Common Lisp and Introduction to Functional Programming
Lecture 5: CLOS, MOP and Macros

Yuri Zhykin

Mar 11, 2021

Common Lisp Object System

- **CLOS, Common Lisp Object System** originally was a “plug-in” object system for the Common Lisp language, implemented as a library.
- Final version of CLOS is included in the ANSI Common Lisp standard and well integrated into the language.
- CLOS is inspired by the earlier Lisp object systems **MIT Flavors** and **CommonLoops**.
- CLOS is considered one of the most powerful and flexible object systems in mainstream programming languages.

CLOS entities

- **Classes** and **Objects**.
- **Generic Functions**.
- **Methods**.
- **Specializers**.
- **Method Combinations**.

Classes and Objects 1/2

- Classes can be declared with the `defclass` form:

```
(defclass <class-name> (<superclass-1> <superclass-2> ...)
  ((<slot-1>
    :accessor <class-slot-1>
    :initarg  :slot-1
    :initform <slot-1-initform>)
   ...)
  <options>)
```

- Instances of classes (**objects**) can be created with:

```
CL-USER> (make-instance <class-name-symbol>
                        :slot-1 <slot-1-value> ...)
#<<class-name> :slot-1 <slot-1-value> ...>
```

Classes and Objects 2/3

- Authenticator class hierarchy example:

```
(defclass auth ()
  ((user :accessor auth-user :initarg :user)))

(defclass secret-auth (auth)
  ((secret :accessor auth-secret :initarg :secret)))

(defclass password-auth (secret-auth) ())

(defclass token-auth (secret-auth) ())
```

Classes and Objects 3/3

- CLOS classes support multiple inheritance.
- Common Lisp provides another type of record entities - **structs**:

```
CL-USER (defstruct point x y)
point
CL-USER> (make-point :x 1 :y 2)
#S(point :x 1 :y 2)
CL-USER> (point-x *)
1
```

- Structs are also classes, but they are easier to use in some cases and may have different internal representation.

Generic Functions

- **Generic function** is an object that consists of a set of all **methods** that have the same symbolic name.
- *Generic function declarations* are used to fix the *contracts* (*signatures*) of its methods and to store documentation.
- For example:

```
(defgeneric authenticate (auth)
  (:documentation "Authenticate the user with the given AUTH."))
```

Methods

- **Methods** are functions that implement a generic function for a particular set of arguments.
- Methods are added to the method set of a generic function with the same name.
- If generic function with a given name does not exist, it is created automatically.
- Example:

```
(defmethod authenticate ((auth auth))
  nil)

(defmethod print-object ((auth secret-auth) stream)
  (format stream "#<SECRET-AUTH(~a)>" (auth-user auth)))

(defmethod authenticate ((auth password-auth))
  (when (equal (sha512 (auth-secret auth)) <stored-pw-hash>)
    ...))
```

Specializers

- *Applicability* of methods is defined by **specializers** - entities that describe a property of an argument that is sufficient to match a given method.
- Current Common Lisp standard defines two types of specializers
  - **type** - class/inheritance tree of the object is used,
  - **eql** - object is tested for equality with a given specializer.
- Example of `eql`-specializer:

```
(defmethod authenticate ((auth (eql :dummy-auth)))
  t)
```

Method combinations 1/2

- **Method combinations** allow to define how *all* applicable methods will be combined into an **effective method**.
- **Standard method combination** defines the notions of **primary method** and **auxiliary methods**.
- Auxiliary methods can be of the following types:
  - **before** - called before primary method; result discarded,
  - **after** - called after primary method; result discarded,
  - **around** - called around before/primary/after method.
- `:around`-methods must call `call-next-method` function or otherwise method call sequence will be terminated.

Method combinations 2/2

- Standard method combination example:

```
(defmethod authenticate :before ((auth token-auth))
  (load-token-storage))

(defmethod authenticate :after ((auth token-auth))
  (unload-token-storage))

;; Alternatively:
(defmethod authenticate :around ((auth token-auth))
  (load-token-storage)
  (prog1
    (call-next-method)
    (unload-token-storage)))
```

Metaobject Protocol

- **Metaobject** is an object that manipulates, creates, describes, or implements objects (including itself).
- **Metaobject protocol** is an object-based model that consists of *metaobjects* and allows to manipulate the structure and behaviour of an object system:
  - create or delete new classes,
  - create new generic functions and methods,
  - define class slot access,
  - manipulate class inheritance relations,
  - manipulate the **effective method** computation.
- Metaobject protocol can be used to create an additional object system that better fits the domain of the software.

Class Precedence Lists

- Some of the MOP utilities are available in standard Common Lisp but most of MOP is usually available as an additional package:

```
CL-USER> (find-class 'password-auth)
#<standard-class password-auth>
```

- One of the most important MOP concepts is **class precedence list**:

```
CL-USER> (mop:class-precedence-list *)
(#<standard-class password-auth> #<standard-class secret-auth>
 #<standard-class auth> #<standard-class standard-object>
 #<built-in-class t>)
```

Effective Methods

- *Class precedence lists* are used when computing **effective methods**.
- **Effective method** is a function that is actually called when a generic function is called.
- *Effective method* is a *closure* that combines a subset of *applicable methods* based on *class precedence* and *method combinations*:

```
(lambda (&rest args)
  (if <around-methods-exists-p>
      <call-around-methods> ;; calls before/primary/after-methods
      (progn
        <call-before-method>
        <call-primary-methods>
        <call-after-method>))
```

Macros

- **Macros** are special forms that expand into actual Lisp forms during compilation.
- Powerful macro system allows to extend the syntax of the language with the new special forms and control structures.
- More primitive languages (e.g. *C*) use macros that generate text via simple textual substitutions.
- Lisp macros generate Lisp forms represented as data structures (lists), and any Lisp code can be executed when computing the macro expansion.
- **Common Lisp macros are Common Lisp functions that are called during compilations and return Common Lisp code represented as trees that will be further compiled.**

Read/Compile/Load/Evaluate Phases

- **Read** phase reads *textual* representation of Lisp and returns a tree representation for each *top-level* form.
- **Compile** phase recursively expands macros until there is nothing to expand and compiles the result Lisp code.
- **Load** phase loads the compiled Lisp code (usually in the form of `*.fasl` files into the running Lisp executing all top-level forms.
- **Execute** phase is started when a Lisp form is invoked within a Lisp system (e.g. from REPL).

Lisp Code that Returns Lisp Code 1/2

- Simplest Lisp macros can simply construct and return lists that represent the resulting expansion:

```
(defmacro %when (condition &body true-body)
  (list 'if condition (cons 'progn true-body) nil))

CL-USER> (macroexpand '(%when t (print "true") t))
(if t (progn (print "true") t) nil)
t
```

Lisp Code that Returns Lisp Code 2/2

- More concise way to construct Lisp forms is to use the **quasi-quote**:

```
(defmacro %when (condition &body true-body)
  `(if ,condition
       (progn ,@true-body)
       nil))

CL-USER> (macroexpand '(%when t (print "true") t))
(if t (progn (print "true") t) nil)
t
```

Why Macros?

- Can the if-form be implemented as a function?

```
(if (some-condition-p)
    (print "true")
    (print "false"))
```

Useful Resources

- **The Art of the Metaobject Protocol** by Daniel G. Bobrow and Gregor Kiczales - one of the best OOP books out there
- **Let Over Lambda** by Doug Hoyte - great book on macro programming in Common Lisp

The End

Thank you!