

Programare funcțională MACRO-uri

Isabela Drămnesc

1 MACROURI

În Lisp nu există proceduri, cu numai funcții deoarece orice rutină întoarce obligatoriu și o valoare.

DEFINIȚII

- Un MACRO este funcție care definește o funcție.
- MACROURILE sunt mijloace prin care putem extinde sintaxa Lisp-ului.
- MACROURILE sunt funcții care au un mecanism special de evaluare în doi pași:
 1. expandarea
 2. evaluarea propriu-zisă

Pasul 1) Se construiește o expresie în definiție (s.n. macroexpandare)
- definiția conține nume-macro, listă-parametri, corp
Pasul 2) se evaluează expresia

defmacro seamănă cu defun

- - amândouă dau un nume unui grup de forme
- - permit utilizarea unor variabile

Macro-urile Lisp ne permit definirea unor operații care sunt implementate prin transformare.

Macro-urile sunt funcții care modifică expresii.

Printr-un macro se pot controla evaluările argumentelor din apel.

DAR sunt multe diferențe între funcții și macro-uri:

- o funcție produce rezultate
- un macro produce expresii, care atunci când sunt evaluate, produc un rezultat

Exemple:

```

; folosind defmacro = defineste cum trebuie transformat un apel
> (defmacro zece (var) (list 'setq var 10))
ZECE

; expandare (lucreaza cu expresii)
> (zece z)
10

; evaluare (lucreaza cu valoarea expresiei)
> z
10

```

Când evaluăm (zece z) Lisp îl interpretează pe zece ca pe un nume de macro și construiește macro-expandarea

```
(list 'setq var 10) ==> (setq z 10)
```

care este apoi evaluată în locul apelului original către macro-ul zece.

Deci apelul de forma (zece z) este transformat în (setq z 10) înainte de a fi compilat sau evaluat.

Pentru a vedea codul generat de macro-ul definit de noi folosim:

```

> (macroexpand-1 '(zece z))
(SETQ Z 10) ;
T

```

Diferența dacă folosim defun function

```

> (defun zece2 (vars)
  (list 'setq vars 100))
ZECE2

> (zece2 '1)
(SETQ L 100)

> (eval (zece2 '1))
100

> 1
100

```

1.1 EVALUARE ARGUMENTE

- funcțiile scrise cu defun își evaluează toate argumentele
- construcțiile macro nu își evaluează argumentele (scriu un cod în program care se poate vedea cu macroexpand)
- într-un apel de funcție definită de utilizator parametri actuali sunt evaluați înainte ca parametri formali din definiție să se lege la aceștia

Concluzie:

NU poate fi realizată o evaluare diferențiată a parametrilor printr-o definiție de funcție.

1.2 EXPANDARE

```
> (defun f (x y)
      (list '+ x y)
    )
F
```

```
> (f 2 3)          ; functia f construiește o lista
(+ 2 3)
```

```
> (defmacro f (x y)
      (list '+ x y)
    )
F
```

```
> (f 2 3)          ; macroul întoarce rezultatul evaluării primei evaluări
5
```

; pentru a vedea pașii folosim următoarele:

```
> (macroexpand '(f 2 3))
(+ 2 3) ;      ; prin evaluare, expandarea, poate fi apelată explicit
T          ; folosind macroexpand
```

```
> (eval (macroexpand '(f 2 3)))
5          ; echivalent cu apel macro
```

1.3 Backquote

Putem utiliza o facilitare prin care într-o expresie anumite fragmente sunt totuși evaluate: *(backquote)*(')

Comportament:

- la fel cu quote;
- devine util atunci când apar combinații cu ,(comma) @ (comma-at)
- ' este close-quote; iar ‘ este open-quote

Exemplu:

```
> ‘(a b c)
(A B C)
```

1.3.1 Caracterul virgulă (,)

Anumite forme pe care dorim să le evaluăm sunt precedate de caracterul virgulă (,):

```
> '(a b c)
(A B C)
```

```
> (setq a 10 b 20 c 30)
30
```

```
> '(a este ,a si b este ,b)
(A ESTE 10 SI B ESTE 20)
```

```
> '(a ,b c)
(A 20 C)
```

```
> '(a b c)
(A B C)
```

```
> '(a ,b c)
*** - READ: comma is illegal outside of backquote
The following restarts are available:
ABORT          :R1      Abort main loop
```

;DECI nu putem folosi ' in combinatie cu ,

```
> '(a ,b ,c)
(A 20 30)
```

```
> '(a ,(+ a b c) c)
(A 60 C)
```

Efectul backquote este anulat de , (trebuie să respectăm corespondența lor).
NU scriem ceva de genul:

```
'(,(+ ,a b) c 30)
```

pentru că a doua virgula nu are un backquote corespondent.

Backquote este folosit de obicei la construirea listelor (citim mai ușor expresiile). Îl folosim în loc de apelul list;
De exemplu rescriem funcția *zece* de mai sus

```
> (defmacro zece (var) (list 'setq var 10))
```

```
> (defmacro zece (var) '(setq ,var 10))
```

1.3.2 Comma-at @

Nuanțare:

```
> (setq l '(x y z))
(X Y Z)
```

```
> '(a ,l ,z)
(A (X Y Z) 10)
```

@ este util în macro-uri atunci când avem parametri rest (ex. while -mai jos).

@ se folosește pentru înglobarea listei în rezultat

Exemplu:

(practic folosirea lui append în loc de cons în construirea rezultatului)

```
> '(a ,@l ,c)
(A X Y Z 30)
```

```
> l          ; l ramane neschimbat
(X Y Z)
```

Exemplu pentru înglobarea distructivă:

(practic folosește nconc în loc de append)

```
> '(a ,.l ,c)
(A X Y Z 30)
```

```
> l
(X Y Z 30)
```

1.4 CAND FOLOSIM MACRO-URI ???

ORICE OPERATOR CARE TREBUIE SĂ ACCESEZE PARAMETRI ÎNAINTE
CA ACEȘTIA SĂ FIE EVALUAȚI TREBUIE SCRIS CA MACRO!!!

Există cazuri când bucăți de cod pot fi scrise atât utilizând funcții, cât și macro-uri.

Exemple:

1.4.1 ori3

```
> (defun ori3 (x) (* 3 x))
ORI3
```

```
> (ori3 10)
30
```

```
> (ori3 0)
0
```

```
> (ori3 -10)
-30
```

```
> (setq x 11)
11
```

```
> (defmacro ori3m (x) `(* 3 ,x))
ORI3M
```

```
> (ori3m x)
33
```

```
> x
11
```

1.4.2 MY-IF

Ne propunem să realizăm o funcție care să se comporte ca if.

utilizand defun (o functie care foloseste if)

```
> (defun my-if (test da nu)
  (if test da nu))
MY-IF
```

```
> (my-if t (setq x 'da) (setq x 'nu))
DA
```

```
> x      ; !!!!!
NU
```

```
> (my-if '(< 6 7) (setq s 'DA) (setq s 'NU))
DA
```

```
> s      ; !!!!!
NU
```

Funcția NU satisface cerința. La intrarea în funcție toți cei 3 parametri sunt evaluați.

Dacă vrem să setăm lui x valoarea DA în cazul în care test se evaluează la true și NU dacă test se evaluează la nil. Acest lucru nu este posibil cu ajutorul funcției.

utilizand defmacro

```
> (defmacro daca (test da nu)
  '(if ,test ,da ,nu))
DACA
```

```
> (daca t (setq x 'da) (setq x 'nu))
DA
```

```
> x
DA      ; OK
```

Remarci:

- Un macro se expandează și apoi se evaluează
 1. pasul macroexpandării operează cu expresii
 2. pasul evaluării operează cu valorile lor
- macroexpandările se vor face de către compilator la momentul compilării

- un apel de macro ce apare în corpul definiției unei funcții va fi expandat la momentul compilării funcției
- expresia (sau codul obiect rezultat) nu va fi evaluată decât în momentul când funcția e apelată

Concluzie:

Nu putem trata apeluri recursive de macro-uri, compilatorul expandează apelul interior înainte de evaluare, ceea ce duce la un alt apel macro, care trebuie și el expandat și tot așa; este imposibil de controlat un asemenea proces.

1.4.3 NTHMCR, FACTORIAL

```
> (defmacro nthmcr (n lista)
  '(if (= ,n 0) (car ,lista)
        (nthmcr (- ,n 1) (cdr ,lista))))
NTHMCR
; expandare
> (nthmcr 3 '(a b c d e))
D
; evaluare

> (defmacro factorial (n)
  '(if (zerop ,n) 1
        (* ,n (factorial (- ,n 1)))))
FACTORIAL
; expandare

> (factorial 4)
24
; evaluare

> (factorial 30)
265252859812191058636308480000000
```

1.4.4 WHEN

```
> (defmacro my_when (primexecut testare &rest corp)
  '(progn ,primexecut (if ,testare (progn ,@corp))))
MY-WHEN

> (my_when (print 'valoarea?)
            (setq val (read))
            (print 'este_bun)
            (print val)
            nil)

VALOAREA? 45

ESTE_BUN
45
NIL
```

1.4.5 WHILE

WHILE nu poate fi scris decat cu macro:

Exemplu: WHILE bazat pe DO folosind @

```
> (let ((x 0))
    (while (< x 10)
      (princ x)
      (incf x)
    )
  )
0123456789
NIL
```

Vrem un macro while care își evaluează corpul său atâta timp cât condiția de test e true.

Folosim un parametru rest care colectează lista expresiilor în corp, apoi @ înglobează lista în rezultatul expansiunii:

```
> (defmacro while (test &rest corp)
  '(do ()
      ((not ,test))
      ,@corp
    )
  )
WHILE
```

Integrează expresiile pe care le primește ca și rest în corpul lui do unde ele vor fi evaluate doar dacă test întoarce true.

Astfel,

```
(while (< x 10) (prin1 x) (princ " ") (setq x (1+ x))))
devine
(do () ((not (< x 10))) (prin1 x) (princ " ")
      (setq x (1+ x))))
```

TESTE:

```
> (setq x 0)
0

> (while (< x 10) (prin1 x) (princ " ") (setq x (1+ x)))
0 1 2 3 4 5 6 7 8 9
NIL

> x
10

> (setq x 0)
0
```



```
> (do () ((not (< x 10))) (prin1 x) (princ " ")
                                     (setq x (1+ x)))
0 1 2 3 4 5 6 7 8 9
NIL
```

1.4.6 Alte exemple

```
> (setq something 'some-value)
SOME-VALUE

> something
SOME-VALUE

> (defun demo-macro (something)
    (print something))
DEMO-MACRO

> (demo-macro something)

SOME-VALUE
SOME-VALUE

> (demo-macro 'some)

SOME
SOME

;——
> (defun demo-fn (something)
    (print something))
DEMO-FN

> (demo-fn something)

> (demo-fn 'some)

;——

> (defmacro suma2 (x y)
    '(+ ,x ,y))
SUMA2

> (suma2 x 8)
18

> (defmacro suma (&rest numere)
    '(+ ,@numere))
SUMA
```

```
> (suma 1 2 3 4 5)
15
```

*;cum scriem o definitie de functie care sa
;aiba acelasi comportament?*

REMARCA:

- De fiecare data cand definiti un macro, defapt definiti un nou limbaj si scrieti un nou compilator pentru el
- Macro-urile Lisp sunt puternice, deoarece "codul" si "datele" au aceeasi forma de reprezentare;
- DECI acordati mai multa atentie macrourilor deoarece a scrie macro-uri inseamna a gandi ca un "language designer". [GRAHAM]

2 PROBLEME SPECIFICE ÎN SCRIEREA MACROURILOR

- Captura variabilelor
- Evaluarea repetata

Exemple:

```
(defmacro ntimes (n &rest corp)      ; Gresit! De ce?
  '(do ((x 0 (+ x 1)))
        ((>= x ,n))
        ,@corp))
```

NTIMES

```
> (ntimes 3 (princ "."))
...
NIL
> (ntimes 5 (princ "."))
.....
NIL
```

Functioneaza!!!!

DAR e gresita: -greseala: captura variabilelor (definitia creeaza o variabila x), deci daca macro-ul e apelat intr-un loc unde exista deja o variabila cu același nume x, atunci rezultatul nu va fi cel asteptat.

Exemplu:

```
> (let ((x 10))
  (ntimes 5 (setf x (+ x 1))))
10
; ar fi trebuit sa fie 15 rezultatul

> (let ((x 10))
  (ntimes 5 (princ ".") (setf x (+ x 1))))
```

```

x)
...
10

; daca utilizam alt nume, nu x, obtinem rezultatul
; dorit (asta in cazul in care nu mai avem un alt simbol
; in program cu numele p)

> (let ((p 10))
    (ntimes 5 (setf p (+ p 1)))
    p)
15

;; macroexpandarea

> (macroexpand-1 '(ntimes 5 (setf x (+ x 1))))
(DO ((X 0 (+ X 1))) ((>= X 5)) (SETF X (+ X 1))) ;
T

```

o solutie ar fi

Folosirea:

- *gensym* generează un nume de simbol “disponibil” (nu există nici o şansă ca numele simbolului generat să fie același cu unul deja existent în program);
- *symbol – name* returnează numele unui anumit simbol

```

> (defmacro ntimes (n &rest corp)           ; Gresit! De ce?
  (let ((g (gensym)))
    ;; (print (symbol-name g))
    `(do ((,g 0 (+ ,g 1)))
        ((>= ,g ,n))
        ,@corp)))

```

Totuși are probleme din cauza evaluării repetate deoarece primul argument este inserat direct în do, el va fi evaluat la fiecare iterație.

Exemplu:

```

> (let ((v 10))
    (ntimes (setf v (- v 1))
            (princ ".") ))
.....
NIL

```

v este inițial 10, rezultatul lui setf este 9, deci ar trebui să afișeze 9 puncte, dar afișează doar 5

Ne uităm la expresia macroexpandării:

```

> (macroexpand-1 '(ntimes (setf v (- v 1))
                          (princ ".") ))

```

```
(DO ((#:G3377 0 (+ #:G3377 1))) ((>= #:G3377
                                     (SETF V (- V 1)))) (PRINC ".") ) ;
```

T

*; La fiecare pas se compara o expresie #:G3377 nu cu 9,
; ci cu un v care scade de cate ori e evaluat.*

Soluția pentru a evita evaluarea multiplă este

Să setăm unei variabile valoarea în discuție înainte de orice iterație:

```
> (defmacro ntimes (n &rest corp)
  (let ((g (gensym))
        (h (gensym)))
    `(let ((,h ,n))
      (do ((,g 0 (+ ,g 1)))
          ((>= ,g ,h)
           ,@corp))))

> (setq v 10)
10

> (ntimes v (setq v (decf v)) (princ "."))
.....
NIL

;; expandarea
> (macroexpand-1 '(ntimes (setf v (- v 1))
                          (princ ".")))
(LET ((#:G3407 (SETF V (- V 1))))
  (DO ((#:G3406 0 (+ #:G3406 1))) ((>= #:G3406 #:G3407))
    (PRINC ".") )) ;

T
```

3 Tema

3.1 Studiați următoarele exemple:

Macro dotimes (fara si cu destructuring lambda list):
-expandare, testare, modificare daca e necesar

```
(defmacro do-times (l &body corp)
  (let ((g (gensym)))
    `(do ((, (car l) 0 (+ , (car l) 1))
          (,g ,(cadr l)))
        ((>= ,(car l) ,g) ,(caddr l))
        ,@corp)))

(defmacro do-times ((i n &optional r) &body corp)
  (let ((g (gensym)))
```

```

‘(do ((,i 0 (+ ,i 1))
      (,g ,n))
    ((>= ,i ,g) ,r)
  ,@corp)))

```

3.2 EXERCITIU

Studiați următorul exemplu:

-testare, expandare, modificare dacă e necesar pentru a nu avea erori de scriere.

IFNR - un test numeric cu următoarea configurație:

```

(ifnr (<var><test>)
      (<ramura - >)
      (<ramura 0 >)
      (<ramura + >)
)

```

NU putem folosi o funcție deoarece aceasta își evaluează toate argumentele.

Condiționala pe care ne-o propunem presupune evaluarea testului și doar a uneia din cele trei forme. Deci folosim construcția macro (care nu își evaluează toate argumentele).

```

> (defmacro ifnr (test ramura- ramura0 ramura+)
  (list 'let (list test)
        (list 'cond (append
                  (list (list 'minusp (car test)))
                      ramura-
                  )
          (append
            (list (list 'zerop (car test)))
              ramura0
            )
          (append '(t) ramura+ )
        )
  )
)
IFNR

```

```

> (ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
      ((princ "Ramura_0") 0)
      ((princ "Ramura_pozitiva") x)
)
5
Ramura pozitiva
5

```

```

> (ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
      ((princ "Ramura_0") 0)
)

```

```

      ((princ "Ramura_pozitiva") x)
    )
  -900
  Ramura negativa
  900

```

```

> (ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
      ((princ "Ramura_0") 0)
      ((princ "Ramura_pozitiva") x)
    )
0
Ramura 0
0

```

```

> (macroexpand '(ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
      ((princ "Ramura_0") 0)
      ((princ "Ramura_pozitiva") x)
    )
  )
(LET ((X (READ)))
  (COND ((MINUSP X) (PRINC "Ramura_negativa") (- X))
        ((ZEROP X) (PRINC "Ramura_0") 0)
        (T (PRINC "Ramura_pozitiva") X))) ;
T

```

*; in exemplul anterior folosim backquote
 ;; Utilizand facilitatile (' , @) putem rescrie ifnr*

```

(defmacro ifnr (test ramura- ramura0 ramura+)
  '(let ( ( ,(car test) ,(cadr test))
    (cond ((minusp ,(car test))
      ,@ramura-
      ((zerop ,(car test))
        ,@ramura0
        (t ,@ramura+)
      )
    )
  )
)

```

```

> (ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
      ((princ "Ramura_0") 0)
      ((princ "Ramura_pozitiva") x)
    )
5

```

```
Ramura pozitiva
5
```

```
> (ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
      ((princ "Ramura_0") 0)
      ((princ "Ramura_pozitiva") x)
    )
-900
Ramura negativa
900
```

```
> (ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
      ((princ "Ramura_0") 0)
      ((princ "Ramura_pozitiva") x)
    )
0
Ramura 0
0
```

```
;;; DESTRUCTURARE


---


;;; si mai simplu
```

```
(defmacro ifnr ((var val) ramura- ramura0 ramura+)
  '(let ((,var ,val))
    (cond ((minusp ,var) ,@ramura-)
          ((zerop ,var) ,@ramura0)
          (t ,@ramura+))
  )
)
IFNR
```

```
> (macroexpand '(ifnr (x (read))
  ((princ "Ramura_negativa") (- x))
  ((princ "Ramura_0") 0)
  ((princ "Ramura_pozitiva") x)
))
(LET ((X (READ)))
  (COND ((MINUSP X) (PRINC "Ramura_negativa") (- X))
        ((ZEROP X) (PRINC "Ramura_0") 0)
        (T (PRINC "Ramura_pozitiva") X))) ;
T
```

```
> (ifnr (x (read))
      ((princ "Ramura_negativa") (- x))
```

```

      ((princ "Ramura_0") 0)
      ((princ "Ramura_pozitiva") x)
    )
  -980
  Ramura negativa
  980

```

3.3 Suplimentar

Studiați următoarele:

```

(defmacro -prog1 (arg1 &rest args)
  (let ((g (gensym)))
    ‘(let ((,g ,arg1))
      ,@args
      ,g)))

(defmacro -prog2 (arg1 arg2 &rest args)
  (let ((g (gensym)))
    ‘(let ((,g (progn ,arg1 ,arg2)))
      ,@args
      ,g)))

(defmacro -progn (&rest args) ‘(let nil ,@args))

(defun -rem (n m)
  (nth-value 1 (truncate n m)))

(defun -stringp (x) (typep x 'string))

(defmacro -dolist ((var lst &optional result) &rest body)
  (let ((g (gensym)))
    ‘(do ((,g ,lst (cdr ,g)))
      ((atom ,g) (let ((,var nil)) ,result))
      (let ((,var (car ,g)))
        ,@body))))

(defmacro -unless (arg &rest body)
  ‘(if (not ,arg)
    (progn ,@body)))

(defmacro -when (arg &rest body)
  ‘(if ,arg (progn ,@body)))

```