MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Recitation – Wednesday, April 7

1. Student Presentation

Explain to the class, using box and pointer diagrams, what the following procedure does:

```
(define (make-ring x)
  (set-cdr! (last x) x)
  "The dirty deed is done!")
```

when applied to the list letters `(define letters (list 'a 'b 'c))`. (Last is a procedure that returns the last element of a list). An interesting way to approach this explanation is to write the two procedures and the list on the board, and then ask the class what happens when you type `(make-ring letters)`.

2. Review of Environment Diagrams

Define a one-parameter procedure `prev` that returns as its value the argument that was passed to it the previous time it was called. The first time `prev` is called it should return `nil`.

```
(prev 'a)       ==>  ()
(prev 3)        ==>  a
(prev (+ 1 5))  ==>  3
```

```
(define prev
  (let ((lastval '()))
    (lambda (x)
      (let ((return lastval))
        (set! lastval x)
        return))))
```

Now define a procedure `make-prev` that can be used to create the procedure `prev`. `Make-prev` should take as an argument the value that should be returned by the first call to `prev`. Then draw the environment diagram for the following expressions:

```
(define make-prev
  (lambda (lastval)
    (lambda (x)
      (let ((return lastval))
        (set! lastval x)
        return))))

(define prev (make-prev 0))
(prev 1) ==> 0
(prev 2) ==> 1
```

Finally, write the procedure `make-n-prev` that causes a "delay" of `n` where the first `n` calls to the procedure return `val`: (Hint: `compose` calls to `make-prev`)

```
(define prev-2 (make-n-prev 2 0))
(prev-3 1)      ==> 0
(prev-3 2)      ==> 0
(prev-3 3)      ==> 1
```

```
(define (make-n-prev n val)
  (if (= n 0) identity
      (compose (make-prev val)
               (make-n-prev (- n 1) val))))
```

3. Review of Object Oriented Programming

Below is the object oriented system from the March 19th Lecture Notes (just included for reference):

```
(define (get-method object message)
  (object message))

(define (no-method name)
  (list 'no-method name))

(define (no-method? x)
  (if (pair? x)
      (eq? (car x) 'no-method)
      false))

(define (method? x)
  (not (no-method? x)))

(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message (cadr method)))))
```

4. Object Oriented Stacks

Using this object oriented style, write the function `create-stack` that will create a stack object. Recall that stacks are data structures that include the operations `push`, `pop`, `peek`, and `clear`. Objects get `push`ed on and `pop`ped off the stack in a last-in-first-out manner. For example:

```
(define s (create-stack))
(ask s 'push 5)
(ask s 'push 3)
(ask s 'pop)           ==> 3
(ask s 'push 1)
(ask s 'pop)           ==> 1
(ask s 'pop)           ==> 5
```

Complete the function `create-stack`, below:

```
(define (create-stack)
  (let ((value '()))
    (lambda (message)
      (case message
        ((PUSH)
         (lambda (self new)
           (set! value (cons new value))))
        ((PEEK)
         (lambda (self) (car value)))
        ((CLEAR)
         (lambda (self) (set! value '())))
        ((POP)
         (lambda (self)
           (let ((return (ask self 'peek)))
             (set! value (cdr value))
             return)))))))
```

5. Object Oriented Variables

Next, let's write an abstraction for variables in the object oriented style. We do this so that in addition to the `get` (lookup) and `set!` operations that we have in scheme, we also want to implement an `undo!` method that un-does the last `set!`ing of the variable. We want to store an arbitrary number of undoes. For example:

```
(define a (create-var 1))
(ask a 'get)                ==> 1
(ask a 'set! 2)
(ask a 'get)                ==> 2
(ask a 'undo!)
(ask a 'get)                ==> 1
```

How can we use stacks to help us with this `undo!`? Complete the code for `create-var`, below:

```
(define (create-var value)
  (define undo (create-stack))
  (ask undo 'push value)
  (lambda (message)
    (case message
      ((SET!)
       (lambda (self new)
         (ask undo 'push new)))
      ((GET)
        (lambda (self)
          (ask undo 'peek)))
      ((UNDO!)
       (lambda (self)
         (ask undo 'pop))))))
```

6. Object Oriented Pairs

Well, we wrote it for one variable, now let's write it for pairs. Consider the function `create-pair` that generates a pair of variables (the car and the cdr) that you can get or set. When the message `undo!` is sent, the last setting gets undone (be it a `set-car!` or `set-cdr!`. Consider the following example:

```
(define p (create-pair 1 2))
(ask p 'print)              ==> (1 . 2)
(ask p 'set-car! 3)
(ask p 'print)              ==> (3 . 2)
(ask p 'set-cdr! 4)
(ask p 'print)              ==> (3 . 4)
(ask p 'set-car! 5)
(ask p 'print)              ==> (5 . 4)
(ask p 'undo!)
(ask p 'print)              ==> (3 . 4)
(ask p 'undo!)
(ask p 'print)              ==> (3 . 2)
(ask p 'undo!)
(ask p 'print)              ==> (1 . 2)
```

Complete the code for `create-pair`, below:

```
(define (create-pair car-val cdr-val)
  (define last-change (create-var '()))
  (define the-car (create-var car-val))
  (define the-cdr (create-var cdr-val))
;; Continued on next page
```

```
    (lambda (message)
      (case message
        ((SET-CAR!)
         (lambda (self new)
           (ask last-change 'set! 'car)
           (ask the-car 'set! new)))
        ((SET-CDR!)
         (lambda (self new)
           (ask last-change 'set! 'cdr)
           (ask the-cdr 'set! new)))
        ((CAR)
         (lambda (self) (ask the-car 'get)))
        ((CDR)
         (lambda (self) (ask the-cdr 'get)))
        ((PRINT)
         (lambda (self)
           (newline)
           (display (cons (ask self 'car) (ask self 'cdr)))
           (newline)))
        ((UNDO!)
         (lambda (self)
           (case (ask last-change 'undo!)
             ((CAR) (ask the-car 'undo!))
             ((CDR) (ask the-cdr 'undo!))))))))
```

7. Object Oriented Variables with Redo!

Now we have the ability to `undo!` the results to set, let's add a `redo!` feature, such that successive calls to `undo!` and be taken back using a call to `redo!`. For example,

```
(define a (create-var 1))
(ask a 'set! 2)
(ask a 'undo!)
(ask a 'get)    ==> 2
(ask a 'undo!)
(ask a 'get)    ==> 1
(ask a 'redo!)
(ask a 'get)    ==> 2
```

Complete the code for `create-var`, below:

```
(define (create-var value)
  (define undo (create-stack))
  (define redo (create-stack))
  (ask undo 'push value)
  (lambda (message)
    (case message
      ((SET!)
       (lambda (self new)
         (ask redo 'clear)
         (ask undo 'push new)))
      ((GET)
        (lambda (self)
          (ask undo 'peek)))
      ((UNDO!)
       (lambda (self)
         (ask redo 'push (ask undo 'pop))))
      ((REDO!)
       (lambda (self)
         (ask undo 'push (ask redo 'pop)))))))
```