MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Friday, April 2

1. Object Oriented Programming

Below is the object oriented system from the April 1st Lecture Notes (just included for reference):

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

(define method? procedure?)
```

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

2. 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. Complete the function `create-stack`.

```
(define (create-stack)

    (lambda (message)
      (case message
        ((PUSH)

                                          )
        ((PEEK)
                                     )
        ((CLEAR)
                                          )
        ((POP)



                          )))))
```

```
(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
```

3. 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 undos. How can we use stacks to help us with this `undo!`? Complete the code for `create-var`.

```
(define (create-var value)


    (lambda (message)
      (case message
        ((SET!)

                                 )
        ((GET)

                              )
        ((UNDO!)

                            )))))
```

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

4. 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 and complete the code for `create-pair`.

```
(define (create-pair car-val cdr-val)



   (lambda (message)
    (case message
      ((SET-CAR!)



                                   )
      ((SET-CDR!)



                                   )
      ((CAR)
                                           )
      ((CDR)
                                           )
      ((PRINT)
       (lambda (self)
         (display (ask self 'car))
         (display "."
         (display (ask self 'cdr))))
      ((UNDO!)




                                            )))))
```

```
(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)
```

5. 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!`. Complete the code for `create-var`.

```
(define (create-var value)



   (lambda (message)
    (case message
      ((SET!)



                                )
      ((GET)


                             )
      ((UNDO!)


                                            )
      ((REDO!)


                                            )))))
```

```
(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
```