LUCRAREA 4- Domeniul variabilelor. Forme iterative. Salturi nelocale

1. SCOPUL LUCRĂRII

În această lucrare se prezintă modul în care se face asocierea dintre un nume de simbol şi o valoare în timpul evaluării formelor Lisp. Lucrarea mai are ca scop familiarizarea cu formele recursive care se pot utliza în funcțiile Lisp, precum şi cu unele noțiuni privitoare la salturile nelocale, salturi între blocuri diferite.

2. CONSIDERAȚII TEORETICE

2.1. DOMENIUL VARIABILELOR

O serie de funcții (LET, DO, PROG, LAMBDA, etc.) permit introducerea de nume a căror valoare este diferită de cea globală pe parcursul construcțiilor pe care le introduc. O variabilă care apare în lista de parametri a unei astfel de construcții sau care este menționată în lista de parametri formali a unei funcții se numeşte "legată" în corpul acelei forme, respectiv în corpul funcției definite de utilizator.

O variabilă care este utilizată în corpul unei funcții definite de utilizator, dar care nu este menționată în lista de parametri a acelei funcții se numeşte "liberă".

Exemplu:

```
(DEFUN foo (x)          ; var. x este legata in corpul functiei foo
       (SETF q x))      ; variabila q este libera
```

Întrebarea la care vrem să răspundem este cum se face asocierea la un moment dat dintre un nume şi o valoare în Lisp. În cazul variabilelor legate răspunsul este acelaşi, indiferent de dialectul Lisp. Anume la intrarea într-un bloc care introduce un nume local se salvează legarea curentă a acelei variabile. Pe parcursul blocului respectiv variabila legată poate lua orice valori şi poate fi modificată de oricâte ori. La ieşirea din bloc se restaurează valoarea anterioară intrării în el. Exemplu:

EXECUTIE	REZULTAT
*(SETF x 'orice)	orice
*(LET (x) (SETF x 'ceva) (PRINT x))	ceva ceva
x	orice

Pentru a stabili valoarea ataşată unei variabile libere într-un bloc, se cunosc două convenții: legarea lexicală şi legarea dinamică. Legarea dinamică a fost mult utilizată în primele implementări de Lisp datoritã faptului că este uşor de implementat. Această modalitate de legare produce unele neplăceri. Codul compilat obținut prin convenția de legare lexicală este mai eficient. În Common Lisp se utilizează în mod implicit legarea lexicală. Totuşi, atunci când se doreşte, se poate utiliza şi convenția legării dinamice. În acest scop numele respective trebuie declarate "speciale".

Observație: Golden Common Lisp v1.00 utilizează convenția legării dinamice!

Legarea lexicală ataşează unui nume liber într-un bloc valoarea din blocul cel mai apropiat în care blocul curent este inclus şi în care respectivul nume este legat. Dacă nu există un astfel de bloc exterior, atunci numele respectiv referă valoarea globală ataşată lui. Pentru a determina care este valoarea ataşată unui nume în cazul convenției lexicale este suficientă examinarea textului Lisp, ataşarea fiind independentă de ordinea de apelare a diverselor funcții.

Exemplu:

EXECUTIE	REZULTAT
*(DEFUN silly-function (x) (test) (PRINT x))	SILLY-FUNCTION
*(DEFUN test () (PRINT x))	TEST
*(SETQ x 'GLOBAL-VALUE)	GLOBAL-VALUE
*(silly-function 7)	GLOBAL-VALUE 7 7
*(DEFUN even-silly-funct (x) (LET ((y 5)) (PRINT (CONS x y)) (foo y x)))	EVEN-SILLY-FUNCT
*(DEFUN foo (x z) (PRINT (LIST x y z)))	FOO
*(SETF x 'UN y 'DOI z 'TRE)	TRE
*(foo 3 4)	(3 DOI 4) (3 DOI 4)
*(even-silly-funct 7)	(7 . 5) (5 DOI 7) (5 DOI 7)

Legarea dinamică ataşează unui nume valoarea cea mai recent ataşată acelui nume din punct de vedere istoric, adică ținând cont de ordinea de evaluare a funcțiilor. Presupunând valabile definițiile funcțiilor din exemplul de mai sus, în cazul folosirii legării dinamice s-ar obține următoarele rezultate:

EXECUTIE	REZULTAT
*(SETQ x 'UNU y 'DOI z 'TREI)	TREI
*(test)	UNU UNU
*(silly-function 7)	7 7 7
*(foo 3 4)	(3 DOI 4) (3 DOI 4)
*(even-silly-funct 7)	(7 . 5) (5 5 7) (5 5 7)

În cazul legării lexicale nu este posibil ca o variabilă locală unui bloc să fie modificată în corpul unei funcții apelate în acel bloc, lucru posibil în cazul legării dinamice. Exemplu:

EXECUTIE	REZULTAT
*(SETQ x 1)	1
*(DEFUN f1 (x) (f2) x)	f1
*(DEFUN f2 () (SETQ x (+ x 1)))	f2
LEGARE LEXICALA	
*(f1 3)	3
*x	2
LEGARE DINAMICA	
*(f1 3)	4
*x	1

În Lisp orice simbol este caracterizat de o serie de atribute, cum ar fi numele, valoarea globală, lista de proprietăți, funcția ataşată etc. Aceste informații se păstrează pentru toți atomii simbolici cunoscuți de sistem la un moment dat într-o tabelă de simboluri cu numele generic de OBLIST. În OBLIST se păstrează numai valoarea globală a unui simbol, valoarea dinamică stabilindu-se, în funcție de convenția de legare, prin alte mecanisme. În cazul legării dinamice acest lucru se poate implementa foarte uşor cu ajutorul unei liste de asociații numite ALIST.

Putem privi ALIST-ul ca o stivă de perechi punct de forma (simbol . valoare-curenta). Inițial stiva ALIST e vida, dar la intrarea în fiecare domeniu în care există variabile legate (locale acelui domeniu) ea se extinde cu noi perechi punct ce păstrează valorile locale ale acelor parametri. La ieşirea din acel domeniu stiva ALIST se descarcă.

Valoarea unui simbol, în cazul legării dinamice, se stabileşte cercetând stiva ALIST dinspre vârf spre bază. Dacă nu e găsită în ALIST nici o pereche menționând simbolul căutat, atunci se caută în OBLIST valoarea globală. Altfel valoarea ataşată numelui este cea menționată în perechea punct respectivă.

Ca exemplu vom trasa evoluția stivei ALIST pe parcursul execuției formei (even-silly-funct 7)

MOMENT	ALIST
- înainte de apel	()
- imediat după intrarea în corpul funcției	((x . 7))
- imediat după intrarea în forma LET	((y . 5) (x . 7))
- după intrarea în funcția foo	((x . 5) (z . 7) (y . 5) (x . 7))
- la ieşirea din corpul funcției foo	((y . 5) (x . 7))
- la ieşirea din LET	((x . 7))
- la terminarea apelului even-silly-funct	()

Pentru ca în Common Lisp să utilizăm **legarea dinamică** pentru anumite simboluri trebuie adăugată declarația **(DECLARE (SPECIAL <var1> ...))** imediat după lista de parametri a unei funcții. Spre exemplu, pentru ca funcția "test" definită mai sus să producă comportamentul descris la legarea dinamică, în Common Lisp definiția ei ar trebui să fie:

```
*(DEFUN test ()
        (DECLARE (SPECIAL x))    (PRINT x)
 )
```

Un stil bun de scriere a funcțiilor Lisp evită folosirea variabilelor libere în descrierea funcțiilor.

2.2. FORME ITERATIVE

Metoda naturală de programare în Lisp este recursivitatea, exprimările recursive sunt în general mai elegante şi mai concise. Din păcate ele nu sunt la fel de eficiente în timpul execuției ca şi versiunile ce folosesc exprimări iterative. În scopul creşterii eficienței execuției au fost introduse în Lisp şi construcții ce permit iterația. Câteva dintre acestea sunt prezentate în continuare.

1. Forma **LET**

Sintaxa:

```
(LET ( <var1>     |     (<var1> <fv1>)
  <var2> | (<var2>  <fv2>) ... )
          <f1> <f2> ... <fn>     )
```

Are ca efect **legarea temporară** a simbolurilor <var1>, <var2>, ... la valorile ce rezultă în urma evaluării formelor <fv1>, <fv2>, ... Dacă vreo formă fvi lipseşte, simbolul corespondent se inițializează cu NIL. Inițializarea variabilelor se face în paralel, adică întâi se evaluează toate formele de inițializare şi abia apoi se face legarea valorilor la variabile. În continuare are loc evaluarea secvențială a formelor <f1>, <f2>, ..., <fn>, rezultatul ultimei forme evaluate fiind cel întors de forma LET.

La ieşirea din forma LET variabilele **îşi recapătă vechile valori**. Exemplu:

EXECUTIE	REZULTAT
*(SETF a 1 b 1)	1
*(LET ((a '(a)) (b a) (c (+ 2 3)) d (e)) (LIST a b c d e))	((a) 1 5 NIL NIL)

2. Forma LET*

Este identică cu LET doar ca **inițializările se fac secvențial, nu în paralel!**

Exemplu:

EXECUTIE	REZULTAT
*(SETF a 1 b 1)	1
(LET ((a '(a)) (b a) (c (+ 2 3)) d (e)) (LIST a b c d e))	((a) (a) 5 NIL NIL)

3. Forma LOOP

Sintaxa:

(LOOP <f1> <f2> ... <fn>)

Are ca efect repetarea de un număr nedefinit de ori a secvenței de forme <f1>, <f2>, ..., <fn>. Evaluarea unei forme (RETURN <formă-return>) provoacă ieşirea din ciclu şi întoarce rezultatul evaluării formei <formă-return>.

4. Forma DO

Sintaxa:

```
(DO (( <var1> [ <finit1> [ <fpas1>] ])
       ( <var2> [ <finit2> [ <fpas2>] ])
              ...
       ( <varn> [ <finitn> [ <fpasn>] ])
     )
 (<trigger> <fend1> <fend2>... <fendm> )
  <f1> <f2> ... <fp>
)
```

Are ca efect utilizarea variabilelor <var1>, <var2>, ..., <varn> ca variabile locale formei DO. La intrare vor fi legate la valorile ce rezultă în urma evaluării formelor de inițializare <finit1>, <finit2>, ..., <finitn>. Legarea se face în paralel, ca la LET. Dacă lipseşte forma de inițializare, variabila corespondentă se inițializează cu NIL. Execuția formei DO constă în paşii:

(a) Se creează variabilele locale cu valorile inițiale conform blocului de inițializare.

(b) Se evaluează forma <trigger> care controlează terminarea buclării. Dacă rezultatul este NIL se continuă cu pasul (c), altfel se întrerupe buclarea, continuându-se cu evaluarea formelor <fend1>, <fend2>, ..., <fendm>, rezultatul ultimei forme evaluate fiind cel întors de DO.

(Dacă aceste forme lipsesc, atunci forma DO întoarce NIL.)

(c) Dacă rezultatul testului a fost NIL se continuă cu evaluarea secvenței de forme <f1>, <f2>, ..., <fp>.

(d) După terminarea evaluării secvenței fiecare variabilă <vari> pentru care a existat specificată forma <fpasi> se leagă la valoarea întoarsă de forma respectivă. Variabilele de ciclu care nu au asociate asemenea forme nu se resetează. Se reia ciclul cu pasul (b).

Forma DO poate fi părăsită în orice punct dacă se execută o forma RETURN. Exemplu:

Iată valorile variabilelor la fiecare iteratie:

EXECUTIE	REZULTAT	Iteratie	I	result
*(DO ((I '(THIS IS A LIST) (REST I)) ;specificare (result NIL)) ;parametri locali ((NULL L) result) ;clauza test (SETF result (CONS (FIRST I) result)) ;corpul)	(LIST A IS THIS)	1	(THIS IS A LIST)	()
		2	(IS A LIST)	(THIS)
		3	(A LIST)	(IS THIS)
		4	(LIST)	(A IS THIS)
		5	()	(LIST A IS THIS)

<table>
<tr><td>5. Forma DO*</td><td>Este identică cu DO doar că variabilele se leagă secvențial ca la LET*.
Exemplu:
<table>
<tr><th>EXECUTIE</th><th>REZULTAT</th></tr>
<tr><td>*(SETF a 1)</td><td>1</td></tr>
<tr><td>*(DO ((a (+ a 1) (+ a 1))
(b (+ a 1) (+ a 1))
)
((> a 3) 'gata)(PRINT `(,a ,b)))</td><td>(2 2)
(3 3)
gata</td></tr>
<tr><td>*(SETF a 1)</td><td>1</td></tr>
<tr><td>*(DO* ((a (+ a 1) (+ a 1)) (b (+ a 1) (+ a 1)))
((> a 3) 'gata) (PRINT `(,a ,b))
)</td><td>(2 3)
(3 4)
gata</td></tr>
</table></td></tr>
<tr><td>6. Forma PROG
Sintaxa:
(PROG (<var1> | (<var1> [<finit1>])
<var2> | (<var2> [<finit2>])
...
<varn> | (<varn> [<finitn>]))
[<et1>] <f1>
[<et2>] <f2>
...
[<etm>] <fm>)</td><td>Are ca efect utilizarea variabilelor <var1>, <var2>, ..., <varn> ca variabile locale formei PROG, care se leagă inițial la valoarea formei de inițializare corespunzătoare sau la NIL, (legările facându-se în paralel, ca la LET).
Cu rol de etichete se utilizează formele <eti> care trebuie să fie atomice şi nu se evaluează.
Evaluarea corpului formei PROG se face secvențial <f1>, <f2>, ..., <fm> dacă nu se întâlnesc pe parcurs forme GO sau RETURN.
Dacă se ajunge la sfârşitul corpului PROG se întoarce NIL (nu valoarea ultimei forme evaluate!).
Evaluarea unei forme (GO <eti>) în interiorul corpului PROG are ca efect transferul controlului la forma ce urmează etichetei <eti>.
Evaluarea unei forme (RETURN <formă-return>) provoacă părăsirea formei PROG şi întoarcerea valorii ce rezultă în urma evaluării formei <formă-return>.
<table>
<tr><th>EXECUTIE</th><th>REZULTAT</th></tr>
<tr><td>*(PROG ()
(GO salt)
(RETURN 1)
salt (RETURN 2)
)</td><td>2</td></tr>
</table></td></tr>
<tr><td>7. Forma PROG*</td><td>Este analoagă cu PROG, doar că inițializările se fac ca la LET*</td></tr>
<tr><td>8. Forma PROG1
Sintaxa:
(PROG1 <f1> <f2> ... <fn>).</td><td>Evaluarea formei PROG1 are ca efect evaluarea pe rând a formelor <f1>, <f2>, ..., <fn> şi întoarcerea ca valoare a apelului PROG1 a rezultatului evaluării primei forme din secvență: <f1>.</td></tr>
<tr><td>9. Forma PROG2</td><td>Este asemănătoare cu PROG1, doar ca se întoarce valoarea celei de-a doua forme din secvență: <f2>.</td></tr>
</table>

2.3. SALTURI NELOCALE

Formele **GO şi RETURN** permit salturi la nivelul superficial al formei PROG care contine atât saltul cât şi eticheta la care se face saltul. În Lisp sunt prevăzute facilități pentru salturi nelocale prin mecanismul **CATCH-THROW**. Acest mecanism este activat prin inițierea unui lanț de evaluări de forme declanşate de forma CATCH care va forța la un moment ulterior de timp evaluarea unei forme THROW. Revenirea din THROW are loc direct în CATCH întorcând forma evaluată de către THROW.

Forma CATCH serveşte ca un indicator țintă pentru transferul controlului execuției de la un salt nelocal THROW.

? CATCH	• **aşteptă doi parametri, un indicator şi o listă de forme. Forma pentru indicator este evaluată pentru a produce un obiect necesar în identificarea formei CATCH destinație a saltului dintr-un THROW (în cazul mai multor apeluri CATCH avem mai multe destinații posibile). Formele din listă sunt evaluate succesiv, iar valoarea ultimei forme este întoarsa ca rezultat, cu excepția cazului în care se întâlneşte pe parcursul evaluării o formă THROW, caz în care se întoarce ca rezultat valoarea formei THROW, iar evaluarea restului de forme din CATCH este oprită.**
? THROW	• **aşteaptă doi parametri, un indicator şi o formă. Forma pentru indicator este evaluată pentru a produce un obiect necesar în identificarea formei CATCH destinație a saltului. Testul de identificare se face prin intermediul predicatului EQ, iar dacă nu există forma CATCH corespunzătoare este semnalată o eroare. Forma este evaluată iar valoarea este întoarsa ca rezultat al formei CATCH corespunzătoare.**

Uneori este necesară evaluarea unei forme chiar dacă au apărut efecte laterale în succesiunea procesului de evaluare. Astfel, spre exemplu, într-o formă

```
(PROG (deschide_fişier)
(prelucrare_fişier)
(închid_fişier) )
```

funcția de închidere a fişierului trebuie evaluată chiar şi în cazul unor posibile erori intervenite în prelucrarea fişierului, urmate de inițierea unui salt nelocal printr-o formă THROW la o formă anterioară CATCH unde sunt capturate. Pentru înlăturarea acestui neajuns este folosită forma prezentată în continuare:

? UNWIND-PROTECT	**- aşteaptă ca parametri două forme care reprezintă forma protejată şi respectiv forma curată, care este evaluată chiar şi în cazul unei ieşiri anormale. Forma întoarce ca rezultat valoarea formei protejate şi neglijează toate rezultatele evaluării formelor curate.** **Ca regulă generală, UNWIND-PROTECT garantează execuția formelor curate după orice ieşire din forma protejată, indiferent de faptul că ieşirea a fost normală sau inițiată de un salt nelocal THROW la o formă corespunzătoare CATCH. Este de remarcat faptul că UNWIND-PROTECT garantează execuția formelor curate, atât contra tuturor salturilor nelocale CATCH-THROW, cât şi contra ieşirilor lexicale de tip GO sau RETURN.**

Observație: **Utilizarea formelor iterative şi a salturilor nelocale nu este indicată într-o programare funcțională, eleganta în limbajul Lisp!**

3. DESFĂSURAREA LUCRĂRII

1. Să se evalueze secvența şi să se determine ieşirile interpretorului în cazul în care avem (i) legare lexicală şi (ii) legare dinamică. În cazul legării dinamice să se traseze evoluția stivei ALIST.

a)	b)
*(SETF var 'libera)	*(DEFUN f2 (x) (SETF x 1) y)
*(SETF alta-var 'libera)	*(SETF y 0)
*(DEFUN f1 (alta-var) (SETF var alta-var) (SETF alta-var 'orice))	*(f2 y)
*var	*(DEFUN ev-prim (l) (EVAL (CAR l)))
*alta-var	*(SETF l '1)
*(f1 'legata)	*(ev-prim '(y p q))
*var	*(ev-prim '(l p q))
*alta-var	

2. Se vor testa funcțiile care utilizează forme iterative prezentate în continuare.

3. Să se compare consumul de resurse şi durata de execuție între variantele recursivă, recursivă cu parametru de acumulare şi iterativă ale unei funcții.

4. Încercați să scrieți o variantă total iterativă echivalentă cu funcția EQUAL. Comparați efortul de programare cu cel necesar scrierii versiunii recursive.

5. Studiați prin trasare salturile nelocale prin CATCH-THROW din exemplul de test prezentat în cadrul surselor precum şi în micro-editorul care are unele comenzi de parcurgere liste.

6. Studiați în exemplele de test prezentate în surse protecția la salturi locale şi nelocale a execuției unei forme prin UNWIND-PROTECT.

4. ÎNTREBARI SI PROBLEME

1. Să se descrie **variante iterative** pentru funcțiile: **diferență, test-inclusă, extind1, fuzion, reun-clase.**

2. Să se descrie în variantele **(i) recursivă, (ii) recursivă cu parametru de acumulare, (iii) iterativă,** funcțiile care:

a) calculează lungimea unei liste
b) testează dacă o listă este ordonată crescător
c) elimină dintr-o listă elementele nenumerice
d) elimină dintr-o listă toți atomii nenumerici, indiferent de nivelul de imbricare pe care se află
e) însumează atomii numerici de pe nivelul superficial al unei liste
f) însumează atomii numerici de pe toate nivelurile unei liste
g) calculează al n-lea element din şirul lui Fibonacci

3. Să se scrie o variantă a funcției reun-clase în care clasa şi elem să fie externe funcției - caz în care trebuie declarate "speciale" pentru a se folosi legarea dinamică.

4. Să se descrie funcțiile care implementează operațiile elementare (construcție, reuniune, intersecție, diferență) asupra multiseturilor. Un multiset este o generalizare a noțiunii de set, în care un element are ataşat numărul de apariții.
Exemplu: lista (a b c a a b) are asociat multisetul ((a . 3) (b . 2) (c . 1))

5. SURSE

```
;;; Calculul valorii functiei exponentiale
;;; cu baza intreaga si exponent natural
;; iteratie cu "DO" - varianta 1 ("rez" e externa lui "do")

(DEFUN exp3 (m n)
(LET ((rez 1))
        (DO (   (exp n (- exp 1)) )
                ((ZEROP exp) )
                (SETF rez (* rez m)) )
rez     ))
```

```
;;; Calculul factorialului unui numar
;; iteratie cu "DO" - varianta 1

(DEFUN fact2 (n)
(DO (   (nn n (- nn 1))
        (rez 1 (* rez nn)) )
        ((ZEROP nn) rez)
))
```

```
;; iteratie cu "DO" - varianta 2

(DEFUN exp4 (m n)
(DO (   (rez 1)
        (exp n (- exp 1)) )
        ((ZEROP exp) rez)
        (SETF rez (* m rez))
))
```

```
;;iteratie cu "DO" - varianta 2

(DEFUN fact3 (n)
(DO ( (rez 1) (nn n (- nn 1)) )
        ((ZEROP nn) rez)
        (SETF rez  (* rez nn) )))
```

```
;; iteratie cu "DO" - varianta 3
(DEFUN exp5 (m n)
(DO (   (rez 1 (* rez m))
        (exp n (- exp 1)) )
        ((= exp 0)  rez)
))
```

```
;; iteratie cu "DO* "
(DEFUN fact4 (n)
(DO* (  (rez 1 (* rez nn))
        (nn n  (- nn 1)) )
        ((ZEROP nn) rez )))
```

```
;; iteratie cu "DO*" - varianta gresita !

(DEFUN factrau (n)
(DO* (  (nn n (- nn 1))
        (rez 1 (* rez nn)) )
        ((ZEROP nn) rez ))
)
```

```
;; iteratie cu "LOOP"

(DEFUN exp6 (m n)
(LET ((rez 1))
        (LOOP
                (IF (ZEROP n) (RETURN rez))
                (SETF n (- n 1) rez (* rez m)) )
))
```

```
;; iteratie cu "LOOP"

(DEFUN fact5 (n)
(LET ((rez 1))
 (LOOP (WHEN (ZEROP n) (RETURN rez))
                (SETF rez (* rez n))
                (SETF n (- n 1))
        )
))
```

```
;; iteratie cu "PROG"

(DEFUN exp7 (m n)
(PROG ((rez 1))
        cic     (IF (ZEROP n) (RETURN rez))
                (SETF n (- n 1) rez (* rez m))
                (GO cic)
))
```

```
;; iteratie cu "PROG"

(DEFUN fact6 (n)
(PROG ((rez))
                (SETF rez 1)
        cic     (IF (ZEROP n) (RETURN rez) )
                (SETF rez (* rez n))
                (SETF n (- n 1))
                (GO cic)
))
```

```
;;; Operatii simple pe liste
;;; ultima celula CONS a unei liste.
;; iteratie cu "DO"

(DEFUN last1 (lis)
(DO ( (var lis (CDR var)) )
    ((OR (ATOM var) (ENDP (REST var))) var)
))


;; iteratie cu "LOOP"

(DEFUN last2 (lis)
(LOOP (IF (OR (ATOM lis) (ENDP (CDR lis)))
                  (RETURN lis) )
          (SETF lis (CDR lis))
))
```

```
;;; Operatii cu clase de echivalenta
;;; fuzioneaza perechile echivalente dintr-o lista returnind clasele de echivalenta

(DEFUN fuzionare (perechi)
          (fuzion perechi NIL) )

;; se acumuleaza in parametrul "clase" clasele de echivalenta care corespund multimii de relatii binare din "perechi"

(DEFUN fuzion (perechi clase)
          (IF (NULL perechi)
                    clase
                    (fuzion (REST perechi)
                    (absorb (FIRST perechi) clase) ) ))
```

```
;;; Lista primelor "n" elemente dintr-o lista data; iteratie cu "DO*"

(DEFUN fata2 (ls n)
(DO* (  (rez NIL (CONS (CAR lis) rez))
       (lis ls (REST lis))
       (nn n (- nn 1)) )
      ((OR (= nn 0) (ENDP lis)) (REVERSE rez))
))

;;; inversarea elementelor unei liste
;; iteratie cu "PROG"

(DEFUN rev2 (ls)
(PROG ( (rez NIL) )
   cic   (WHEN (ENDP ls) (RETURN rez) )
          (SETF rez (CONS (CAR ls) rez) )
          (SETF ls (CDR ls))
          (GO cic)
))
```

```
;; Reuneşte clasele de echivalenţă ale celor două elemente din "pereche". Dacă nu găseşte clasele corespunzătoare, creează o nouă clasă.

(DEFUN absorb (pereche clase)
(LET (  (is-first (MEMBER (FIRST pereche)
          (FIRST clase)))
 (is-second (MEMBER (SECOND pereche)
           (FIRST clase))) )
    (COND       ((ENDP clase) (LIST pereche))
                ((AND is-first is-second) clase)
                ((NOT (OR is-first is-second )) 
                (CONS (FIRST clase)
          (absorb pereche (REST clase))))
          ((reun-clase
                (FIRST clase)
   (OR (AND  is-first  (SECOND pereche))
   (AND  is-second (FIRST  pereche)))
  (REST clase))) )
))
```

```
;;; eliminarea parantezelor interioare din listă; parcurgere iterativă în lățime, recursivă în adâncime

(DEFUN striv2 (lis)
( DO (  (par lis (REST par) )
         (rez)
       )
       ( (NULL par) rez )
         (SETF rez (APPEND rez
         (IF (ATOM (FIRST par))
                   (LIST (FIRST par))
         (striv2 (FIRST par))))
))
)
```

```
;; reuneste clasa "clasa" cu cea coresp. lui "elem" din "clase"

(DEFUN reun-clase (clasa elem clase)
        (COND   ((ENDP clase)
              (LIST (CONS elem clasa)) )
           ((MEMBER elem (FIRST clase))
           (CONS (APPEND clasa (FIRST clase))
              (REST clase)))
           ((CONS          (FIRST clase)
           (reun-clase clasa elem (REST clase))))
))
```

```
;;; Salturi nelocale utilizind THROW si CATCH
(DEFUN test-catch (m)
          (CATCH 'exit (test-c 1)) )
(DEFUN test-c           (n)
(PRINT `(in ,@(if (= n 1) '(primul) `(al ,n - lea)) test-c))
   (IF (< n m) (CATCH n (test-c (+ n 1)) ) )
       (PRINT `(ies din al ,n - lea test-c))
        (COND ((= n (- m 2)) (THROW (- n 2)))
 ((= n 4) (THROW 'exit `(vin din throw-ul cu n = ,n)))
))
```

```
;;; implementarea unui EDITOR simplu pentru forme Lisp; comenzi:

;;"jos"  - avans editare pe car-ul formei;  "dr"        - avans editare pe CDR-ul formei
;;"st"   - revenire spre stinga ;"sus" - revenirea la forma ce continea forma curenta
;;"exit"- terminare editare;   altceva-se evalueaza, se reia editarea din pozitia curenta
;; Revenirea spre stinga se face prin iesirea din LOOP cu RETURN
; iar revenirea in sus prin (THROW '$edit)

(DEFUN edit (form)
  (LET ( comanda ) (CATCH '$exit (edit1 form nil nil))
))
(DEFUN edit1 (form are-sus are-stg)
  (LOOP (PRINC "Forma curenta ")
           (PRINT form)
           (PRINC "Comanda: ")
           (SETQ comanda (read)
  )
  (COND ((EQ comanda 'jos)  (IF (NOT (ATOM form))   (CATCH '$edit (edit1 (CAR form) T nil))) )
          ((EQ comanda 'dr)     (AND (CDR form) (edit1 (CDR form) are-sus T)))
          ((EQ comanda 'sus)   (AND are-sus (THROW '$edit)))
          ((EQ comanda 'st)     (AND are-stg (RETURN NIL)))
          ((EQ comanda 'exit)   (THROW '$exit 'EXIT))
          (T (PRINT (EVAL comanda)) )
)))
```

```
;;; Exemplu de protejare a formelor cu UNWIND-PROTECT
;;forme neprotejate, salt local

(DEFUN neprotej (n)
(BLOCK bloc
(PROGN
          (PRINT `(deschid fisier))
          (PRINT `(incep prelucrare fisier))
         (COND ((= 0 (- n (* 3 (TRUNCATE (/ n 3)))))
                          (RETURN-FROM bloc)))
          (PRINT `(termin prelucrare fisier))
          (PRINT `(inchid fisier))
)))
```

```
;; forme neprotejate, salt nelocal

(LET ((foo '(1)))
(CATCH 'tag
          (PROGN
                    (SETF foo
                              (CONS 2 foo))
                    (THROW 'tag NIL))
          (SETF foo (CONS 3 foo)))
foo)
```

```
;; forme protejate corespunzatoare, salt local

(DEFUN protejat (n)
(BLOCK bloc
(UNWIND-PROTECT
(PROGN
          (PRINT `(deschid fisier))
          (PRINT `(incep prelucrare fisier))
          (COND ((= 0 (- n (* 3 (TRUNCATE (/ n 3)))))
                          (RETURN-FROM bloc)))
          (PRINT `(termin prelucrare fisier)))
          (PRINT `(inchid fisiser))
)))
```

```
;; forme protejate corespunzatoare, salt nelocal

(LET ((foo '(1)))
(CATCH 'tag (UNWIND-PROTECT
          (PROGN
                    (SETF foo
                              (CONS 2 foo))
                    (THROW 'tag NIL))
          (SETF foo (CONS 3 foo))))
foo)
```