CURS 12

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

Cuprins

1. Generatori

Ca exemplu sugestiv pentru această problematică să consideram exemplul unei funcţii **VERIF** care primeşte o listă de liste şi întoarce T dacă toate sublistele de la nivel superficial sunt liniare şi 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

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

```
(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 toatele sublistele de la nivel superficial liste liniare.

  Model recursiv

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

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

Să observăm că funcţiile LIN şi VERIF de mai sus au aceeaşi structură, conformă şablonului

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

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

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

Model recursiv

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

```
(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 funcţia VERIF (L) astfel:

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

2. Argumente opționale

În lista parametrilor formali ai unei funcţii 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 rămaşi neatribuiţi, respectiv NIL dacă nu mai există parametri actuali neatribuiţi.

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) → 6
- (g 4 5) → 9
- (g 3) → 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 funcţie care să adune 1 la valoarea unei expresii:

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

Această funcţie se va utiliza în felul următor:

- (INC 10) se va evalua la 11

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

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

In marea majoritate a cazurilor, totuşi, vom avea nevoie de incrementarea cu 1 a valorii expresiei corespunzătoare lui NUMAR. In această situaţie putem folosi facilitatea argumentelor opţionale. Iată în continuare definiţia funcţiei INC cu un argument opţional:

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

```
            (T (+ NUMAR INCREMENT))
        )
    )
```

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

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

Putem, de asemenea, să luăm în considerare o valoare implicită pentru parametrii opţionali. Iată o definiţie a funcţiei INC în care parametrul opţional are valoarea implicită 1:

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

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

Să notăm că putem avea oricâte argumente opţionale. 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 opţional semnalat prin &REST, a cărui valoare devine lista tuturor argumentelor date cu excepţia celor care corespund parametrilor obligatorii şi opţionali. Fie o nouă versiune a funcţiei INC:

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

Iată câteva exemple de aplicare a acestei funcţii:

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

3. OBLIST şi ALIST

Un obiect Lisp este o structură alcătuită din:

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

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

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

In general, o funcţie definită cu *n* parametri P_1, P_2, ..., P_n şi apelată prin (F A_1 A_2 ... A_n) va adăuga la ALIST n perechi de forma (P_i . A_i) daca funcţia îşi evaluează argumentele sau (P_i . A_i'), unde A_i' este valoarea argumentului A_i, daca mediul Lisp evaluează argumentele. De exemplu, pentru funcţia

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

apelată cu (F 'X '(1.1)), vom avea pentru ALIST structura (... (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 şi cei şir de caractere prin valoarea lor direct în celula care îi conţine.

Iniţial, la începutul programului, ALIST este vidă. Pe măsura evaluării de noi funcţii, se adaugă perechi la ALIST, iar la terminarea evaluării funcţiei, perechile create la apel se şterg. In corpul funcţiei în curs de evaluare valoarea unui simbol *s* este căutată întâi în perechile din ALIST începând dinspre vârf spre bază (această acţiune este cea care stabileşte dinamic domeniul de vizibilitate). Dacă valoarea nu este găsită în ALIST se continuă căutarea în OBLIST. Aşadar, este clar că **ALIST şi OBLIST formează contextul curent al execuţiei unui program.**

Exemplu

- (SETQ X 1) iniţializează simbolul X cu valoarea 1;

- (SETQ Y 10) iniţializează simbolul Y cu valoarea 10;
- (DEFUN DEC (X) (SETQ X (- X 1))) defineşte o funcţie 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 ieşirea din corpul funcţiei şi revenirea în contextul apelator.

4. Macrodefiniții

Din punct de vedere sintactic, macrodefiniţiile se construiesc în acelaşi mod ca funcţiile, cu diferenţa că în loc să se folosească funcţia DEFUN se va folosi funcţia DEFMACRO:

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

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

- evaluarea argumentelor face parte din procesul de evaluare al macrodefiniției

Modul de lucru al macrodefiniţiilor create cu DEFMACRO este diferit de cel al funcţiilor create cu DEFUN:

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

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

> (DEFUN F (N) (PRINT N)) > (SETQ N 10) > (F N) 10 10	> (DEFMACRO G (N) (PRINT N)) > (SETQ N 10) > (G N) N 10
Notă Parametrul N este evaluat de la bun început, PRINT va tipări valoarea acestuia, şi o va întoarce ca valoare a apelului funcţiei.	**Notă** 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 macrodefiniţiei. Apoi această valoare intermediară este evaluată şi se va produce 10.

Ca un alt exemplu, dorim să producem o macrodefiniţie DEC care să decrementeze cu 1 valoarea parametrului ei, ca în exemplul următor:

- (SETF X 10) se evaluează la 10
- (DEC X) va produce 9
- X se evaluează la 9

Aceasta înseamnă că forma intermediară va trebui să fie

(SETQ parametru (- parametru 1))

Iată o posibilitate:

```
(DEFMACRO DEC (N)
        (LIST 'SETQ N (LIST '- N 1))
)
```

5. Apostroful invers (backquote)

Apostroful invers (`) uşurează foarte mult scrierea macrocomenzilor. Oferă o modalitate de a crea expresii în care cea mai mare parte este fixă, şi 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 excepţia 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 combinaţie produce şi ea evaluarea expresiei care urmează, dar cu diferenţa că valoarea rezultată trebuie să fie o listă. Elemente acestei liste sunt dizolvate în lista în care apare combinaţia **,@**. 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, macrodefiniţia DEC se va scrie mai simplu astfel:

```
(DEFMACRO DEC (N)
        `(SETQ ,N (- ,N 1))
)
```

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

6. Forme iterative

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

Forma PROG

(PROG l f_1 ... f_n) : e

- evaluarea argumentelor face parte din procesul de evaluare al funcției
- **l** este o listă de iniţializare conţinând elemente de forma
  "(**variabilă valoare**)" sau doar "**variabilă**"
  În primul caz variabila este iniţializată cu valoarea specificată, iar în al doilea caz primeşte valoarea NIL.
- variabilele sunt temporare şi locale lui PROG
- Corpul funcţiei PROG este alcătuit din formele f_1, ..., f_n care sunt evaluate secvenţial.
- La întâlnirea sfârşitului corpului funcţiei 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_1 l_2 f_1 ... f_n) : e

- evaluarea argumentelor face parte din procesul de evaluare al funcției
- lista **l_1** numită **listă de inițializare** are forma

  ((var$_1$ val_init$_1$ pas$_1$)
  ...
  (var$_n$ val_init$_n$ pas$_n$)
)

  efectul fiind legarea simbolurilor **var$_i$** de valorile inițiale **val_init$_i$** 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_2** numită **specificația de terminare** are forma
  (test_terminare g_1 ... g_n)

- corpul ciclului, alcătuit din formele ce se vor evalua repetat (f_1 ... f_n) 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_i și se va returna rezultatul ultimei evaluări, g_i (dacă nu există g_n se va returna NIL).

(2) dacă **test_terminare** se evalueaza la NIL atunci se trece la evaluarea formelor ce formează corpul lui DO, adică f_1, ..., f_n.

(3) La întâlnirea sfârșitului corpului funcției fiecarei variabile index **var$_i$** îi este asignată valoarea formelor de actualizare **pas$_i$** corespunzatoare. Dacă forma **pas$_i$** lipsește, variabila **var$_i$** 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 funcţionale

7. Parametrii Lisp în context "Dynamic Scoping"

Aşa cum am văzut până acum, toate apariţiile 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.

Deşi parametrul se leagă de argument (în sensul apelului prin referinţă din limbajele imperative, adică numele argumentului şi cel al parametrului devin sinonime), nu este apel prin referinţă, deoarece modificările valorilor parametrilor formali nu afectează valoarea argumentelor (ceea ce aminteşte 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

In secvenţa de mai sus Y se evaluează la 0, iar X se leaga la 0. Apelul FCT este astfel echivalent cu (FCT 0). X e parametru formal, Y e parametru actual, şi ca atare modificarea lui X nu afectează pe Y, deci NU avem de-a face cu apel prin referinţă.

Aceasta demonstrează că transmiterea de parametri în LISP nu se face prin referinţă. Cum justificăm însă că nu este nici apel prin valoare? Există totuşi situaţii în care valoarea parametrului actual poate fi modificată, de exemplu prin actiunea funcţiilor cu efect distructiv RPLACA şi RPLACD.

Să mai remarcăm faptul că în ciuda caracteristicii "Dynamic scoping", variabila Y nu se leagă la execuţie de parametrul formal tocmai înlocuit cu Y, adica Y-ul "intern" funcţiei, ci îşi păstrează caracteristica de variabilă globală pentru FCT.

Corpul unei funcţii LISP este domeniul de vizibilitate a parametrilor săi formali, care se numesc variabile legate în raport cu funcţia respectivă. Celelalte variabile ce apar în definitia funcţiei se numesc variabile libere.

Contextul curent al execuţiei 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 funcţii, evaluarea în LISP făcându-se într-un context dinamic (dynamic scoping), prin ordinea de apel a funcţiilor, şi nu static, adică relativ la locul de definire.

Să reamintim în acest scop diferenţa între determinarea statică şi respectiv cea dinamica 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 acţionează în mediul de definire şi ca urmare ea va "vedea" întotdeauna variabila a globală. Este şi cazul limbajului Pascal, situatie în care P apelat în Q va tipari 5 şi nu 7.

Pe de alta 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 evidenţă 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 funcţionale însă pot să apară interacţiuni nedorite. Pentru a ilustra genul de probleme care pot apărea să luăm exemplul formei funcţionale twice, care aplica de două ori argumentul funcţie 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 şi în cadrul lui func, ţinând 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 obţinem). Aceasta deoarece ultima legare (dinamică deci) a lui val s-a efectuat ca parametru formal al funcţiei twice, deci val a devenit 3.

Această problemă a fost denumită problema FUNARG (funcţional argument). Este o problemă de conflict între DSDV şi DDDV. Rezolvarea ei nu a constat în renunţarea la DDDV pentru LISP ci în introducerea unei forme speciale numite **function**, care realizeaza 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 şi DSDV prin intermediul constructiei function.

8. Universul expresiilor şi universul instrucţiunilor

Orice limbaj de programare poate fi împărţit în două aşa-numite universuri (domenii):
1. universul expresiilor;
2. universul instrucţiunilor.

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

Universul instrucţiunilor include instrucţiunile unui limbaj de programare. Acestea sunt de două feluri:

(i) instrucţiuni ce influenţează fluxul de control al programului:
- instrucţiuni conditionale;
- instrucţiuni de salt;
- instrucţiuni de ciclare;
- apeluri de proceduri şi funcţii.

(ii) instrucţiuni ce alterează starea memoriei:
- atribuirea (alterează memoria internă, primară);
- instrucţiuni de intrare/ieşire (alterează memoria externa, secundară)

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

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

au efect diferit fata de instrucţiunile

```
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 substituţie:

```
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 aceeaşi valoare.

În cadrul universului instrucţiunilor situaţia se schimbă, datorită posibilelor efecte secundare. De exemplu, pentru secvenţa

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

factorizarea subexpresiei comune furnizează secvenţa neechivalentă

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

Deşi 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 şi dificil de implementat. Concluzionăm deci că universul expresiilor prezintă un avantaj asupra universului instrcţiunilor, cel puţin referitor la acest aspect al eliminării subexpresiilor comune (există, oricum, şi alte avantaje care vor fi evidenţiate în continuare).

Scopul programării funcţionale este extinderea la nivelul întregului limbaj de programare a avantajelor pe care le promoveaza universul expresiilor faţă de universul instrucţiunilor.

9. Independenţa 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 numelor a, b, c şi x. Deci, valoarea acestei expresii este dependentă de contextul evaluarii. Odată acest lucru precizat, mai este important să subliniem că valoarea expresiei E nu depinde de ordinea de evaluare (adică de înlocuire a numelor, mai întâi primul factor sau cel de-al doilea, etc). Este posibilă chiar evaluarea paralelă. Aceasta deoarece în cadrul expresiilor pure (aşa cum sunt cele matematice) evaluarea unei subexpresii nu poate afecta valoarea nici unei alte subexpresii.

Definiţie. O expresie pură este o expresie liberă de efecte secundare, adică nu conţine atribuiri nici explicit (gen C) şi nici implicit (apeluri de funcţii).

Această proprietate a expresiilor pure, şi anume independenţa ordinii de evaluare, se numeşte 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 potenţialul de paralelism etalat de această proprietate.

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

Una dintre caracteristicile noţatiei matematice este interfaţa manifestă, adică, conexiunile de intrare ieşire între o subexpresie şi expresia înconjuratoare sunt vizual evidente (de exemplu în expresia 3 + 8 nu există intrări “ascunse”) şi ieşirile depind numai de intrări. Producătorii de efecte secundare, deci şi funcţiile în general, nu au interfaţă manifestă ci o interfaţă nemanifestă (hidden interface).

Să trecem în revistă proprietăţile expresiilor pure:

- valoarea este independentă de ordinea de evaluare;
- expresiile pot fi evaluate în paralel;
- transparenţa referenţiala;
- lipsa efectelor secundare;
- intrările şi efectele unei operatii sunt evidente.

Scopul programării funcţionale este extinderea tuturor acestor proprietăţi la întreg procesul de programare.

Programarea aplicativă are o singură construcţie sintactică fundamentală şi anume aplicarea unei funcţii argumentelor sale. Modurile de definire a funcţiilor sunt următoarele:

1. **Definire enumerativă**. Este posibilă doar când funcţia are un domeniu finit de dimensiune redusă.
2. **Definire prin compunere de funcţii 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 definiţii recursive. În programarea funcţională recursivitatea este metoda de bază pentru a descrie un proces iterativ.

O altă distincţie ce trebuie făcută relativ la definirea de funcţii se referă la definiţii explicite şi implicite. O definiţie explicită ne spune ce este un lucru. O definiţie implicită statuează anumite proprietăţi pe care le are un anumit obiect.

Într-o definiţie explicită variabila definită apare doar în membrul stâng al ecuaţiei (de exemplu y = 2ax). Definiţiile 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 ecuaţie în care ea apare în ambii membri (de exemplu 2a = a + 3). Pentru a-i găsi valoarea trebuie rezolvată ecuaţia.

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

Un avantaj al programării funcţionale este că, la fel ca şi în cazul algebrei elementare, ea simplifică transformarea de definiţii implicite în definiţii explicite. Acest lucru este foarte important deoarece specificaţiile formale ale sistemelor soft au de obicei forma unor definiţii implicite, însă conversia specificaţiilor în programe are loc mai uşor în cazul definiţiilor explicite.

Să mai subliniem că definiţiile recursive sunt prin natura lor implicite. Totuşi, datorită formei lor regulate de specificare (adică membrul stâng e format numai din numele şi argumentele funcţiei) ele pot fi folosite ca reguli de rescriere. Condiţia de oprire asigură terminarea procesului de substituţii.

Faţă de limbajele convenționale, limbajele funcţionale diferă şi în ceea ce priveşte modelele operaţionale utilizate pentru descrierea execuţiei programelor. Nu este proprie de exemplu pentru limbajele funcţionale urmărirea execuţiei pas cu pas, deoarece nu contează ordinea de evaluare a subexpresiilor. Deci nu avem nevoie de un depanator în adevăratul înţeles al cuvântului.

Totuşi, limbajul Lisp permite urmărirea execuţiei unei forme Lisp. Pentru aceasta există macrodefiniţia TRACE care primeşte ca argument numele funcţiei dorite, rezultatul întors de TRACE fiind T dacă funcţia respectivă există şi NIL în caz contrar.