Modern Lisp 36 lectures — function object and lambda

函数对象和闭包

```
(defun make-counter ()
  (let ((count 0))
    (lambda ()
      (incf count))))

(defvar *c1* (make-counter))
(defvar *c2* (make-counter))

(funcall *c1*) ; => 1
(funcall *c1*) ; => 2
(funcall *c2*) ; => 1
(funcall *c1*) ; => 3

(defun make-adder (n)
  (lambda (x) (+ x n)))

(defvar *add5* (make-adder 5))
(funcall *add5* 10) ; => 15
(funcall *add5* 20) ; => 25

(defvar *state* 0)
(defun step-state ()
  (let ((old *state*))
    (incf *state*)
    old))

(step-state) ; => 0
(step-state) ; => 1
(step-state) ; => 2
```

- 函数是一流对象：可赋值、传参、返回
- 闭包捕获的是变量本身，不是值的副本
- 每次调用 `make-counter` 创建一个新的计数器
- 闭包可以修改捕获的变量，这是状态的来源

宏：代码生成

```
(defmacr when (test &rest body)
  `(if ,test (progn ,@body) nil))

(defmacr unless (test &rest body)
  `(if (not ,test) (progn ,@body) nil))

(defmacr my-and (&rest forms)
  (cond ((null forms) t)
        ((null (cdr forms)) (car forms))
        (t `(if ,(car forms) (my-and ,@(cdr forms)) nil))))

(defmacr with-timing (&rest body)
  `(let ((start (get-internal-real-time))
         (result (progn ,@body))
         (elapsed (- (get-internal-real-time) start)))
     (format t "~a ms~%" elapsed)
     result))

(defmacr with-open-file-safe (spec &rest body)
  `(let ((stream nil)
         (result nil))
     (unwind-protect
         (progn
           (setf stream ,(car spec)
                 result (progn ,@body))
           result)
       (when stream (close stream)))))

(defmacr with-connection (conn &rest body)
  `(let ((*db-connection* ,conn))
     (unwind-protect
         (progn ,@body)
       (disconnect *db-connection*))))
```

- 宏在编译时展开，生成代码
- 反引号 `` ` `` 模板，`,` 插入表达式，`,@` 展开列表
- `unwind-protect` 保证清理代码执行
- 宏是 Lisp 区别于其他语言的核心能力

高阶函数

```
(defvar *users* '(alice bob carol dave))

(mapcar #'length *users*) ; => (5 3 4 4)
(mapcar #'(lambda (x) (* x x)) '(1 2 3 4 5)) ; => (1 4 9 16 25)

(defvar *nums* '(1 2 3 4 5 6 7 8 9 10))
(remove-if #'evenp *nums*) ; => (1 3 5 7 9)
(remove-if-not #'evenp *nums*) ; => (2 4 6 8 10)

(defvar *scores* '(85 92 78 96 88))
(reduce #'+ *scores*) ; => 439
(reduce #'max *scores*) ; => 96

(defvar *data* '(3 1 4 1 5 9 2 6))
(sort (copy-list *data*) #'<) ; => (1 1 2 3 4 5 6 9)
(find 5 *data*) ; => 5
(position 9 *data*) ; => 7
(count 1 *data*) ; => 2

(defvar *pairs* '((a 1) (b 2) (c 3)))
(assoc 'b *pairs*) ; => (B 2)
(rassoc 2 *pairs*) ; => (B 2)
```

- 函数作为参数：`#'name` 或 `#'(lambda ...)`
- `mapcar` 对每个元素应用函数，返回新列表
- `reduce` 折叠列表，`:initial-value` 指定初值
- `sort` 会修改原列表，用 `copy-list` 保护原数据
- `find` / `position` / `count` 接受 `:test` 参数，默认 `eql`

哈希表

```
(defvar *db* (make-hash-table :test #'equal))

(setf (gethash "alice" *db*) 30)
(setf (gethash "bob" *db*) 25)
(setf (gethash "carol" *db*) 28)

(gethash "alice" *db*) ; => 30, T
(gethash "dave" *db*) ; => NIL, NIL

(maphash #'(lambda (k v) (format t "~a: ~a~%" k v)) *db*)
; alice: 30
; bob: 25
; carol: 28

(defvar *counts* (make-hash-table :test #'eq))
(dolist (x '(a b a c b a))
  (incf (gethash x *counts* 0)))
(maphash #'(lambda (k v) (format t "~a: ~a~%" k v)) *counts*)
; A: 3
; B: 2
; C: 1
```

- `:test #'equal` 用字符串做键，`#'eq` 用符号做键
- `gethash` 返回 `(values 值 存在?)`，区分"不存在"和"存在但为 NIL"
- `incf (gethash k table 0)` 是计数器的惯用写法
- 哈希表顺序不保证，需要有序用 alist 或 sort 后遍历

结构体

```
(defstruct point
  (x 0 :type fixnum)
  (y 0 :type fixnum))

(defstruct (person (:constructor make-person)
                   (:print-object print-person))
  name
  age
  (email nil :read-only t))

(defstruct (employee (:extends person))
  salary
  department)

(defstruct (node (:type vector))
  value
  left
  right)

(defvar *p1* (make-point :x 3 :y 4))
(point-x *p1*) ; => 3
(point-y *p1*) ; => 4
(setf (point-x *p1*) 10)

(defvar *e1* (make-person :name "alice" :age 30 :email "a@b.c"))
(person-name *e1*) ; => "alice"
(person-email *e1*) ; => "a@b.c"
(setf (person-age *e1*) 31)

(defvar *n1* (make-node :value 5 :left nil :right nil))
(node-value *n1*) ; => 5
```

- `:type fixnum` 编译器会生成更快的访问代码
- `:read-only t` 防止修改，适合不变数据
- `:extends` 实现单继承，子类继承父类所有槽
- `:type vector` 用向量存储，内存紧凑，适合大量实例
- `:print-object` 自定义打印格式，调试时很有用

错误处理

```
(defcondition my-error (error)
  (:report (lambda (c s) (format s "my error occurred"))))

(defcondition file-not-found (error)
  (:report (lambda (c s) (format s "~a not found" (pathname c)))))

(defcondition parse-error (error)
  (:report (lambda (c s) (format s "parse error at ~a" (position-error c)))))

(defcondition connection-lost (error)
  (:report (lambda (c s) (format s "connection lost"))))

(defcondition warning-deprecated (warning)
  (:report (lambda (c s) (format s "deprecated: ~a" (deprecated-name c)))))

(defcondition user-interrupt (error)
  (:report (lambda (c s) (format s "user interrupted"))))

(defcondition retry-exhausted (error)
  (:report (lambda (c s) (format s "retry exhausted"))))

(defcondition validation-error (error)
  (:report (lambda (c s) (format s "validation error"))))

(defcondition timeout (error)
  (:report (lambda (c s) (format s "timeout"))))

(defcondition resource-busy (error)
  (:report (lambda (c s) (format s "resource busy"))))

(defcondition permission-denied (error)
  (:report (lambda (c s) (format s "permission denied"))))

(defcondition unexpected-eof (error)
  (:report (lambda (c s) (format s "unexpected eof"))))

(defcondition encoding-error (error)
  (:report (lambda (c s) (format s "encoding error"))))

(defcondition overflow (error)
  (:report (lambda (c s) (format s "overflow"))))

(defcondition underflow (error)
  (:report (lambda (c s) (format s "underflow"))))

(defcondition syntax-error (error)
  (:report (lambda (c s) (format s "syntax error"))))

(defcondition type-error (error)
  (:report (lambda (c s) (format s "type error"))))

(defcondition unbound-variable (error)
  (:report (lambda (c s) (format s "unbound variable"))))

(defcondition stack-overflow (error)
  (:report (lambda (c s) (format s "stack overflow"))))

(defcondition heap-exhausted (error)
  (:report (lambda (c s) (format s "heap exhausted"))))

(defcondition deadlock (error)
  (:report (lambda (c s) (format s "deadlock"))))

(defcondition starvation (error)
  (:report (lambda (c s) (format s "starvation"))))

(defcondition priority-inversion (error)
  (:report (lambda (c s) (format s "priority inversion"))))

(defcondition checksum-mismatch (error)
  (:report (lambda (c s) (format s "checksum mismatch"))))

(defcondition corrupt-data (error)
  (:report (lambda (c s) (format s "corrupt data"))))

(defcondition version-mismatch (error)
  (:report (lambda (c s) (format s "version mismatch"))))

(defcondition protocol-error (error)
  (:report (lambda (c s) (format s "protocol error"))))

(defcondition unsupported (error)
  (:report (lambda (c s) (format s "unsupported"))))

(defcondition not-implemented (error)
  (:report (lambda (c s) (format s "not implemented"))))

(defcondition internal-error (error)
  (:report (lambda (c s) (format s "internal error"))))

(defcondition fatal (error)
  (:report (lambda (c s) (format s "fatal"))))

(defcondition recoverable (error)
  (:report (lambda (c s) (format s "recoverable"))))

(defcondition non-recoverable (error)
  (:report (lambda (c s) (format s "non-recoverable"))))

(defcondition warning-1 (warning)
  (:report (lambda (c s) (format s "warning 1"))))

(defcondition warning-2 (warning)
  (:report (lambda (c s) (format s "warning 2"))))

(defcondition warning-3 (warning)
  (:report (lambda (c s) (format s "warning 3"))))

(defcondition
```