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

Recitation – Wednesday, April 21

1. Memoization and Streams

Memoization is a technique to improve performance by recording previously computed values. Whenever a computation is repeated, the recorded result is used instead of performing the same work a second time. Today, we will look at a simple example of how to memoize a procedure:

```
(define (memoize proc)
  (let ((run? #f) (result #f))
    (lambda ()
      (cond ((not run?)
             (set! result (proc))
             (set! run? #t)
             result)
            (else result)))))
```

Consider now what happens if we change the definition of `delay` as follows.
`(delay s) <==> (memoize (lambda () s))`
Draw the environment diagrams for the following expressions with and without stream memoization.

```
(define a 5)
(define flip (cons-stream a (cons-stream (- a) flip)))
(stream-car (stream-cdr a))
(set! a 7)
(stream-car (stream-cdr a))
```

2. Playing with Scope

Let's take a look at our old friend `make-acct`:

```
(define make-acct
  (lambda (balance)
    (lambda (msg . args)
      (case msg
        ((BALANCE) balance)
        ((DEPOSIT) (set! balance (+ balance (cadr args))))))))

(define a (make-acct 100))
(a 'balance)           ==> 100

(define b
  (let ((balance 1000))
    a))
(b 'balance)           ==> 100
```

Why does `b` have a balance of 100 and not 1000? Quickly sketch the environment diagram for the above expressions to see why.

3. Re-scoping a procedure

Consider a new special form called `rescope` that takes a procedure and returns a procedure with the same parameters and body, but whose environment pointer points to the current environment. Consider defining another account `c` by rescoping `a` (instead of by calling `make-acct`).

```
(define c
  (let ((balance 1000))
    (rescope a)))
(c 'balance)            ==> 1000
```

Let's add the special form `rescope` to the mc-evalutator.

(1) Define Data Abstraction

(2) Add the appropriate `cond` clause to `mc-eval`

(3) Write `eval-rescope`

```
(define (eval-rescope exp env)







)
```

4. Grabbing procedure's scope

Consider a new special form called `inscope` that takes a procedure and an expression and then evaluates the expression with respect to the procedure's environment. For example

```
(define d
  (inscope a (lambda () (set! balance 'loser))))

(a 'balance)            ==> 100
(d)
(a 'balance)            ==> loser
```

Let's add the special form `inscope` to the mc-evalutator.

(1) Define Data Abstraction

(2) Add the appropriate `cond` clause to `mc-eval`

(3) Write `eval-inscope`

```
(define (eval-inscope exp env)







)
```