CS 61A, Summer 2006 — July 24, 2006

LAB ASSIGNMENT 5A

1. Given below is a simplified version of the `make-account` procedure on page 223 of Abelson and Sussman.

```
(define (make-account balance)
  (define (withdraw amount)
    (set! balance (- balance amount)) balance)
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (dispatch msg)
    (cond
      ((eq? msg 'withdraw) withdraw)
      ((eq? msg 'deposit) deposit) ) )
  dispatch)
```

Fill in the blank in the following code so that the result works exactly the same as the `make-account` procedure above, that is, responds to the same messages and produces the same return values. The differences between the two procedures are that the inside of `make-account` above is enclosed in the `let` below, and the names of the parameter to `make-account` are different.

```
(define (make-account init-amount)
  (let ( *FILL IN HERE*                       )
    (define (withdraw amount)
      (set! balance (- balance amount)) balance)
    (define (deposit amount)
      (set! balance (+ balance amount)) balance)
    (define (dispatch msg)
      (cond
        ((eq? msg 'withdraw) withdraw)
        ((eq? msg 'deposit) deposit) ) )
    dispatch) )
```

2. Modify either version of `make-account` so that, given the message `balance`, it returns the current account balance, and given the message `init-balance`, it returns the amount with which the account was initially created. For example:

```
> (define acc (make-account 100)
acc
> (acc 'balance)
100
```

Continued on next page...

Continued:

3. Modify `make-account` so that, given the message `transactions`, it returns a list of all transactions made since the account was opened. For example:

```
> (define acc (make-account 100))
acc
> ((acc 'withdraw) 50)
50
> ((acc 'deposit) 10)
60
> (acc 'transactions)
((withdraw 50) (deposit 10))
```

4. Given this definition:

```
(define (plus1 var)
  (set! var (+ var 1))
  var)
```

Show the result of computing

```
(plus1 5)
```

using the substitution model. That is, show the expression that results from substituting `5` for `var` in the body of `plus1`, and then compute the value of the resulting expression. What is the actual result from Scheme?

Continued on next page...

This lab activity consists of example programs for you to run in Scheme. Predict the result before you try each example. If you don't understand what Scheme actually does, ask for help!

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(make-adder 3)

((make-adder 3) 5)

(define (f x) (make-adder 3))

(f 5)

(define g (make-adder 3))

(g 5)

(define (make-funny-adder n)
  (lambda (x)
    (if (equal? x 'new)
        (set! n (+ n 1))
        (+ x n))))

(define h (make-funny-adder 3))

(define j (make-funny-adder 7))

(h 5)

(h 5)

(h 'new)

(h 5)

(j 5)

(let ((a 3))
  (+ 5 a))

(let ((a 3))
  (lambda (x) (+ x a)))

((let ((a 3))
   (lambda (x) (+ x a)))
 5)

((lambda (x)
   (let ((a 3))
     (+ x a)))
 5)

(define k
  (let ((a 3))
    (lambda (x) (+ x a))))

(k 5)

(define m
  (lambda (x)
    (let ((a 3))
      (+ x a))))

(m 5)

(define p
  (let ((a 3))
    (lambda (x)
      (if (equal? x 'new)
          (set! a (+ a 1))
          (+ x a)))))

(p 5)

(p 5)

(p 'new)

(p 5)

(define r
  (lambda (x)
    (let ((a 3))
      (if (equal? x 'new)
          (set! a (+ a 1))
          (+ x a)))))

(r 5)

(r 5)

(r 'new)

(r 5)
```

Continued on next page...

Continued:

```
(define s
  (let ((a 3))
    (lambda (msg)
      (cond ((equal? msg 'new)
             (lambda ()
               (set! a (+ a 1))))
            ((equal? msg 'add)
             (lambda (x) (+ x a)))
            (else (error "huh?"))))))

(s 'add)

(s 'add 5)

((s 'add) 5)

(s 'new)

((s 'add) 5)

((s 'new))

((s 'add) 5)
```

```
(define (ask obj msg . args)
  (apply (obj msg) args)))

(ask s 'add 5)

(ask s 'new)

(ask s 'add 5)

(define x 5)

(let ((x 10)
      (f (lambda (y) (+ x y))))
  (f 7))

(define x 5)
```