Functional Programming – Laboratory 11
MACROS

Isabela Drămnesc

May 28, 2013

1 MACROS

DEFINITIONS

- A MACRO is a functions which defines a function.
- By defining MACROS we can extend the Lisp syntax.
- MACROS are functions with a special evaluation mechanism in two steps:
  1. expand
  2. eval

Step 1) In the definition an expression is built (macro-expand)
- the definition contains macro-name, the-list-of-parameters, body
Step 2) The expression is evaluated

defmacro is similar with defun

- - both give a name to a group of forms
- - allow the use of variables

Macros are functions which modify expressions.

By using macros we can control the evaluation of the arguments when calling the macro.

But, there are several differences between functions and macros:

- a function generates a result
- a macro generates expressions, and after the evaluation of these expressions the result is generated

Examples:

```
; when using defmacro  = defines how the call should be transformed
> (defmacro ten (var) (list 'setq var 10))
TEN
                                ;expand (works with expressions)
> (ten z)                      ;evaluate (works with the value of the expression)
10

> z
10
```

When we evaluate (ten z) Lisp finds ten as a name of a macro and builds the macro-expand

```
(list 'setq var 10) ⟹ (setq z 10)
```

Therefore, the call of the form (ten z) is transformed in (setq z 10) before being compiled or evaluated.

In order to see the code generated by our macro we use:

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

The difference when using a function

```
> (defun ten2 (vars)
          (list 'setq vars 100))
TEN2

> (ten2 'l)
(SETQ L 100)

> (eval (ten2 'l))
100

> l
100
```

1.1 THE EVALUATION OF ARGUMENTS

- the functions defined using defun evaluate all their arguments
- macros do not evaluate their arguments (write a code in the program which can be seen using macroexpand)
- in a user defined function call the actual parameters are evaluated before binding the formal parameters

Conclusion:
We cannot realize a separate evaluation of parameters if we define functions.

1.2 EXPAND

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

> (f 2 3)              ; the function f builds a list
(+ 2 3)
```

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

> (f 2 3)
5

; in order to see all the steps we use:

> (macroexpand '(f 2 3))
(+ 2 3) ;
T

> (eval (macroexpand '(f 2 3)))
5                    ; equivalent to macro call
```

1.3 Backquote

(*backquote*)(‘)

Behavior:

- the same like quote;
- we can have combinations with ,(comma) @ (comma-at)
- ′ is a closed-quote; but ‘ is an open-quote

Example:

```
> ‘(a b c)
(A B C)
```

1.3.1 Comma (,)

If we want to evaluate certain form we use (,) before them:

```
> ‘(a b c)
(A B C)

> (setq a 10 b 20 c 30)
30

> ‘(a is ,a and b is ,b)
(A IS 10 AND B IS 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
```

;THEREFORE we cannot use ' in combination to ,

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

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

The effect of backquote is canceled by , (but we have to respect their correspondence).

DO NOT write something like:

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

because the second comma does not have a corresponding backquote.

Backquote is used for building lists (we can easily read the expressions). We can use it instead of list;
For example we rewrite the function above *ten*

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

> (defmacro ten (var) `(setq ,var 10))
```

1.3.2 Comma-at @

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

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

@ is useful in macros when we have parameters rest (ex. while -below).
@ is used for embedding lists in the result
Example:
(like the use of append instead of cons)

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

> l          ; l stays unchanged
(X Y Z)
```

An example for the destructive embedding:
(like the use of nconc instead of append)

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

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

1.4 WHEN DO WE USE MACROS ???

ANY OPERATOR WHICH HAS TO ACCESS ITS PARAMETERS BEFORE EVALUATING THEM, HAS TO BE WRITTEN AS A MACRO!!!

Example:

1.4.1 times3

```
> (defun times3 (x) (* 3 x))
TIMES3

> (times3 10)
30

> (times3 0)
0

> (times3 -10)
-30

> (setq x 11)
11

> (defmacro times3m (x) `(* 3 ,x))
TIMES3M

> (times3m x)
33

> x
11
```

1.4.2 MY-IF

Let's write a function which behaves like if.

using defun (a function which uses if)

```
> (defun my-if (test yes no)
       (if test yes no))
MY-IF

> (my-if t (setq x 'yes) (setq x 'no))
YES

> x      ; !!!!!
```

```
NO

> (my-if '(< 6 7) (setq s 'yes) (setq s 'no))
YES

> s     ; !!!!!
NO
```

This function does not satisfy the request. All the 3 parameters of the function are evaluated.

But we want to set to x the value YES in the case when test evaluates to true and NO if test evaluates to nil. This is not possible when using functions.

using defmacro

```
> (defmacro if-m (test yes no)
    `(if ,test ,yes ,no))
IF-M

> (if-m t (setq x 'yes) (setq x 'no))
YES

> x
YES             ; OK
```

1.4.3 NTHMCR, FACTORIAL

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

> (defmacro factorial (n)
    `(if (zerop ,n) 1
                 (* ,n (factorial (- ,n 1)))))
FACTORIAL                                 ; expand

> (factorial 4)
24                                        ; eval

> (factorial 30)
265252859812191058636308480000000
```

1.4.4 WHEN

```
> (defmacro my_when (primexecut test &rest corp)
`(progn ,primexecut (if ,test (progn ,@corp))))
MY_WHEN
```

```
> (my_when (print 'value?)
                (setq val (read))
                (print 'is_good)
                (print val)
                nil)

VALUE? 45

IS_GOOD
45
NIL
```

1.4.5 WHILE

WHILE can be written only by using macro:
Example: WHILE based on DO using @

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

We want a macro while which repeatedly evaluates its body until the test condition is true.

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

Astfel,

```
(while (< x 10) (prin1 x) (princ "␣")(setq x (1+ x))))
```

devine

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

TESTS:

```
> (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 Other examples

```
> (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 sum2 (x y)
      `(+ ,x ,y))
```

```
SUM2

> (sum2 x 8)
18

> (defmacro sum (&rest numbers)
    `(+ ,@numbers))
SUM

> (sum 1 2 3 4 5)
15
        ;how do we write a function definition
        ;which behaves the same?
```

REMARK:

- Each time you define a macro, actually you define a new language and you write a new compiler for it
- Macros in Lisp are strong, because the "code" and the "data" are represented the same;
- Therefore, write more macros because this means thinking like a "language designer". [GRAHAM]

2 SPECIFIC PROBLEMS IN WRITING MACROS

- Variables capture
- re-evaluation

Example:

```
(defmacro ntimes (n &rest corp)        ;Wrong! Why?
    `(do ((x 0 (+ x 1)))
         ((>= x ,n))
         ,@corp))
NTIMES

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

Works!!!!
BUT is wrong: -the mistake: the variable capture (the definition creates a variabile x), therefore if the macro is called in a place where already exists a variable with the same name x, then we might have an unexpected result.
Example:

```
> (let ((x 10))
     (ntimes 5 (setf x (+ x 1)))
     x)
10                ; the result should be 15

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

;if we use other name, except x, we get the desired result
; (this will happen if we do not have another symbol p
; in the program)

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

;; macroexpand

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

one solution can be

The use of:

- *gensym* which generates an "available" name of a symbol (there is no chance to generate an existing symbol name in the program);
- $symbol - name$ returns the name of a certain symbol

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

Repeated evaluation (the first argument is directly inserted in do, and it will be evaluated at each iteration).
Example:

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

the initial value of v is 10, the result of setf 9, therefore the result has to be 9 points, but prints only 5
Let's see the macroexpand:

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

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

; At each step compares an expression #:G3377 not with 9,
; but with v which decreases each time is evaluated.
```

The solution in order to avoid the multiple evaluation is

To set the current value to another variable:

```
> (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 Homework

3.1 Study the following examples:

Macro dotimes (with and without destructuring lambda list):
-expand, test, modify if necessary

```
(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 Extra

Study the following:

```
(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)))
```