liniare.

Model recursiv

$$\text{VERIF}(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } \neg \text{LIN}(l_1) \\ \text{VERIF}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN VERIF (L)
  (COND
    ((NULL L) T)
    (T (AND (LIN (CAR L)) (VERIF (CDR L) )))
  )
)
```

Să observăm că funcțiile LIN și VERIF de mai sus au aceeași structură, conformă șablonului

```
(DEFUN F (L)
  (COND
    ((NULL L) T)
    (T (AND (F1 (CAR L)) (F (CDR L))))
  )
)
```

cu F1 = LIN în cazul F = VERIF și F1 = ATOM în cazul F = LIN. O astfel de structură este des folosită, “rețeta” ei de lucru fiind: elementele unei liste L sunt transmise pe rând (în ordinea apariției lor în listă) unei funcții F (în cazul de mai sus F1) care le va prelucra. Dacă F întoarce NIL acțiunea se încheie, rezultatul final fiind NIL. Altfel, parcurgerea continuă până la epuizarea tuturor elementelor, caz în care rezultatul final este T. Cum această rețetă se poate aplica pentru orice funcție F ce realizează prelucrarea într-un anume fel a tuturor elementelor unei liste, apare ideea de a scrie o funcție **generică** (șablon) GEN ce respecta “rețeta”, având doi parametri: funcția ce va realiza prelucrarea și lista asupra căreia se va acționa.

Terminologia adoptată pentru astfel de funcții (sau similare) este cea de **generatori**. Nu este necesar ca un generator să întoarcă T sau NIL. În funcție de implementare se pot concepe generatori care să întoarcă, de exemplu, lista tuturor rezultatelor non-NIL ale lui F culese până în momentul încetării acțiunii generatorului (L este vidă sau F întoarce NIL).

Model recursiv

$$\text{GEN}(F, l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } \neg F(l_1) \\ \text{GEN}(F, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```

(DEFUN GEN (F L)
  (COND
    ((NULL L) T)
    (T (AND (FUNCALL F (CAR L)) (GEN F (CDR L))))
  )
)

```

După ce a fost definit generatorul, se poate defini funcția VERIF (L) astfel:

```

(DEFUN VERIF (L)
  (GEN #'(LAMBDA (L)
    (GEN #'ATOM L)
  )
  L
)
)

```

## 2. Argumente opționale

În lista parametrilor formali ai unei funcții putem folosi următoarele variabile: &OPTIONAL și &REST.

- &OPTIONAL - dacă apare în lista parametrilor formali atunci următorul parametru formal va lua ca valoare parametrul corespunzător din lista parametrilor actuali, respectiv NIL dacă parametrul actual nu există;
- &REST - dacă apare în lista parametrilor formali atunci următorul parametru formal va lua ca valoare lista parametrilor actuali rămași neatribuiți, respectiv NIL dacă nu mai există parametri actuali neatribuiți.

### Exemple

#### Exemplu 1

```

(defun f (x &optional y)
  (cond
    ((null y) x)
    (t (+ x y))
  )
)

```

- (f 1 2) → 3
- (f 3) → 3

## Exemplu 2

```
(defun g (x &rest y)
  (+ x (apply #' + y))
)
```

- $(g\ 1\ 2\ 3) \rightarrow 6$
- $(g\ 4\ 5) \rightarrow 9$
- $(g\ 3) \rightarrow 3$

## Exemplu 3

```
(DEFUN F (L1 &REST L2)
  (COND
    ((NULL (CAR L2)) NIL)
    (T (CONS (MAPCAR #'* L1 (CAR L2)) (F L1 (CADR L2))))
  )
)
```

- $(F\ '(1\ 2)\ '(3\ 4)\ '(5\ 6))$  se evaluează la  $((3\ 8)\ (5\ 12))$
- **Exemplu 4**

Să presupunem că dorim să construim o funcție care să adune 1 la valoarea unei expresii:

```
(DEFUN INC (NUMAR)
  (+ NUMAR 1)
)
```

Această funcție se va utiliza în felul următor:

- $(INC\ 10)$  se va evalua la 11

Sigur că, în măsura în care dorim, putem adăuga funcții asemănătoare și pentru alte valori de incrementare. Alternativ, putem să rescriem funcția INC astfel încât să accepte un al doilea parametru:

```
(DEFUN INC (NUMAR INCREMENT)
  (+ NUMAR INCREMENT)
)
```

În marea majoritate a cazurilor, totuși, vom avea nevoie de incrementarea cu 1 a valorii expresiei corespunzătoare lui NUMAR. În această situație putem folosi facilitatea argumentelor opționale. Iată în continuare definiția funcției INC cu un argument opțional:

```
(DEFUN INC (NUMAR &OPTIONAL INCREMENT)
  (COND
    ((NULL INCREMENT) (+ NUMAR 1))
  )
)
```

```

        (T (+ NUMAR INCREMENT))
    )
)

```

Caracterul & din &OPTIONAL semnaleză faptul că &OPTIONAL este un separator de parametri, nu un parametru propriu-zis. Parametrii care urmează după &OPTIONAL sunt legați la intrarea în procedură la fel ca și ceilalți parametri. Dacă nu există parametru actual corespunzător, valoarea unui parametru formal opțional este considerată NIL. Iată câteva exemple de utilizare a funcției INC pe care tocmai am construit-o:

- (INC 10) se evaluează la 11
- (INC 10 3) se evaluează la 13
- (INC (\* 10 2) 5) se evaluează la 25

Putem, de asemenea, să luăm în considerare o valoare implicită pentru parametrii opționali. Iată o definiție a funcției INC în care parametrul opțional are valoarea implicită 1:

```

(DEFUN INC (NUMAR &OPTIONAL (INCREMENT 1))
  (+ NUMAR INCREMENT)
)

```

Observați că de această dată în locul parametrului opțional este precizată o listă formată din două elemente: primul element este numele parametrului opțional, iar al doilea element este valoarea cu care se inițializează acesta atunci când argumentul actual corespunzător nu este prezent.

Să notăm că putem avea oricâte argumente opționale. Toate aceste argumente vor fi trecute în lista parametrilor după simbolul &OPTIONAL, care va apare o singură dată. De asemenea, putem avea un singur argument opțional semnalat prin &REST, a cărui valoare devine lista tuturor argumentelor date cu excepția celor care corespund parametrilor obligatorii și opționali. Fie o nouă versiune a funcției INC:

```

(DEFUN INC (NUMAR &REST INCREMENT)
  (COND
    ((NULL INCREMENT) (+ NUMAR 1))
    (T (+ NUMAR (APPLY #'INCR INCREMENT))))
)

```

Iată câteva exemple de aplicare a acestei funcții:

- (INC 10) se evaluează la 11
- (INC 10 1 2 3) se evaluează la 16

### 3. OBLIST și ALIST

Un obiect Lisp este o structură alcătuită din:

- numele simbolului;
- valoarea sa;
- lista de proprietăți asociată simbolului.

Gestiunea simbolurilor folosite într-un program Lisp este realizată de sistem cu ajutorul unei table speciale numită lista obiectelor (**OBLIST**). Orice simbol este un obiect unic în sistem. Apariția unui nou simbol determină adăugarea lui la OBLIST. La fiecare întâlnire a unui atom, el este căutat în OBLIST. Dacă se găsește, se întoarce adresa lui.

Pentru implementarea mecanismului de apel, sistemul Lisp folosește o altă listă, numită lista argumentelor (**ALIST**). Fiecare element din **ALIST** este o pereche cu punct formată dintr-un parametru formal și argumentul asociat sau valoarea acestuia.

În general, o funcție definită cu  $n$  parametri  $P_1, P_2, \dots, P_n$  și apelată prin  $(F A_1 A_2 \dots A_n)$  va adăuga la ALIST  $n$  perechi de forma  $(P_i . A_i)$  dacă funcția își evaluează argumentele sau  $(P_i . A_i')$ , unde  $A_i'$  este valoarea argumentului  $A_i$ , dacă mediul Lisp evaluează argumentele. De exemplu, pentru funcția

```
(DEFUN F (A B)
  (COND
    ((ATOM B) A)
    (T (CONS A B))
  )
)
```

apelată cu  $(F 'X '(1.1))$ , vom avea pentru ALIST structura  $(\dots (A . X) (B . (1 . 1)))$ .

*Dacă la momentul apelului simbolurile reprezentând parametrii formali aveau deja valori, acestea sunt salvate în vederea restaurării ulterioare.*

Simbolurile sunt reprezentate în structură prin adresa lor din OBLIST, iar atomii numerici și cei șir de caractere prin valoarea lor direct în celula care îi conține.

Inițial, la începutul programului, ALIST este vidă. Pe măsura evaluării de noi funcții, se adaugă perechi la ALIST, iar la terminarea evaluării funcției, perechile create la apel se șterg. În corpul funcției în curs de evaluare valoarea unui simbol  $s$  este căutată întâi în perechile din ALIST începând dinspre vârf spre bază (această acțiune este cea care stabilește dinamic domeniul de vizibilitate). Dacă valoarea nu este găsită în ALIST se continuă căutarea în OBLIST. Așadar, este clar că **ALIST și OBLIST formează contextul curent al execuției unui program.**

#### Exemplu

- $(SETQ X 1)$  inițializează simbolul  $X$  cu valoarea 1;

- (SETQ Y 10) inițializează simbolul Y cu valoarea 10;
- (DEFUN DEC (X) (SETQ X (- X 1))) definește o funcție de decrementare a valorii parametrului;
- (DEC X) se evaluează la 0;
- X se evaluează tot la 1;
- (DEC Y) se evaluează la 9;
- Y se evaluează tot la 10.

*Să observăm deci că modificările operate asupra valorilor simbolurilor reprezentând argumentele formale se vor pierde după ieșirea din corpul funcției și revenirea în contextul apelator.*

## 4. Macrodefiniții

Din punct de vedere sintactic, macrodefinițiile se construiesc în același mod ca funcțiile, cu diferența că în loc să se folosească funcția DEFUN se va folosi funcția DEFMACRO:

**(DEFMACRO s l f1 ... f n): s**

Funcția DEFMACRO creează o macrodefiniție având ca nume primul argument (simbolul s), iar ca parametri formali elementele simboluri ale listei ce constituie al doilea argument; corpul macrodefiniției create este alcătuit din una sau mai multe forme aflate, ca argumente, pe a treia poziție și eventual pe următoarele. Valoarea întoarsă ca rezultat este numele macrocomenzii create. Funcția DEFMACRO nu-și evaluează niciun argument.

- evaluarea argumentelor face parte din procesul de evaluare al macrodefiniției

Modul de lucru al macrodefinițiilor create cu DEFMACRO este diferit de cel al funcțiilor create cu DEFUN:

- parametrii macrodefiniției nu sunt evaluați;
- se evaluează corpul macrodefiniției și se va produce o S-expresie intermediară;
- această S-expresie va fi evaluată, acesta fiind momentul în care sunt evaluați parametrii.

Să vedem mai întâi un exemplu comparativ. Fie următoarele definiții și evaluări:

<pre>&gt; (DEFUN F (N)   (PRINT N)   ) &gt; (SETQ N 10) &gt; (F N) 10 10</pre> <p><b>Notă</b> Parametrul N este evaluat de la bun început, PRINT va tipări valoarea acestuia, și o va întoarce ca valoare a apelului funcției.</p>	<pre>&gt; (DEFMACRO G (N)   (PRINT N)   ) &gt; (SETQ N 10) &gt; (G N) N 10</pre> <p><b>Notă</b> În primul moment nu se face o încercare de a se evalua parametrul N. Ca atare, PRINT va tipări N, după care îl va întoarce pe N ca valoare a formei intermediare a macrodefiniției. Apoi această valoare intermediară este evaluată și se va produce 10.</p>
--	--

Ca un alt exemplu, dorim să producem o macrodefiniție INC care să mărească cu 1 valoarea parametrului ei, ca în exemplul următor:

- (SETF X 10) se evaluează la 10
- (INC X) va produce 11
- X se evaluează la 11

Aceasta înseamnă că forma intermediară va trebui să fie

(SETQ parametru (+ 1 parametru))

Iată o posibilitate:

```
(DEFMACRO INC (N)
  (LIST 'SETQ N (LIST '+ 1 N))
)
```

## 5. Apostroful invers (backquote)

**Apostroful invers** (```) ușurează foarte mult scrierea macrocomenzilor. Oferă o modalitate de a crea expresii în care cea mai mare parte este fixă, și în care doar câteva detalii variabile trebuie completate. Efectul apostrofului invers este asemănător cu al apostrofului normal (`'`), în sensul că blochează evaluarea S-expresiei care urmează, cu excepția că orice **virgulă** (`,`) care apare va produce *evaluarea* expresiei care urmează. Iată un exemplu:

- (SETQ V 'EXEMPLU)
- `(ACESTA ESTE UN ,V) se evaluează la (ACESTA ESTE UN EXEMPLU)

De asemenea, apostroful invers acceptă construcția `,@`. Această combinație produce și ea evaluarea expresiei care urmează, dar cu diferența că valoarea rezultată trebuie să fie o listă. Elemente acestei liste sunt dizolvate în lista în care apare combinația `,@`. Iată un exemplu:

- (SETQ V '(ALT EXEMPLU))
- `(ACESTA ESTE UN ,V) se evaluează la (ACESTA ESTE UN (ALT EXEMPLU))
- `(ACESTA ESTE UN ,@V) se evaluează la (ACESTA ESTE UN ALT EXEMPLU)

Folosind apostroful invers, macrodefiniția INC se va scrie mai simplu astfel:

```
(DEFMACRO INC (N)
  `(SETQ ,N (+ 1 ,N))
)
```



Să observăm de asemenea că utilizarea apostrofului invers ușurează tipărirea rezultatelor intermediare.

## 6. Forme iterative

Limbajul Lisp prevede un set de forme iterative care permit controlul execuției într-un stil apropiat de cel cunoscut din limbajele clasice imperative.

### Forma PROG

#### (PROG I f<sub>1</sub> ... f<sub>n</sub>) : e

- evaluarea argumentelor face parte din procesul de evaluare al funcției
- I este o listă de inițializare conținând elemente de forma  
    “(variabilă valoare)” sau doar “variabilă”  
    În primul caz variabila este inițializată cu valoarea specificată, iar în al doilea caz primește valoarea NIL.
- variabilele sunt temporare și locale lui PROG
- Corpul funcției PROG este alcătuit din formele f<sub>1</sub>, ..., f<sub>n</sub> care sunt evaluate secvențial.
- La întâlnirea sfârșitului corpului funcției PROG se returnează implicit NIL. Pentru returnarea unei alte valori trebuie utilizată forma

#### (RETURN e) : e

- se evaluează argumentul
- rezultatul este valoare argumentul

Controlul evaluării formelor din corpul lui PROG se face folosind forma

#### (GO f)

- f este un atom etichetă sau o formă evaluabilă la un atom etichetă.
- permite reluarea evaluării lui PROG de la un punct specificat

### Exemplu

Să se determine lungimea unei liste în număr de elemente la nivel superficial.

(lungime '(1 (2 (3)) (4))) → 3

```
(defun lungime(l)
  (prog ((lung 0) (lista l))
    et
    (cond
      ((null lista) (return lung))
      (t (setq lung (+ lung 1))
         (setq lista (cdr lista))
         (go et))
```

## Forma DO

**(DO l<sub>1</sub> l<sub>2</sub> f<sub>1</sub> ... f<sub>n</sub>) : e**

- evaluarea argumentelor face parte din procesul de evaluare al funcției
- lista **l<sub>1</sub>** numită **listă de inițializare** are forma

```
( (var1 val_init1 pas1)  
  ...  
  (varn val_initn pasn)  
)
```

efectul fiind legarea simbolurilor **var<sub>i</sub>** de valorile inițiale **val\_init<sub>i</sub>** corespunzătoare, sau NIL dacă acestea nu se specifică; de remarcat că această operație de legare are loc în același moment pentru toate variabilele;

- lista **l<sub>2</sub>** numită **specificația de terminare** are forma  
(test\_terminare g<sub>1</sub> ... g<sub>n</sub>)
- corpul ciclului, alcătuit din formele ce se vor evalua repetat (f<sub>1</sub> ... f<sub>n</sub>) ca efect al ciclării

Un ciclu iterativ se desfășoară astfel:

- (1) se evaluează forma **test\_terminare** (deci este vorba despre un ciclu cu test inițial, de tip WHILE). La evaluarea diferită de NIL a testului clauzei se vor evalua în ordine formele de ieșire g<sub>i</sub> și se va returna rezultatul ultimei evaluări, g<sub>i</sub> (dacă nu există g<sub>n</sub> se va returna NIL).
- (2) dacă **test\_terminare** se evaluează la NIL atunci se trece la evaluarea formelor ce formează corpul lui DO, adică f<sub>1</sub>, ..., f<sub>n</sub>.
- (3) La întâlnirea sfârșitului corpului funcției fiecărei variabile index **var<sub>i</sub>** îi este asignată valoarea formelor de actualizare **pas<sub>i</sub>** corespunzătoare. Dacă forma **pas<sub>i</sub>** lipsește, variabila **var<sub>i</sub>** nu se va mai actualiza, rămânând cu valoarea pe care o are. Operația de atribuire are loc în același moment pentru toate variabilele. Ciclul se reia începând cu (1).

Ieșirea din DO se poate face la orice moment prin intermediul evaluării unei forme RETURN.

### Exemplu

Să se determine lungimea unei liste în număr de elemente la nivel superficial.

(lungime '(1 (2 (3)) (4))) → 3

```
(defun lungime(l)
  (do ((lista l (cdr lista)) ;lista de inițializare
      (lung 0 (+ lung 1))
      )
    ((null lista) lung) ;specificația de terminare
  ) ;DO un are corp
)
```

*Folosind formele iterative, prelucrarea de liste se realizează la nivel superficial.*

## Alte aspecte ale programării funcționale

### 7. Parametrii Lisp în context “Dynamic Scoping”

Așa cum am văzut până acum, toate aparițiile parametrilor formali sunt înlocuite cu valorile argumentelor corespunzătoare. Deci, avem o variantă de apel prin nume, apelul prin text, deoarece în Lisp avem de-a face cu determinarea dinamică a domeniului de vizibilitate (dynamic scoping) simultan cu înlocuirea textuală a valorilor evaluate.

Deși parametrul se leagă de argument (în sensul apelului prin referință din limbajele imperative, adică numele argumentului și cel al parametrului devin sinonime), nu este apel prin referință, deoarece modificările valorilor parametrilor formali nu afectează valoarea argumentelor (ceea ce amintește de apelul prin valoare).

De exemplu:

- (DEFUN FCT(X) (SETQ X 1) Y) se evaluează la FCT
- (SETQ Y 0) se evaluează la 0
- (FCT Y) se evaluează la 0

În secvența de mai sus Y se evaluează la 0, iar X se leagă la 0. Apelul FCT este astfel echivalent cu (FCT 0). X e parametru formal, Y e parametru actual, și ca atare modificarea lui X nu afectează pe Y, deci NU avem de-a face cu apel prin referință.

Aceasta demonstrează că transmiterea de parametri în LISP nu se face prin referință. Cum justificăm însă că nu este nici apel prin valoare? Există totuși situații în care valoarea parametrului actual poate fi modificată, de exemplu prin acțiunea funcțiilor cu efect distructiv RPLACA și RPLACD.

Să mai remarcăm faptul că în ciuda caracteristicii “Dynamic scoping”, variabila Y nu se leagă la execuție de parametrul formal tocmai înlocuit cu Y, adică Y-ul “intern” funcției, ci își păstrează caracteristica de variabilă globală pentru FCT.

Corpul unei funcții LISP este domeniul de vizibilitate a parametrilor săi formali, care se numesc variabile legate în raport cu funcția respectivă. Celelalte variabile ce apar în definiția funcției se numesc variabile libere.

Contextul curent al execuției este alcătuit din toate variabilele din program împreună cu valorile la care sunt legate acestea în acel moment. Acest concept este necesar pentru stabilirea valorii variabilelor libere care intervin în evaluarea unei funcții, evaluarea în LISP făcându-se într-un context dinamic (dynamic scoping), prin ordinea de apel a funcțiilor, și nu static, adică relativ la locul de definire.

Să reamintim în acest scop diferența între determinarea statică și respectiv cea dinamică a domeniului de vizibilitate în cadrul limbajelor imperative. Fie programul Pascal:

```
var
    a:integer;

procedure P;
begin
    writeln(a);
end;

procedure Q;
var
    a: integer;
begin
    a := 7;
    P;
end;

begin
    a := 5;
    Q;
end.
```

**Determinarea statică a domeniului de vizibilitate** (DSDV, static scoping) presupune că procedura P acționează în mediul de definire și ca urmare ea va “vedea” întotdeauna variabila a globală. Este și cazul limbajului Pascal, situație în care P apelat în Q va tipări 5 și nu 7.

Pe de altă parte, dacă am avea de-a face cu **determinarea dinamică a domeniului de vizibilitate** (DDDV, dynamic scoping), procedura P va tipări întotdeauna ultima (în ordinea apelurilor) variabilă *a* definită (sau legată, în terminologie Lisp). În acest caz se va tipări 7, deoarece ultima legare a lui *a* este relativ la valoarea 7.

Limbajul Lisp dispune de DDDV, această abordare fiind mai naturală decât cea statică pentru cazul sistemelor bazate pe interpretoare. O variantă Lisp a programului de mai sus care pune în evidență DDDV este:

```
(DEFUN P () A)
(DEFUN Q ()
  (SETQ A 7))
```

```

(P)
)
(SETQ A 5)
(SETQ Y (LIST (P) (Q)))
(PRINT Y)

```

Se va tipări lista (5 7), valoarea întoarsă de P fiind de fiecare dată ultimul A legat.

Avantajul legării dinamice este adaptabilitatea, adică o flexibilitate sporită. În cazul argumentelor funcționale însă pot să apară interacțiuni nedorite. Pentru a ilustra genul de probleme care pot apărea să luăm exemplul formei funcționale *twice*, care aplica de două ori argumentul funcției unei valori:

```
(DEFUN twice (func val) (funcall func (funcall func val)))
```

Exemple de aplicare:

- (twice 'add1 5) returnează 7
- (twice '(lambda (x) (\* 2 x)) 3) returnează 12

Dacă se întâmplă însă să folosim identificatorul *val* și în cadrul lui *func*, ținând cont de DDDV, apare o coliziune de nume, cu următorul efect:

- (setq val 2) returnează 2
- (twice '(lambda (x) (\* val x)) 3) returnează 27, nu 12

(cum probabil am dori să obținem). Aceasta deoarece ultima legare (dinamică deci) a lui *val* s-a efectuat ca parametru formal al funcției *twice*, deci *val* a devenit 3.

Această problemă a fost denumită problema FUNARG (funcțional argument). Este o problemă de conflict între DSDV și DDDV. Rezolvarea ei nu a constat în renunțarea la DDDV pentru LISP ci în introducerea unei forme speciale numite **function**, care realizează legarea unei lambda expresii de mediul ei de definire (deci tratarea acelui caz în mod static). Astfel,

- (twice (function (lambda (x) (\* val x))) 3) se evaluează la 12

Concluzia este deci că LISP are două reguli de DDV: DDDV implicit și DSDV prin intermediul construcției *function*.

## 8. Universul expresiilor și universul instrucțiunilor

Orice limbaj de programare poate fi împărțit în două așa-numite universuri (domenii):

1. universul expresiilor;
2. universul instrucțiunilor.

**Universul expresiilor** include toate construcțiile limbajului de programare al căror scop este producerea unei valori prin intermediul procesului de evaluare.

**Universul instrucțiunilor** include instrucțiunile unui limbaj de programare. Acestea sunt de două feluri:

(i) instrucțiuni ce influențează fluxul de control al programului:

- instrucțiuni conditionale;
- instrucțiuni de salt;
- instrucțiuni de ciclare;
- apeluri de proceduri și funcții.

(ii) instrucțiuni ce alterează starea memoriei:

- atribuirea (alterează memoria internă, primară);
- instrucțiuni de intrare/ieșire (alterează memoria externă, secundară)

Ca asemănare a acestor două universuri, putem spune că ambele alterează ceva. Există însă deosebiri importante. În universul instrucțiunilor ordinea în care se execută instrucțiunile este de obicei esențială. Adică instrucțiunile

$$i := i + 1; a := a * i;$$

au efect diferit fata de instrucțiunile

$$a := a * i; i := i + 1;$$

Fie

$$z := (2 * a * y + b) * (2 * a * y + c);$$

Multe compilatoare elimină evaluarea redundantă a subexpresiei comune  $2 * a * y$  prin următoarea substituție:

$$t := 2 * a * y; z := (t + b) * (t + c);$$

Această înlocuire s-a putut efectua în universul expresiilor deoarece în cadrul unei expresii o subexpresie va avea întotdeauna aceeași valoare.

În cadrul universului instrucțiunilor situația se schimbă, datorită posibilelor efecte secundare. De exemplu, pentru secvența

$$y := 2 * a * y + b; z := 2 * a * y + c;$$

factorizarea subexpresiei comune furnizează secvența neechivalentă

$$t := 2 * a * y; y := t + b; z := t + c;$$

Deși un compilator poate analiza un program pentru a determina pentru fiecare subexpresie în parte dacă poate fi factorizată sau nu, acest lucru cere tehnici sofisticate de analiză globală a fluxului (global flow analysis). Astfel de analize sunt costisitoare și dificil de implementat. Concluzionăm deci că universul expresiilor prezintă un avantaj asupra universului instrucțiunilor, cel puțin referitor la acest aspect al eliminării subexpresiilor comune (există, oricum, și alte avantaje care vor fi evidențiate în continuare).

Scopul programării funcționale este extinderea la nivelul întregului limbaj de programare a avantajelor pe care le promoveaza universul expresiilor față de universul instrucțiunilor.

## 9. Independența ordinii de evaluare

A evalua o expresie înseamnă a-i extrage valoarea. Evaluarea expresiei  $6 * 2 + 2$  furnizează valoarea 14. Evaluarea expresiei  $E = (2ax + b)(2ax + c)$  nu se poate face până nu precizăm valorile numerelor  $a$ ,  $b$ ,  $c$  și  $x$ . Deci, valoarea acestei expresii este dependentă de contextul evaluării. Odată acest lucru precizat, mai este important să subliniem că valoarea expresiei  $E$  nu depinde de ordinea de evaluare (adică de înlocuire a numerelor, mai întâi primul factor sau cel de-al doilea, etc). Este posibilă chiar evaluarea paralelă. Aceasta deoarece în cadrul expresiilor pure (așa cum sunt cele matematice) evaluarea unei subexpresii nu poate afecta valoarea nici unei alte subexpresii.

**Definiție.** O expresie pură este o expresie liberă de efecte secundare, adică nu conține atribuiri nici explicit (gen C) și nici implicit (apeluri de funcții).

Această proprietate a expresiilor pure, și anume independența ordinii de evaluare, se numește proprietatea Church-Rosser. Ea permite construirea de compilatoare ce aleg ordini de evaluare care fac uz într-un mod cât mai eficient de resursele masinii. Să subliniem din nou potențialul de paralelism etalat de această proprietate.

Proprietatea de transparență referențială presupune că într-un context fix înlocuirea subexpresiei cu valoarea sa este complet independentă de expresia înconjurătoare. Deci, odata evaluată, o subexpresie nu va mai fi evaluată din nou pentru că valoarea sa nu se va mai schimba. Transparența referențială rezultă din faptul că operatorii aritmetici nu au memorie, astfel orice apel al unui operator cu aceleași intrări va produce același rezultat.

Una dintre caracteristicile noțiunii matematice este interfața manifestă, adică, conexiunile de intrare ieșire între o subexpresie și expresia înconjurătoare sunt vizual evidente (de exemplu în expresia  $3 + 8$  nu există intrări “ascunse”) și ieșirile depind numai de intrări. Producătorii de efecte secundare, deci și funcțiile în general, nu au interfață manifestă ci o interfață nemanifestă (hidden interface).

Să trecem în revistă proprietățile expresiilor pure:

- valoarea este independentă de ordinea de evaluare;
- expresiile pot fi evaluate în paralel;
- transparența referențială;
- lipsa efectelor secundare;
- intrările și efectele unei operații sunt evidente.

Scopul programării funcționale este extinderea tuturor acestor proprietăți la întreg procesul de programare.

Programarea aplicativă are o singură construcție sintactică fundamentală și anume aplicarea unei funcții argumentelor sale. Modurile de definire a funcțiilor sunt următoarele:

1. **Definire enumerativă.** Este posibilă doar când funcția are un domeniu finit de dimensiune redusă.
2. **Definire prin compunere de funcții deja definite.** Dacă e cazul unei definiri în termeni de un număr infinit sau neprecizat de compuneri, o astfel de metodă este utilă doar dacă se poate extrage un principiu de regularitate, principiu care să ne permită generarea cazurilor încă neenumerate pe baza celor specificate. Dacă un astfel de principiu există suntem în cazul unei definiții recursive. În programarea funcțională recursivitatea este metoda de bază pentru a descrie un proces iterativ.

O altă distincție ce trebuie făcută relativ la definirea de funcții se referă la definiții explicite și implicite. O definiție explicită ne spune ce este un lucru. O definiție implicită statuează anumite proprietăți pe care le are un anumit obiect.

Într-o definiție explicită variabila definită apare doar în membrul stâng al ecuației (de exemplu  $y = 2ax$ ). Definițiile explicite au avantajul că pot fi interpretate drept reguli de rescriere (reguli care specifică faptul că o clasă de expresii poate fi înlocuită de alta).

O variabilă este definită implicit dacă ea este definită de o ecuație în care ea apare în ambii membri (de exemplu  $2a = a + 3$ ). Pentru a-i găsi valoarea trebuie rezolvată ecuația.

Aplicarea repetată a regulilor de rescriere se termină întotdeauna. Este posibil însă ca anumite definiții implicite să nu ducă la terminare (adică ele nu definesc nimic). De exemplu, ecuația fără soluție  $a = a + 1$  duce la un proces infinit de substituție.

Un avantaj al programării funcționale este că, la fel ca și în cazul algebrei elementare, ea simplifică transformarea de definiții implicite în definiții explicite. Acest lucru este foarte important deoarece specificațiile formale ale sistemelor soft au de obicei forma unor definiții implicite, însă conversia specificațiilor în programe are loc mai ușor în cazul definițiilor explicite.

Să mai subliniem că definițiile recursive sunt prin natura lor implicite. Totuși, datorită formei lor regulate de specificare (adică membrul stâng e format numai din numele și argumentele funcției) ele pot fi folosite ca reguli de rescriere. Condiția de oprire asigură terminarea procesului de substituții.

Față de limbajele conventionale, limbajele funcționale diferă și în ceea ce privește modelele operaționale utilizate pentru descrierea execuției programelor. Nu este proprie de exemplu pentru limbajele funcționale urmărirea execuției pas cu pas, deoarece nu contează ordinea de evaluare a subexpresiilor. Deci nu avem nevoie de un depanator în adevăratul înțeles al cuvântului.

Totuși, limbajul Lisp permite urmărirea execuției unei forme Lisp. Pentru aceasta există macrodefiniția TRACE care primește ca argument numele funcției dorite, rezultatul întors de TRACE fiind T dacă funcția respectivă există și NIL în caz contrar.