Robot Programming with Lisp

5. Macros, Object-Oriented Programming and Failure Handling

Gayane Kazhoyan

Institute for Artificial Intelligence
Universität Bremen

3rd May, 2016

Outline

Theory
- Macros
- Structures and Hash Tables
- Common Lisp Object System (CLOS)
- Failure Handling

Organizational

Outline

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)
```

Theory Organizational

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 461168601842738790 3)
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 461168601842738790 3)
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
```

Theory Organizational

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 *)
?
```

Theory Organizational

Example Macros

Some Built-in Ones

```
; 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)
```

Outline

Structures

Handling Structs

```
CL-USER> (defstruct player
           id
           (name "mysterious stranger" :type string)
           (hp 10 :type integer)
           (mp 0 :type integer)
           and-so-on)
CL-USER> (defvar *player* (make-player :name "Turtle" :and-so-on '123))
         *player*
#S(PLAYER :ID NIL :NAME "Turtle" :HP 10 :MP 0 :AND-SO-ON 123)
CL-USER> (player-name *)
"Turtle"
CL-USER> (defvar *player-copy* (copy-player *player*))
         (setf (player-name *player-copy*) "Cat")
         *player-copy*
#S(PLAYER :ID NIL :NAME "Cat" :HP 10 :MP 0 :AND-SO-ON SOME-DATA)
CL-USER> *player*
#S(PLAYER :ID NIL :NAME "Turtle" :HP 10 :MP 0 :AND-SO-ON 123)
```

Hash Tables

Handling Hash Tables

```
CL-USER> (defvar *table* (make-hash-table :test 'equal))
*TABLE*
CL-USER> *table*
#<HASH-TABLE :TEST EQUAL :COUNT 0 {100A84AF03}>

CL-USER> (setf (gethash "MZH" *table*) "Bibliothekstrasse 3"
               (gethash "TAB" *table*) "Am Fallturm 1")
"Am Fallturm 1"
CL-USER> (gethash "MZH" *table*)
"Bibliothekstrasse 3"
T
```

Outline

Classes

Handling Classes

```
CL-USER> (defclass shape ()
           ((color :accessor get-shape-color
                   :initarg :set-color)
            (center :accessor shape-center
                    :initarg :center
                    :initform '(0 . 0))))
#<STANDARD-CLASS SHAPE>
CL-USER> (defvar *red-shape* (make-instance 'shape :set-color 'red))
*RED-SHAPE*
CL-USER> (describe *red-shape*)
#<SHAPE {100536B6A3}>
  [standard-object]

Slots with :INSTANCE allocation:
  COLOR    = RED
  CENTER   = (0 . 0)
CL-USER> (get-shape-color *red-shape*)
RED
```

Classes [2]

Inheritance

```
CL-USER> (defclass circle (shape)
           ((radius :initarg :radius)))
#<STANDARD-CLASS CIRCLE>
CL-USER> (defvar *circle*
           (make-instance 'circle :set-color 'green :radius 10))
*CIRCLE*
CL-USER> (describe *circle*)
#<CIRCLE {1005F61973}>
  [standard-object]

Slots with :INSTANCE allocation:
  COLOR    = GREEN
  CENTER   = (0 . 0)
  RADIUS   = 10
CL-USER> (slot-value *circle* 'radius)
10
```

Lisp class vs. Java class

Lisp classes have / support:

- attributes
- getter-setter methods
- multiple inheritance

Lisp classes don't have:

- attribute access specifications (managed with package namespaces)
- methods

Function Overloading: Generic Programming

Defining Generic Functions

```
CL-USER> (defgeneric area (x)
           (:documentation "Calculates area of object of type SHAPE."))
CL-USER> (area 1)
; #<SIMPLE-ERROR "~@<There is no applicable method for ..."
CL-USER> (defmethod area (x)
           (error "AREA is only applicable to SHAPE instances"))
CL-USER> (defmethod area ((obj shape))
           (error "We need more information about OBJ to know its area"))
CL-USER> (defmethod area ((obj circle))
           (* pi (expt (slot-value obj 'radius) 2)))
CL-USER> (area 1)
; #<SIMPLE-ERROR "AREA is only applicable to SHAPE instances">
CL-USER> (area *red-shape*)
; #<SIMPLE-ERROR "We need more information about OBJ to know its area"
CL-USER> (area *circle*)
314.1592653589793d0
```

Function Overloading: Generic Programming [2]

Method combinations: :before, :after, :around

```
CL-USER> (defmethod area :before (obj)
           (format t "Before area. "))
CL-USER> (area *circle*)
Before area.
314.1592653589793d0
CL-USER> (defmethod area :around ((obj shape))
           (format t "Taking over shape area. "))
CL-USER> (area *red-shape*)
Taking over shape area.
CL-USER> (defmethod area :around ((obj shape))
           (format t "Taking over shape area. ")
           (call-next-method))
CL-USER> (area *red-shape*)
Taking over shape area. Before area. ; #<SIMPLE-ERROR "We need ..."
CL-USER> (defmethod area :around ((obj shape))
           (* 2 (call-next-method)))
CL-USER> (area *circle*)
                  Before area.
                  628.3185307179587d0
```

Theory

Organizational

Function Overloading: Generic Programming [3]

Custom :method-combination

```
CL-USER> (defgeneric awesome-function (x)
           (:method-combination +))
#<STANDARD-GENERIC-FUNCTION AWESOME-FUNCTION (0)>
CL-USER> (defmethod awesome-function + ((x number))
           x)
#<STANDARD-METHOD AWESOME-FUNCTION + (NUMBER) {1006E16443}>
CL-USER> (awesome-function 2)
2
CL-USER> (typep 2 'number)
T
CL-USER> (typep 2 'integer)
T
CL-USER> (defmethod awesome-function + ((x integer))
           x)
#<STANDARD-METHOD AWESOME-FUNCTION + (INTEGER) {10072D6323}>
CL-USER> (awesome-function 2)
4
```

OOP in Lisp

Summary

OOP:

- Everything is an object.
- Objects interact with each other.
- Methods "belong" to objects.

Functional programming:

- Everything is a function.
- Functions interact with each other.
- Objects "belong" to (generic) functions.

OOP principles in Lisp:

- inheritance (`defclass`)
- encapsulation (`closures`)
- subtyping polymorphism (`defclass`)
- parametric polymorphism (generic functions)

Outline

Invoking Conditions

define-condition, error

```
CL-USER> (error "oops, something went wrong...")
; #<COMMON-LISP:SIMPLE-ERROR "oops, something went wrong...">.
CL-USER> (define-condition input-not-a-number (simple-error)
           ((actual-input :initarg :actual-input
                          :reader actual-input
                          :initform nil))
           (:report (lambda (condition stream)
                      (format stream "~a is not a number!"
                              (actual-input condition)))))
INPUT-NOT-A-NUMBER
CL-USER> (let ((input (read)))
           (if (numberp input)
               input
               (error (make-condition 'input-not-a-number
                                      :actual-input input))))
asdf
; Evaluation aborted on #<COMMON-LISP-USER::INPUT-NOT-A-NUMBER>.
```

Catching Conditions

handler-case

```
CL-USER> (defparameter *result* nil)
         (let ((x (random 3)))
           (setf *result* (/ 123.0 x))
           (format t "new result is: ~a~%" *result*)
           (setf *result* 0)
           (format t "cleaned up: ~a~%" *result*))
; Evaluation aborted on #<COMMON-LISP:DIVISION-BY-ZERO {1008D6E5B3}>.
CL-USER> (defparameter *result* nil)
         (let ((x (random 3)))
           (handler-case
               (progn
                 (setf *result* (/ 123.0 x))
                 (format t "new result is: ~a~%" *result*)
                 (setf *result* 0)
                 (format t "cleaned up: ~a~%" *result*))
             (division-by-zero (error)
               (format t "      ~a~%" error)))
           (format t "      final result: ~a~%" *result*))
Theory  arithmetic error DIVISION-BY-ZERO signalled                Organizational
      final result: NIL
```

Catching Conditions [2]

unwind-protect

```
CL-USER> (defparameter *result* nil)
         (let ((x (random 3)))
           (handler-case
               (unwind-protect
                    (progn
                      (setf *result* (/ 123.0 x))
                      (format t "new result is: ~a~%" *result*))
                 (setf *result* 0)
                 (format t "cleaned up: ~a~%" *result*))
             (division-by-zero (error)
               (format t "~a~%" error)))
           (format t "final result: ~a~%" *result*))
cleaned up: 0
arithmetic error DIVISION-BY-ZERO signalled
final result: 0
```

Links

- Cool article by Paul Graham on programming languages (a debate on macros included):
  `http://www.paulgraham.com/avg.html`
- "Practical Common Lisp" failure handling chapter:
  `http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html`

Outline

Organizational Info

- Assignment due: 10.05, Tuesday, 08:00 AM German time.
- Assignment points: 10 out of 50.
- Next class: 10.05, 16:15.
- Next class topic: introduction to ROS.
  Please make sure your roslisp_repl is working.

Q & A

Thanks for your attention!