Robot Programming with Lisp

5. Macros, Object-Oriented Programming and Packaging

Gayane Kazhoyan

Institute for Artificial Intelligence
Universität Bremen

11th November, 2014

Outline

Theory
- Macros
- Structures and Hash Tables
- Common Lisp Object System (CLOS)
- Lisp Packages and ASDF Systems

Practice

Outline

Generating Code

Backquote and Coma

```
CL-USER> '(if t 'yes 'no)
(IF T
    'YES
    'NO)
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)
```

Theory Practice

Defining Macros

Macros transform code into other code by means of code.

defmacro and macroexpand

```
CL-USER> (defmacro x^3 (x) (* x x x))
X^3
CL-USER> (x^3 3)
27

CL-USER> (defmacro test-macro (&whole whole arg-1
                               &optional (arg-2 1) arg-3)
           `'(,whole ,arg-1 ,arg-2 ,arg-3 something-else))
TEST-MACRO
CL-USER> (macroexpand '(test-macro some-symbol some-other-symbol))
'((TEST-MACRO SOME-SYMBOL SOME-OTHER-SYMBOL) SOME-SYMBOL
  SOME-OTHER-SYMBOL NIL SOMETHING-ELSE)
```

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> (defparameter *release-or-debug* :debug)
*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 (* 2 3) 4)))
(EXPT (* 2 3) 8)
```

Outline

Structures

Handling Structs

```
CL-USER> (defstruct tcp-ip-packet
           id
           (flags #*001001001 :type bit-vector)
           (checksum 0 :type integer)
           (protocol 6 :type integer)
           and-so-on)
CL-USER> (make-tcp-ip-packet :id 1234 :protocol 4 :and-so-on 'some-data)
#S(TCP-IP-PACKET
   :ID 1234
   :FLAGS #*001001001
   :CHECKSUM 0
   :PROTOCOL 4
   :AND-SO-ON SOME-DATA)
CL-USER> (tcp-ip-packet-id *)
1234
CL-USER> (tcp-ip-packet-p **)
T
CL-USER> (defvar *packet-copy* (copy-tcp-ip-packet ***))
```

Theory Practice

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 (cons 0 0))))
#<STANDARD-CLASS SHAPE>
CL-USER> (make-instance 'shape :set-color 'red)
#<SHAPE {100D7D91C3}>
CL-USER> (describe *)
#<SHAPE {100D7D91C3}>
  [standard-object]
Slots with :INSTANCE allocation:
  COLOR    = RED
  CENTER   = (0 . 0)
CL-USER> (get-shape-color **) ; or (slot-value ** 'color)
RED
```

Classes [2]

Inheritance

```
CL-USER> (defclass circle (shape)
           ((radius :accessor circle-radius
                    :initarg :radius)))
#<STANDARD-CLASS CIRCLE>
CL-USER> (make-instance 'circle :set-color 'green :radius 10)
#<CIRCLE {100DE6AA53}>
CL-USER> (describe *)
#<CIRCLE {100DE6AA53}>
  [standard-object]

Slots with :INSTANCE allocation:
  COLOR    = GREEN
  CENTER   = (0 . 0)
  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."))
STYLE-WARNING: redefining COMMON-LISP-USER::AREA in DEFGENERIC
#<STANDARD-GENERIC-FUNCTION AREA (0)>
CL-USER> (defmethod area (x)
           (error "AREA is only applicable to SHAPE instances"))
#<STANDARD-METHOD AREA (T) {100E0C8F83}>
CL-USER> (defmethod area ((obj shape))
           (error "We need more information about OBJ to know its area"))
#<STANDARD-METHOD AREA (SHAPE) {100E214693}>
CL-USER> (defmethod area ((obj circle))
           (* pi (expt (circle-radius obj) 2)))
#<STANDARD-METHOD AREA (CIRCLE) {100E3FDD03}>
CL-USER> (area (make-instance 'circle :set-color 'green :radius 10))
314.1592653589793d0
```

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

Lisp Packages

Lisp packages define namespaces.
They are used to avoid naming clashes and control access permissions.

Lisp Packages

```
CL-USER> (defun lambda () #\L)
Lock on package COMMON-LISP violated when proclaiming LAMBDA as ...
CL-USER> (defpackage :i-want-my-own-lambda)
CL-USER> (in-package :i-want-my-own-lambda)
#<COMMON-LISP:PACKAGE "I-WANT-MY-OWN-LAMBDA">
I-WANT-MY-OWN-LAMBDA> (cl-user::defun lambda () #\L)
LAMBDA
I-WANT-MY-OWN-LAMBDA> (cl-user::in-package :cl-user)
#<PACKAGE "COMMON-LISP-USER">
CL-USER> (describe *)
#<PACKAGE "COMMON-LISP-USER">
Documentation:
  public: the default package for user code and data
Nicknames: CL-USER
Use-list: COMMON-LISP, SB-ALIEN, SB-DEBUG, SB-EXT, SB-GRAY, SB-PROFILE
```

Theory Practice

Lisp Packages [2]
Defining a Package

defpackage *defined-package-name [[option]] => package*

option::= (:nicknames nickname*)* |
(:documentation string) |
(:use package-name*)* |
(:shadow symbol-name*)* |
(:shadowing-import-from package-name symbol-name*)* |
(:import-from package-name symbol-name*)* |
(:export symbol-name*)* |
(:intern symbol-name*)* |
(:size integer)

Lisp Packages [3]

Example Package Definition

```
CL-USER> (defpackage :homework
           (:nicknames :hw)
           (:documentation "A namespace for my homework assignments")
           (:use :common-lisp))
#<PACKAGE "HOMEWORK">
CL-USER> (in-package :homework)
#<PACKAGE "HOMEWORK">
HW> (defun say-hello () (print "hello"))
HW> (say-hello)
"hello"
HW> (in-package :cl-user)
#<PACKAGE "COMMON-LISP-USER">
CL-USER> (say-hello)
The function COMMON-LISP-USER::SAY-HELLO is undefined.
CL-USER> (hw:say-hello)
The symbol "SAY-HELLO" is not external in the HOMEWORK package.
CL-USER> (hw::say-hello)
"hello"
```

Theory Practice

ASDF Systems

ASDF is Another System Definition Facility:

- It takes care of compiling and "linking" files together in correct order.
- It is also responsible for finding Lisp files across the file system.

ASDF System Definition

```
(in-package :cl-user)
(asdf:defsystem my-system
  :name "My Super-Duper System"
  :description "My Super-Duper System is for doing cool stuff."
  :long-description "Here's how it does cool stuff: ..."
  :version "0.1"
  :author "First Last <email@bla.bla>"
  :licence "BSD"
  :depends-on (alexandria and-another-system)
  :components ((:file "package")))
```

ASDF Systems [2]

ASDF keeps a *registry* of all the paths where it expects to find `.asd` files. A registry is a list of paths.

There are different types of registries: for users, for administrators, etc. But the simplest is to work with the `*central-registry*`.

Managing the Registry

```
CL-USER> asdf:*central-registry*
(#P"/some/path/"
 #P"/some/other/path/")
CL-USER> (push "~/path/to/dir/of/my-system/" asdf:*central-registry*)
("~/path/to/dir/of/my-system/"
 #P"/some/path/"
 #P"/some/other/path/")
CL-USER> (asdf:load-system :my-system)
T
```

The trailing slash is important ("/some/path/")!

Links

- Cool article by Paul Graham on programming languages (a debate on macros included):

  `http://www.paulgraham.com/avg.html`

- ASDF website (for overview, docs, etc.):

  `http://common-lisp.net/project/asdf/`

Outline

Classical Planner

The assignment code can be found in: **`REPO/assignment_5/src`**.

Definitions:
A *state* is a list of conditions, e.g. `(:at-home :be-hungry)`.
An *action* is a mapping from one state to another state.
The problem is to find a sequence of actions that bring you from the initial state to the goal state.
The code consist of:

- `planner.asd`: your ASDF system definition
- `package.lisp`: your Lisp package definition
- `infrastructure.lisp`: infrastructure and helper definitions
- `domain-lisp-course.lisp`: definition of all possible actions
- `planner.lisp`: the actual planner (your ToDo)
- `tests.lisp`: helper functions for testing your solution

Organizational Info

- Assignment due: 17.11, Monday, 23:59 German time.
- Next class: 18.11, 14:15, always room downstairs now (TAB 1.58)
- Next class topic: introduction to ROS.
  Please fix your roslisp_repl installation.

Q & A

Thanks for your attention!