Robot Programming with Lisp

5. More Functional Programming: Closures, Recursion, Macros

Gayane Kazhoyan

Institute for Artificial Intelligence
University of Bremen

November 15th, 2018
Closures

**Counter**

```lisp
CL-USER> (defun increment-counter ()
  (let ((counter 0))
    (incf counter))
  (increment-counter)
  (increment-counter)
1

CL-USER> (defvar *counter* 0)
(defun increment-counter-function ()
  (incf *counter*)
(increment-counter-function)
(increment-counter-function)
2

CL-USER> (setf *counter* 5)
5
CL-USER> (increment-counter-function)
6
```

Concepts

Gayane Kazhoyan
November 15th, 2018
Closures [2]

**Counter As Closure**

CL-USER> (let ((counter 0))
  (defun increment-counter-closure ()
    (incf counter)))
  (increment-counter-closure)
  (increment-counter-closure)

2

CL-USER> #'increment-counter-function
#<FUNCTION INCREMENT-COUNTER-FUNCTION>

CL-USER> #'increment-counter-closure
#<CLOSURE INCREMENT-COUNTER-CLOSURE>

CL-USER> counter
; Evaluation aborted on #<UNBOUND-VARIABLE COUNTER {10104CE223}>

*Closure* is a function that, in addition to its specific functionality, also encloses its lexical environment.

→ Encapsulation!

---

**Concepts**

Gayane Kazhoyan

November 15th, 2018

**Organizational**

Robot Programming with Lisp
Creating Closures

CL-USER> (let ((input (read)))
  (lambda () (print input)))
"some long sentence or whatever"
#<CLOSURE (LAMBDA ()) {10108F062B}>

CL-USER> (funcall *)
"some long sentence or whatever"

CL-USER> (alexandria:curry #'expt 10)
#<CLOSURE (LAMBDA (&REST ALEXANDRIA...) :IN ALEXANDRIA...) {10040F1D8B}>

CL-USER> (funcall * 3)
1000

CL-USER> (defvar *input* (read))
hello
*INPUT*

CL-USER> (lambda () (print *input*))
#<FUNCTION (LAMBDA ()) {100424317B}>

Concepts

Gayane Kazhoyan
November 15th, 2018
Contents

Concepts
  Closures
  Recursion
  Macros

Organizational

Gayane Kazhoyan
November 15th, 2018
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

Concepts Organizational

Gayane Kazhoyan
November 15th, 2018

Robot Programming with Lisp
Recursion [2]

Primitive Example #2

```lisp
(defun print-list (list)
  (format t "list: ~a" list)
  (when list
    (format t " -> first: ~a~%" (first list))
    (print-list (rest list))))

CL-USER> (print-list '(1 2 3))
list: (1 2 3) -> first: 1
list: (2 3) -> first: 2
list: (3) -> first: 3
list: NIL
NIL

CL-USER> (mapl (lambda (list)
                  (format t "list: ~a -> first: ~a~%" list (first list)))
            '(1 2 3))
list: (1 2 3) -> first: 1
list: (2 3) -> first: 2
list: (3) -> first: 3
'(1 2 3)
```

Concepts

Gayane Kazhoyan
November 15th, 2018
Recursion [3]

Length of a List: calculate on the way up

```lisp
(defun my-length (a-list)
  (if (null a-list)
      0
      (+ 1 (my-length (rest 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)
  ML>(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
```
Recursion [4]

Length of a list: calculate on the way down — Accumulators

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> (trace my-length-inner)
(MY-LENGTH-INNER)
CL-USER> (my-length-inner '(5 a 3 8) 0)
  0: (MY-LENGTH-INNER (5 A 3 8) 0)
    1: (MY-LENGTH-INNER (A 3 8) 1)
      2: (MY-LENGTH-INNER (3 8) 2)
        3: (MY-LENGTH-INNER (8) 3)
          4: (MY-LENGTH-INNER NIL 4)
          4: MY-LENGTH-INNER returned 4
        3: MY-LENGTH-INNER returned 4
      2: MY-LENGTH-INNER returned 4
    1: MY-LENGTH-INNER returned 4
  0: MY-LENGTH-INNER returned 4
Recursion [5]

Length of a list: passing initial accumulator value

```lisp
CL-USER> (defun my-length-outer (a-list)
  (my-length-inner a-list 0))
MY-LENGTH-ACC
CL-USER> (my-length-outer '(5 a 3 8))
4

CL-USER> (defun my-length-acc (a-list &optional (accumulator 0))
  (if (null a-list)
    accumulator
    (my-length-acc (rest a-list) (1+ accumulator))))
MY-LENGTH-ACC
CL-USER> (my-length-acc '(6 3 nj ws))
4
```
Recursion [6]

Tail Recursion Optimization

CL-USER> (trace my-length-acc my-length)
(MY-LENGTH-ACC MY-LENGTH)
CL-USER> (my-length '(a b c))
...
CL-USER> (my-length-acc '(a b c))
...
CL-USER> (proclaim '(optimize speed))
...
CL-USER> (defun my-length-acc (a-list &optional (accumulator 0)) ...)
WARNING: redefining COMMON-LISP-USER::MY-LENGTH in DEFUN
CL-USER> (my-length-acc '(a b c))
 0: (MY-LENGTH-ACC (A B C))
 0: MY-LENGTH-ACC returned 3
3
CL-USER> (my-length '(a b c))
 0: (MY-LENGTH (A B C))
 0: MY-LENGTH returned 3
Recursion [7]

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)

Concepts

Gayane Kazhoyan
November 15th, 2018

Organizational

Robot Programming with Lisp

13
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> `((+ 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

<table>
<thead>
<tr>
<th>CL-USER&gt;</th>
<th>'' (+ 1 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'(+ 1 5)</td>
</tr>
<tr>
<td>CL-USER&gt;</td>
<td>(eval *)</td>
</tr>
<tr>
<td></td>
<td>(+ 1 5)</td>
</tr>
<tr>
<td>CL-USER&gt;</td>
<td>(eval *)</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td>CL-USER&gt;</td>
<td>`(a ,(+ 1 2))</td>
</tr>
<tr>
<td></td>
<td>`(A ,(+ 1 2))</td>
</tr>
<tr>
<td>CL-USER&gt;</td>
<td>(eval *)</td>
</tr>
<tr>
<td></td>
<td>(A 3)</td>
</tr>
<tr>
<td>CL-USER&gt;</td>
<td>`(a ,(+ 1 2))</td>
</tr>
<tr>
<td></td>
<td>'(A 3)</td>
</tr>
</tbody>
</table>
Defining Macros

```lisp
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

Concepts ; caught ERROR: Argument X is not a NUMBER: A
```
Defining Macros [2]

```lisp
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]

Macros transform code into other code by means of code.

Concepts

Gayane Kazhoyan
November 15th, 2018
Defining Macros [4]

Macro arguments

```lisp
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 *)
```

Gayane Kazhoyan

November 15th, 2018
### Example Macros

#### Some Built-in Ones

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(defmacro-mundanely when (test &amp;body forms) `(if ,test (progn ,@forms) nil))</td>
<td>; Alt-. on when shows you:</td>
</tr>
<tr>
<td>(defmacro-mundanely prog1 (result &amp;body body) (let ((n-result (gensym))) `(let ((,n-result ,result)) ,@body ,n-result)))</td>
<td>; Alt-. on prog1 shows:</td>
</tr>
<tr>
<td>(defmacro-mundanely ignore-errors (&amp;rest forms) `(handler-case (progn ,@forms) (error (condition) (values nil condition)))))</td>
<td>; Alt-. on ignore-errors:</td>
</tr>
</tbody>
</table>
Example Macros [2]

More Applications

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

CL-USER> (defmacro defineline (name arglist &body body)
  `(progn (declare (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

```lisp
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)
```
• Functional programmer Bible (available for free):
  http://www.paulgraham.com/onlisp.html
Info Summary

• Assignment code: REPO/assignment_5/src/*.lisp
• Assignment points: 8 points
• Assignment due: 21.11, Wednesday, 23:59 AM German time
• Next class: 22.11, 14:15
Thanks for your attention!