LUCRAREA 7- Funcții cu efect distructiv.Funcții privite ca date. Macrodefiniții

1. SCOPUL LUCRÃRII

Lucrarea are drept scop prezentarea unor noțiuni noi privitoare la funcțiile cu efect distructiv, care lucrează prin actualizări direct asupra perechilor CONS. Este explicată maniera proprie limbajului Lisp de reprezentare unitara a datelor şi a funcțiilor. De asemenea se prezintă macrodefinițiile ca o alternativă la utilizarea funcțiilor.

2. CONSIDERAȚII TEORETICE

2.1. FUNCȚII CU EFECT DISTRUCTIV

Unitatea de bază în spațiul de lucru Lisp este celula de lista, perechea CONS, cu cele doua câmpuri, CAR şi CDR. Pe lângă funcțiile prezentate până acum, care asociază aceste elemente în structuri noi, există şi **posibilitatea modificării unor structuri deja existente**. Funcțiile din această subclasă mai poartă şi numele de funcții chirurgicale, având în vedere că permit actualizarea direct asupra acestor liste fără consum de celule CONS şi deci fără consum de memorie. Principalele funcții din această categorie sunt prezentate în continuare.

? RPLACA primul parametru se evaluează la o celulă CONS, al cărei câmp CAR se înlocuieşte cu valoarea celui de al doilea parametru.

? RPLACD primul parametru se evaluează la o celulă CONS, al cărei câmp CDR se înlocuieste cu valoarea celui de al doilea parametru. Exemple:

EXECUTIE	REZULTAT
*(SETQ x '(a b c))	(A B C)
*(RPLACA x 'd)	(D B C)
*(RPLACD x 'd)	(D . D)

Aceste funcții îşi au originea în primele implementări de Lisp. Programatorii în versiunile mai noi de Lisp preferă ca manieră de programare utilizarea funcției SETF pentru orice modificare de structură. Astfel, următoarele forme sunt echivalente ca efect lateral, dar au rezultate diferite:

(RPLACA x y) = (SETF (FIRST x) y)
(RPLACD x y) = (SETF (REST X) y)

Modificările efectuate asupra unei liste pot avea efect local sau global. În unele situații este recomandată construirea unor copii, în alte situații se recomandă ca modificarea să fie vizibilă peste tot. Un caz reprezentativ este constituit de funcțiile APPEND şi NCONC care realizează concatenarea argumentelor dar în maniere diferite. Funcția APPEND realizează concatenarea listelor furnizate ca parametri prin copierea nivelului superficial, iar NCONC realizează concatenarea listelor furnizate ca parametri prin modificarea tuturor listelor argument în afară de ultima.
Exemple:

EXECUTIE	REZULTAT
*(SETQ x '(a b c))	(A B C)
*(APPEND x y)	(A B C D E F)
*x	(A B C)
*(SETQ y '(d e f))	(D E F)
*(NCONC x y)	(A B C D E F)
*x	(A B C D E F)

2.2. MACRODEFINIȚII

Macrodefinițiile sunt construcții sintactice bazate pe procese substitutive ale textelor parametrizate şi sunt admise atât de către multe limbaje de asamblare puternice, cât şi de unele limbaje de nivel înalt. Spre

deosebire de aceste limbaje, în Lisp un macro este executat prin convertirea expresiilor şi nu a şirurilor de caractere.

Macrodefinițiile Lisp permit extinderea posibilităților de prelucrare într-un mod similar cu definirea unor noi funcții. Deşi se apelează sintactic ca şi o LAMBDA-expresie, tratarea de macro este realizată prin expandare înainte de evaluarea normală. Din acest considerent, definirile de macro sunt scrise în aşa fel încât în urma expandării, care constă în evaluarea formei, să rezulte o formă Lisp corectă care urmează să fie evaluată.

Observații: Prin acest proces de macroexpandare nu sunt evaluate argumentele, parametrii formali se leagă direct la parametrii actuali neevaluați.

? Procesul de macroexpandare se apelează recursiv până când forma rezultată nu mai este macro, în aşa fel încât se pot utiliza apeluri de macro în interiorul altui apel de macro.

Un macro este similar cu o LAMBDA-expresie corespunzătoare unei funcții, dar nu se poate apela ca şi o funcție auxiliară în interiorul unui iterator de tip MAPCAR.

Dacă suntem în cazul unui compilator incremental, cum sunt prevăzute pentru diferite versiuni de Common Lisp, este de remarcat că acestea sunt lipsite de faza de macroexpandare. Din acest motiv toate definirile de macro sunt plasate la începutul fişierului sursă, pentru a avea disponibilă în compilare forma expandată, în momentul apelului acesteia într-o eventuală utilizare ulterioară în interiorul unei funcții sau macrodefiniții.

DEFMACRO este o macrodefiniție care execută definirea unui macro; aşteaptă ca parametru numele de macro iar în continuare o LAMBDA-expresie similară unei definiri de funcție.

Când pe prima poziție a unei liste se găseşte un simbol care corespunde unei macrodefiniții spunem ca ne aflăm în fața unui apel de macro.

Definirea unui macro permite o facilitate adițională de exprimare a unei structuri complexe, care nu se întâlneşte în alte forme Lisp, facilitate cunoscută şi sub numele de destructurare. Spre deosebire de o LAMBDA-expresie corespunzătoare funcțiilor, unde în lista parametrilor nu pot apărea liste imbricate, în LAMBDA-expresiile corespunzătoare macrodefinițiilor pot apare parametri care să corespundă unor astfel de liste. Singura restricție provine din faptul că apelul de macro trebuie să furnizeze ca parametru o listă cu aceeaşi structură.

Lista precedată de apostrof invers este foarte des utilizată în macrodefiniții, având în vedere că permite după expandare obținerea unei forme Lisp, iar evaluarea eventuală a parametrilor se face facil prin precedarea acestora de către caracterul virgulă ",". În cazul în care nu se foloseşte această construcție, aşa cum se va vedea şi din exemplele prezentate, o mare atenție trebuie acordată contextului în care are loc legarea şi evaluarea eventuală a parametrilor.

2.3. FUNCȚII PRIVITE CA DATE

Manipularea funcțiilor ca şi orice alt tip de date creează în Lisp posibilitatea utilizării unor tehnici greu de imaginat în alte limbaje. Le vom exemplifica în continuare, printre altele, prin funcții auto-modificabile, auto-aplicative şi auto-reproductibile.

Primul aspect care relevă uniformitatea modului de tratare a datelor şi funcțiilor îl constituie argumentele funcționale. Am definit argumentele funcționale ca funcții care apar ca argumente ale altor funcții.

Un al doilea aspect care reflectă uniformitatea modului de tratare a datelor şi a funcțiilor îl reprezintă valorile funcționale. Printr-o valoare funcțională se înțelege faptul că asupra unei funcții se pot efectua operații de actualizare ca şi în cazul datelor. O funcție, la fel ca şi o dată, poate fi creată, afişată, modificată şi ştersă.

Pentru listarea definiției unei funcții sau accesul la LAMBDA-expresia care defineşte funcția se poate utiliza forma SYMBOL-FUNCTION. SYMBOL-FUNCTION este o funcție care aşteaptă ca parametru un simbol care desemnează o funcție sau un macro şi returnează definiția globală a funcției sau a macroului. Definirea globală a unei funcții poate fi alterată prin utilizarea funcției SETF cu parametrul SYMBOL-FUNCTION.

Aşa după cum se poate urmări în exemplele prezentate în cadrul surselor, modificarea definiției unei funcții este efectuată prin intermediul unei operații de actualizare asupra listei care reprezintă definiția globală a funcției. În particular, o funcție care ea însăşi execută această modificare se numeşte auto-modificabilă. Pentru exemplificarea acestui fenomen s-a imaginat un program experimental, în evoluție, care trăieşte două perioade: o fază de acumulare, a copilăriei, în care învață o anumită cantitate de cunoştințe şi o fază de maturitate în care aplică cunoştințele înglobate. Această evoluție şi auto-modificare a programului se modelează prin actualizarea şi supravegherea unei variabile globale, în cazul nostru este vorba de variabila liberă vârsta.

În continuare se prezintă o implementare a funcțiilor de tip MEMO în Lisp. Aceste funcții realizează prelucrări complexe asupra argumentelor, memorând într-o structură locala, pentru fiecare combinație de argumente, valoarea calculată. Acest proces de memorare are loc în vederea furnizării directe a rezultatului în urma unor apeluri ulterioare cu aceleaşi argumente.

Asemănător cu inteligența umană, care acumulează anumite cunoştințe, în urma unui apel al unui proces calculațional complex cu o anumită combinație de parametri, se memorează undeva doar sociația între datele de intrare, parametrii de apel şi datele de ieşire, rezultatele. În acest mod, la un apel ulterior în timp, cu aceiaşi parametri, răspunsul este mult mai rapid, fără a implica o eventuală etapă de calcul. În acest mod se obține un compromis între tabelarea datelor urmata de un proces de căutare a lor şi calculul acestora. Acest mecanism nu este aplicabil asupra funcțiilor care în cadrul apelului au efecte laterale sau sunt dependente de variabile globale.

Memoria locală pe care o vom denumi tabel va fi constituită ca o lista de asociații, pe care convenim sa o memorăm ca proprietate memo a numelui de funcție. În aplicație se utilizează funcția mod_memo care, primind ca argument o funcție, o transformă pe aceasta într-o funcție MEMO. Procedeul preia vechea definiție şi o înlocuieşte cu o alta în care se verifică la fiecare apel dacă nu cumva combinația de argumente se află deja în tabel. În caz de răspuns afirmativ se întoarce ca rezultat valoarea găsită în tabel. În caz contrar se calculează valoarea şi aceasta este introdusa în tabel.

După cum se poate observa în cadrul surselor, funcția nr-apel preia vechea definiție a unei funcții adăugând un înveliş care corespunde incrementării proprietății contor a numelui funcției, valoarea inițială a acesteia fiind 0. Această funcție poate fi utilizată pentru numărarea apelurilor şi poate deveni utilă în contextul studierii complexității unor prelucrări. În acest caz numărăm apelurile diferitelor funcții pentru a determina locurile înguste ale prelucrării, funcțiile care sunt apelate foarte des şi sunt mari consumatoare de timp. Optimizarea acestor funcții ar avea influență maxima asupra performantei totale a aplicației.

Unele limbaje mici sunt foarte puternice, în ciuda aparenței lor insignifiante. Deşi există teste care să permită evaluarea codului generat de un compilator sau a vitezei de compilare, nu există un test care să permită evaluarea puterii unui limbaj, a posibilităților sale de exprimare.

Într-un sens mai larg toate limbajele sunt echivalente, pentru ca toate sunt capabile sa acționeze ca maşini Turing, dar este evident că limbajele sunt diferite.

Un test pentru măsurarea puterii de exprimare a limbajelor este cel propus de Ken Thompson, care are drept scop scrierea celui mai scurt program care să se auto-reproducă, în urma execuției să producă o copie exactă a sursei sale. Acest exercițiu poate fi foarte elegant exprimat în Lisp pur, conținând doar funcții fără efecte laterale. Codul scris de John McCarthy şi Carolyn Talcott este prezentat în continuare:

```
((LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))
      (QUOTE
            (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X))) ))
```

---

3. DESFĂSURAREA LUCRĂRII

1. Să se evalueze secvențele:

```
*(SETQ a '(lispul este complicat) b (CDR a) c (CONS 'oare a) )
*(SETQ d (CONS 'ce b))                  *a    *b    *c    *d
*(RPLACA a 'inotul)                     *a    *b    *c    *d
*(SETF (FIRST a) 'aratul)               *a    *b    *c    *d
*(SETF (SECOND b) 'simplu)              *a    *b    *c    *d
*(SETF (CDR (LAST c)) '(?))             *a    *b    *c    *d
*(RPLACD a '(nu e simplu ?))            *a    *b    *c    *d
*(RPLACD b 'bine)                       *a    *b    *c    *d
*(SETF x '(a b) y '(1 2) w (CONS x (LIST x y)))
*(SETF z (APPEND x y))                        *(EQ z x)
*(SETF u (NCONC x y) )                        *(EQ u x)
*(NCONC x '(? !) )                      *w    *z
*(SETF x '(a b (c a (a x) b) a a b b) )
*(SETF y (REMOVE 'a x))                       *x
*(SETF z (DELETE 'b x))                       *x    *(EQ x z)
*(SETF u (DELETE 'a x))                       *x    *u    *(EQ x z)
```

2. Studiați funcția de inversare a elementelor unei liste, urmărind prin trasare efectul funcțiilor cu efect distructiv asupra listei alese ca parametru de apel.

3. Studiați macrodefinițiile următoare şi explicați comportarea apelurilor de macro prezentate în cazurile următoare:

```
*(SETF alfa 'val-a beta 'val-b)
*(DEFMACRO test1 (p1) (PRINT p1))
*(test1 alfa)
*(test1 beta)
*(DEFMACRO test2 (&rest rst) (PRINT (CONS 'test2 rst)))
*(test1 alfa)
*(test1 beta)
```

4. Urmăriți definirea diferitelor variante ale unei forme care să aibă aceeaşi comportare ca şi funcția sistem OR şi explicați erorile apărute.

5. Urmăriți definirea diferitelor variante ale unei forme care să aibă aceeaşi comportare ca şi funcția sistem WHILE şi explicați erorile apărute pe baza exemplelor de test.

6. Studiați exemplul simplu de destructurare propus pe apelul:

(ar-if <expresie_aritmetica> -127 128 (2 1))

al macrodefiniției ar-if prezentate în cadrul surselor, care implementează o formă de IF aritmetic ce tratează cazurile de numere negative şi nule iar pentru numerele pozitive tratează separat numerele pare de cele impare.

7. Studiați definiția unei funcții auto-modificabile prezentate în cadrul surselor. Urmăriți evoluția definiției acestei funcții prin apeluri individuale sau apelul repetat din funcția de test prevăzută. Vizualizați definiția funcției înainte şi după apeluri.

8. Studiați funcția diferență prezentata, precum şi varianta sa ca funcție MEMO.

9. Verificați pe exemplul prezentat în cadrul surselor modificarea unei funcții în aşa fel încât sa îşi numere apelurile.

4. ÎNTREBĂRI SI PROBLEME

1. Anticipați răspunsurile interpretorului la:

*(SETQ x '(a b c)) *(RPLACD (LAST x) x) *(SETF x '(a b c))	*(SETF x '(a b c)) *(RPLACA x (CONS (CDR x) (CDR x))) *(RPLACA x (CONS 'p x))

2. Scrieți forme echivalente pentru funcția NCONC şi pentru funcția de inversare a unei liste utilizând funcția SETF ca funcție cu efect distructiv asupra listelor.

3. Explicați comportarea următorului apel de macrodefiniție: (test2 alfa beta)

4. Concepeți alte macrodefiniții care să utilizeze facilitatea de destructurare.

5. Explicați comportarea funcției auto-modificabile la apeluri repetate ale funcției de test.

6. Studiați comparativ comportarea funcțiilor normale şi a aceloraşi funcții ca funcții MEMO. Stabiliți când tabelarea funcțiilor este avantajoasă şi când poate deveni stânjenitoare.

5. SURSE

```
;;; Inversare prin modificarea distructiva a listei argument
;;; Inverseaza si intoarce adresa ultimei celule CONS a listei.

(DEFUN nreverse1 (lista)
 (DO      ((ante ) (curent ) (urm lista) )
  ((ENDP urm) ante) (SETF curent urm )      (SETF urm (rest urm)) (SETF (CDR curent) ante) (SETF ante current) ))
```

```
;;; Inverseaza o lista, tratand special capul listei originale. Pentru ca aceasta celula sa ramana prima, apeleaza nreverse1.

(DEFUN nreverse (lista)
(LET ((ultim (CAR lista)) (rev (nreverse1 (CDR lista))) )
    (PROG2      (RPLACD (CDR lista) rev)    (RPLACD lista (CDR rev))          (RPLACD rev nil)
                (RPLACA lista (CAR rev))    (RPLACA rev ultim)
)))
```

```
;;; Macrodefinitie pentru actualizarea unei prorietati cu o valoare

(DEFMACRO our-putprop (ob val prop)  `(SETF      (GET (QUOTE ,ob) (QUOTE ,prop))  ,val)
)
```

```
;;; Macrodefinitie pentru o forma de tip IF False THEN f1 ELSE f2

(DEFMACRO if-not (test f1 &optional f2)      `(IF ,test ,f2 ,f1)
)
```

```
;;; Variante pentru o formă echivalentă cu funcția de sistem OR ;; Argumentele se evaluează de una sau de două ori!  Probleme dacă se evaluează simboluri ca args, loc-args, rez.

(DEFUN our-or1 ( &rest args )
(DO* ((loc-args args (REST loc-args)) (rez))
    ((NULL loc-args))
    (IF (SETF rez (EVAL (CAR loc-args)) )
        (RETURN rez) )
))
```

```
;; Aşteaptă argumentele într-o listă ;; Probleme dacă se evaluează simbolurile args, loc-args, rez.

(DEFUN our-or2 (args)
(DO* ((loc-args args (REST loc-args)) (rez))
    ((NULL loc-args))
    (IF (SETF rez (EVAL (CAR loc-args)) )  (RETURN rez) )
))
```

```
;; Probleme dacă se evaluează simbolurile locale args, loc-args,
;; rez. Evaluarea se face în contextul interior!

(DEFMACRO our-or3 (&rest args)
(DO* ((loc-args args (REST loc-args)) (rez) )
    ((ENDP loc-args))
    (IF (SETF rez (EVAL (CAR loc-args)))  (RETURN `(QUOTE ,rez)) )
))
;; our-or, varianta corecta! Evaluarea in contextul exterior!
(DEFMACRO our-or ( &rest args)
`(OR      ,(CAR args)    (our-or ,@(CDR args) )
))
```

```
;;; Variante pentru o forma echivalenta cu functia de sistem WHILE

(DEFMACRO while (cond &rest lforme)  `(DO  ()      ((NOT ,cond)) ,@lforme  ))
```

```
(DEFMACRO while1 (cond &rest lforme)
`(DO ((ret nil))
    ((NOT ,cond) ret) (DO* ((forme (QUOTE ,lforme) (CDR forme))     (forma (CAR forme) (CAR forme)))
            ((ENDP (CDR forme))
            (SETF ret (EVAL forma)) )              (EVAL forma))
))

(DEFMACRO while2 (cond &rest lforme &aux ret) `(DO ()       ((NOT ,cond) ret) (SETF ret (PROGN ,@lforme))
))

(DEFMACRO while3 (cond &rest lforme)
`(DO ((ret (CAR (LAST (QUOTE ,lforme)))))       ((NOT ,cond)(EVAL ret))       ,@lforme)
)
;;; Exemplu de testare forme WHILE
(SETF n 0)
(SETF d (while (< n 9) (PRINT n) (SETF n (+ n 1)) ))
(PRINT 'intoarce)(PRINT d)
(READ)
```

```
;;; Exemplu de macrodefinitie care utilizeaza destructurarea

(DEFMACRO ar-if (test neg zero (par impar))
 (LET ((var (GENSYM)))
  `(LET ((,var ,test))
    (COND((< ,var 0) ,neg)
            ((= ,var 0) ,zero)
            (T (COND ((= 0 (- ,var (* 2 (TRUNCATE (/ ,var 2)))))
                        ,par)
                    (T      ,impar)
            ))
    ))
))
```

```
;;; Functie auto-modificabila
;;; Trece, in functie de variabila globala varsta, prin doua  perioade: acumulare cunostinte si aplicarea lor.

(SETF varsta 0)

(DEFUN evolutie (x)
(COND      ((< varsta 7) (acumuleaza x)(SETF varsta (+ 1 varsta)))
    (T (SETF (SYMBOL-FUNCTION 'evolutie)
      (LIST (CAR (SYMBOL-FUNCTION 'evolutie))
            (CADR (SYMBOL-FUNCTION 'evolutie))
     (CAR (LAST (SYMBOL-FUNCTION 'evolutie))))))
)
(aplica x)
)

(DEFUN acumuleaza (x)
    (PRINT `(acumuleaza cu parametrul ,x)))

(DEFUN aplica (x)
    (PRINT `(aplica cu parametrul ,x))
    (TERPRI))

(DEFUN test ()
(DO ((i 0 (+ i 1)))
    ((= i 10))
    (PRINT `(varsta ,varsta))
    (evolutie i)
))
```

```
;;; Exemplu de functie MEMO

(SETF x '(a b c d ) y '(b c))
(DEFUN dif (x y)
    (DO ((par x (REST par)) (rez))
            ((NULL par) rez)
            (UNLESS (MEMBER (CAR par) y)
            (SETF rez (CONS (CAR par) rez)))))
(DEFUN mod_memo (fun)
(LET* ((old (SYMBOL-FUNCTION fun))
    (larg (SECOND old))
    (corp (CDDR old)) )
(SETF (SYMBOL-FUNCTION fun)
    `(LAMBDA ,larg
      (LET* ((pmemo (GET (QUOTE ,fun) 'memo))
                  (args (LIST ,@larg))
     (tabrez (ASSOC args pmemo :test #'EQUAL)))
    (COND(tabrez (CDR tabrez))
            (T      (SETF tabrez (PROGN ,@corp))
            (SETF  (GET (QUOTE ,fun) 'memo)
            (CONS (CONS args tabrez) pmemo))
                  tabrez))))
)))
(mod_memo 'dif)
```

```
;;; Exemplu de functie care isi numara apelurile intr-o proprietate

(DEFUN p () (PRINT "qwerty"))
(DEFUN nr (fun)
(LET*       ((old (SYMBOL-FUNCTION fun))
     (larg (SECOND old))
     (corp (CDDR old)) )
    (SETF (SYMBOL-FUNCTION fun)
            (CONS 'LAMBDA
            (CONS larg
            (CONS (SUBST        fun '$$$
                  '(SETF (GET '$$$ 'contor)
            (+ 1      (GET '$$$ 'contor))))
                              corp ))))
    (SETF (GET fun 'contor) 0)
))
```