Robot Programming with Lisp

4. More Functional Programming: Closures, Recursion, Macros

Gayane Kazhoyan

Institute for Artificial Intelligence
University of Bremen

November 16th, 2017

Contents

Concepts

- Closures
- Recursion
- Macros

Organizational

Closures

Counter

```
CL-USER> (defun increment-counter ()
           (let ((counter 0))
             (incf counter)))
         (increment-counter)
         (increment-counter)
1
CL-USER> (defun increment-counter-closure ()
           (let ((counter 0))
             (lambda () (incf counter))))
INCREMENT-COUNTER-CLOSURE
CL-USER> (let ((function-object (increment-counter-closure)))
           (format t "counting: ~a ~a~%"
                   (funcall function-object) (funcall function-object)))
counting: 1 2
```

Closure is a function that, in addition to its specific functionality, also encloses its lexical environment (environment as in, e.g., terminal environment variables).

Closures [2]

Counter Again

```
CL-USER> (defun increment-counter-lambda ()
           (let ((counter 0))
             (lambda (counter) (incf counter))))
INCREMENT-COUNTER-LAMBDA
CL-USER> (let ((function-object (increment-counter-lambda)))
           (format t "counter: ~a~%" (funcall function-object 0))
           (format t "once more: ~a~%" (funcall function-object 0)))
counter: 1
once more: 1
CL-USER> (let ((function-object (increment-counter-closure)))
           (format t "counter: ~a~%" (funcall function-object))
           (setf counter 0)
           (format t "counter: ~a~%" (funcall function-object)))
counter: 1
counter: 2
```

Encapsulation!

Contents

Concepts

- Closures
- Recursion
- Macros

Organizational

Recursion

Primitive Example

```
CL-USER> (defun dummy-recursion (my-list)
           (when my-list
             (dummy-recursion (rest my-list))))
DUMMY-RECURSION
CL-USER> (trace dummy-recursion)
         (dummy-recursion '(1 2 3 4 5))
  0: (DUMMY-RECURSION (1 2 3 4 5))
    1: (DUMMY-RECURSION (2 3 4 5))
      2: (DUMMY-RECURSION (3 4 5))
        3: (DUMMY-RECURSION (4 5))
          4: (DUMMY-RECURSION (5))
            5: (DUMMY-RECURSION NIL)
            5: DUMMY-RECURSION returned NIL
          4: DUMMY-RECURSION returned NIL
        3: DUMMY-RECURSION returned NIL
      2: DUMMY-RECURSION returned NIL
    1: DUMMY-RECURSION returned NIL
  0: DUMMY-RECURSION returned NIL
```

Recursion [2]

Primitive Example #2

```
CL-USER> (defun print-list (list)
           (format t "Inside (print-list ~a)... " list)
           (when list
             (format t "~a~%" (first list))
             (print-list (rest list))))
PRINT-LIST
CL-USER> (print-list '(1 2 3))
Inside (print-list (1 2 3))... 1
Inside (print-list (2 3))... 2
Inside (print-list (3))... 3
Inside (print-list NIL)...
CL-USER> (mapl (lambda (list)
                 (format t "List: ~a... ~a~%" list (first list)))
               '(1 2 3))
List: (1 2 3)... 1
List: (2 3)... 2
List: (3)... 3
(1 2 3)
```

Recursion [3]

Length of a List

```
CL-USER> (defun my-length (a-list)
           (if (null a-list)
               0
               (+ 1 (my-length (rest a-list)))))
MY-LENGTH
CL-USER> (trace my-length)
         (my-length '(5 a 3 8))
  0: (MY-LENGTH (5 A 3 8))
    1: (MY-LENGTH (A 3 8))
      2: (MY-LENGTH (3 8))
        3: (MY-LENGTH (8))
          4: (MY-LENGTH NIL)
          4: MY-LENGTH returned 0
        3: MY-LENGTH returned 1
      2: MY-LENGTH returned 2
    1: MY-LENGTH returned 3
  0: MY-LENGTH returned 4
4
```

Recursion [4]

Tail Recursion Optimization

```
CL-USER> (defun my-length-inner (a-list accumulator)
           (if (null a-list)
               accumulator
               (my-length-inner (rest a-list) (1+ accumulator))))
MY-LENGTH-INNER
CL-USER> (my-length-inner '(5 a 3 8) 0)
4
CL-USER> (defun my-length-optimal (a-list)
           (my-length-inner a-list 0))
MY-LENGTH-OPTIMAL
CL-USER> (trace my-length-inner)
(MY-LENGTH-INNER)
CL-USER> (my-length-optimal '(5 a 3 8))
...
CL-USER> (untrace my-length my-length-inner)
T
```

Recursion [5]

Tail Recursion Optimization: Second Try

```
CL-USER> ,
restart-inferior-lisp
CL-USER> (proclaim '(optimize speed))
; No value
CL-USER> (defun my-length-inner (a-list accumulator)
           (if (null a-list)
               accumulator
               (my-length-inner (rest a-list) (1+ accumulator))))
CL-USER> (defun my-length-optimal (a-list)
           (my-length-inner a-list 0))
CL-USER> (trace my-length-optimal my-length-inner)
(MY-LENGTH-OPTIMAL MY-LENGTH-INNER)
CL-USER> (my-length-optimal '(5 a 3 8))
  0: (MY-LENGTH-OPTIMAL (5 A 3 8))
    1: (MY-LENGTH-INNER (5 A 3 8) 0)
    1: MY-LENGTH-INNER returned 4
  0: MY-LENGTH-OPTIMAL returned 4
4
```

Recursion [6]

What Does This Function Do?

```
CL-USER> (defun sigma (n)
           (labels ((sig (c n)
                      (declare (type fixnum n c))
                      (if (zerop n)
                          c
                          (sig (the fixnum (+ n c))
                               (the fixnum (- n 1))))))
             (sig 0 n)))
SIGMA
CL-USER> (trace sigma)
(SIGMA)
CL-USER> (sigma 5)
  0: (SIGMA 5)
  0: SIGMA returned 15
15
```

(**declare** (type typespec var*))

(**the** return-value-type form)

Contents

Concepts

- Closures
- Recursion
- Macros

Organizational

Generating Code

Backquote and Coma

```
CL-USER> '(if t 'yes 'no)
(IF T
    'YES
    'NO)
CL-USER> (eval *) ; do not ever use EVAL in code
YES
CL-USER> `(if t 'yes 'no)
(IF T
    'YES
    'NO)
CL-USER> `((+ 1 2) ,(+ 3 4) (+ 5 6))
((+ 1 2) 7 (+ 5 6))
CL-USER> (let ((x 26))
           `(if ,(oddp x)
                'yes
                'no))
?
```

Generating Code

Backquote and Coma

```
CL-USER> '(if t 'yes 'no)
(IF T
    'YES
    'NO)
CL-USER> (eval *) ; do not ever use EVAL in code
YES
CL-USER> `(if t 'yes 'no)
(IF T
    'YES
    'NO)
CL-USER> `((+ 1 2) ,(+ 3 4) (+ 5 6))
((+ 1 2) 7 (+ 5 6))
CL-USER> (let ((x 26))
           `(if ,(oddp x)
                'yes
                'no))
    (IF NIL
        'YES
        'NO)
```

Generating Code [2]

Double Quote

```
CL-USER> ''(+ 1 5)
'(+ 1 5)
CL-USER> (eval *)
(+ 1 5)
CL-USER> (eval *)
6
CL-USER> '`(a ,(+ 1 2))
`(A ,(+ 1 2))
CL-USER> (eval *)
(A 3)
CL-USER> `'(a ,(+ 1 2))
'(A 3)
```

Defining Macros

defmacro

```
CL-USER> (defun x^3-fun (x)
           (format t "type of X is ~a~%" (type-of x))
           (* x x x))
CL-USER> (x^3-fun 4)
type of X is (INTEGER 0 4611686018427387903)
64
CL-USER> (defmacro x^3-macro (x)
           (format t "type of X is ~a~%" (type-of x))
           (* x x x))
CL-USER> (x^3-macro 4)
type of X is (INTEGER 0 4611686018427387903)
64
CL-USER> (x^3-macro (+ 2 2))
type of X is CONS
; #<SIMPLE-TYPE-ERROR expected-type: NUMBER datum: (+ 2 2)>.
CL-USER> (defun use-x^3 (a)
           (x^3-macro a))
type of X is SYMBOL
; caught ERROR: Argument X is not a NUMBER: A
```

Defining Macros [2]

macroexpand

```
CL-USER>  (defmacro x^3-backquote (x)
           (format t "type of X is ~a~%" (type-of x))
           `(* ,x ,x ,x))
CL-USER> (defun use-x^3 (a)
           (x^3-backquote a))
type of X is SYMBOL
STYLE-WARNING: redefining COMMON-LISP-USER::USE-X^3 in DEFUN
CL-USER> (use-x^3 4)
64
CL-USER> (macroexpand '(x^3-backquote 4))
type of X is (INTEGER 0 4611686018427387903)
(* 4 4 4)
CL-USER> (x^3-backquote (+ 2 2))
type of X is CONS
64
CL-USER> (macroexpand '(x^3-backquote (+ 2 2)))
type of X is CONS
(* (+ 2 2) (+ 2 2) (+ 2 2))
```

Defining Macros [3]

defmacro continued

```
CL-USER> (defmacro x^3-let (x)
           (format t "type of X is ~a~%" (type-of x))
           `(let ((z ,x))
              (* z z z)))
CL-USER> (x^3-let (+ 2 2))
type of X is CONS
64
CL-USER> (macroexpand '(x^3-let (+ 2 2)))
type of X is CONS
(LET ((Z (+ 2 2)))
  (* Z Z Z))
T
```

Macros transform code into other code by means of code.

Defining Macros [4]

Macro arguments

```
CL-USER> (defmacro test-macro (&whole whole
                               arg-1
                               &optional (arg-2 1) arg-3)
           (format t "whole: ~a~%" whole)
           (format t "arg-1: ~a~%" arg-1)
           (format t "arg-2: ~a~%arg-3: ~a~%" arg-2 arg-3)
           `',whole)
TEST-MACRO
CL-USER> (macroexpand '(test-macro something))
whole: (TEST-MACRO SOMETHING)
arg-1: SOMETHING
arg-2: 1
arg-3: NIL
'(TEST-MACRO SOMETHING)
CL-USER> (test-macro something)
whole: (TEST-MACRO SOMETHING) ...
(TEST-MACRO SOMETHING)
CL-USER> (eval *)
?
```

Example Macros

Some Built-in Ones

```lisp
; Alt-. on when shows you:
(defmacro-mundanely when (test &body forms)
  `(if ,test (progn ,@forms) nil))

; Alt-. on prog1 shows:
(defmacro-mundanely prog1 (result &body body)
  (let ((n-result (gensym)))
    `(let ((,n-result ,result))
       ,@body
       ,n-result)))

; Alt-. on ignore-errors:
(defmacro-mundanely ignore-errors (&rest forms)
  `(handler-case (progn ,@forms)
     (error (condition) (values nil condition))))
```

Example Macros [2]

More Applications

```
CL-USER> (defmacro get-time ()
           `(the unsigned-byte (get-internal-run-time)))
GET-TIME

CL-USER> (defmacro definline (name arglist &body body)
           `(progn (declaim (inline ,name))
                   (defun ,name ,arglist ,@body)))
DEFINLINE

CL-USER>
*RELEASE-OR-DEBUG*
CL-USER> (defmacro info (message &rest args)
           (when (eq *release-or-debug* :debug)
             `(format *standard-output* ,message ,@args)))
INFO
CL-USER> (info "bla")
bla
```

Advanced Macros

A Better Example

```
CL-USER> (defmacro square (&whole form arg)
           (if (atom arg)
               `(expt ,arg 2)
               (case (car arg)
                 (square (if (= (length arg) 2)
                             `(expt ,(nth 1 arg) 4)
                             form))
                 (expt (if (= (length arg) 3)
                           (if (numberp (nth 2 arg))
                               `(expt ,(nth 1 arg) ,(* 2 (nth 2 arg)))
                               `(expt ,(nth 1 arg) (* 2 ,(nth 2 arg))))
                           form))
                 (otherwise `(expt ,arg 2)))))
CL-USER> (macroexpand '(square (square 3)))
(EXPT 3 4)
CL-USER> (macroexpand '(square (expt 123 4)))
(EXPT 123 8)
```

Links

- Functional programmer Bible (available for download):
  `http://www.paulgraham.com/onlisp.html`

Info Summary

- Assignment code: `REPO/assignment_5/src/*.lisp`
- Assignment points: 10 points
- Assignment due: 22.11, Wednesday, 23:59 AM German time
- Next class: 23.11, 14:15
- Next class topic: introduction to ROS.
  Please make sure your ROS and roslisp_repl are working.

Q & A

Thanks for your attention!