CS61A FALL 2008 — MIDTERM 3 REVIEW

by Min Xu, Chongyang Wang, and Theron Ji[1]

Special thanks to David Zeng and the HKN tutoring committee.

1. Environmental Diagrams

Exercise 1. Draw the environmental diagrams and show the return values:

```
(define x 3)

(define baz
  (let ((x 100)
        (f (lambda (y) (+ x y)))
      (lambda (x) (f (* x x)))))

(baz 7)

Answer: 52
```

Note: for environment diagrams, it's best to step through the process in EnvDraw.

```
(define x 'old-x)
(define y 'old-y)

((lambda () (define y 6)
            (set! x 2)))

x
y

Answer: x is 2, y is 'old-y
```

Date: November 10, 2008.

[1]based on CS 61A review sheet from Summer 2008

2. OOP Below the Line

Exercise 2. Fill in the blanks with the response to the indicated expressions. The answers are 11, 121, 1001, 1111 but not necessarily in that order!

(HINT: You shouldn't have to draw environmental diagrams to figure this out. In each case, ask yourself: is `a` a class variable or an instance variable? Is `b` a class variable or an instance variable?)

```
(define make-foo1
  (let ((a 1))
     (lambda ()
        (let ((b 1))
          (lambda ()
             (set! a (* a 10))
             (set! b (+ b a))
              b )))))
(define foo1-1 (make-foo1))
(define foo1-2 (make-foo1))
(foo1-1)
(foo1-1)
(foo1-2) ==> ?

Answer: 1001


(define make-foo2
  (let ((a 1)
        (b 1))
       (lambda ()
           (lambda ()
              (set! a (* a 10))
              (set! b (+ b a))
              b))))
(define foo2-1 (make-foo2))
(define foo2-2 (make-foo2))
(foo2-1)
(foo2-1)
(foo2-2) ==> ?

Answer: 1111
```

```
(define make-foo3
  (let ((b 1)
        (lambda ()
          (let ((a 1))
            (lambda ()
               (set! a (* a 10))
               (set! b (+ b a))
              b)))))
(define foo3-1 (make-foo3))
(define foo3-2 (make-foo3))
(foo3-1)
(foo3-1)
(foo3-2) ==> ?
```

Answer: 121

```
(define make-foo4
   (lambda ()
     (let ((a 1)
           (b 1))
         (lambda ()
            (set! a (* a 10))
            (set! b (+ b a))
            b))))
(define foo4-1 (make-foo4))
(define foo4-2 (make-foo4))
(foo4-1)
(foo4-1)
(foo4-2) ==> ?
```

Answer: 11

3. List Mutation

Exercise 3.

```
(define f (cons 1 2))
(define g (list f f))
(set! f 3)
(set-car! (car g) 4)
g
```

```
Answer: ((4 . 2) (4 . 2))
```

```
(define f (list 2 3))
(define g (append f f))
(set-car! g f)
g
```

```
Answer: ((2 3) 3 2 3)
```

```
(let ((x (list 1 2 3)))
     (set-car! x (list 'a 'b 'c))
     (set-car! (cdar x) 'd)
     x)
```

```
Answer: ((a d c) 2 3)
```

Exercise 4. Write a procedure `link-first!` that takes a non-empty list of numbers and destructively links together all its elements with the same value as the car of the list. For example:

```
STk> (define ls '(1 2 3 2 1 1 2 1))
STk> (link-first! ls)
STk> ls
  (1 1 1 1)
```

DO NOT create any additional pairs!

```
(define (link-first! ls)
   (cond ((null? (cdr ls)) '())
         ((= (car ls) (cadr ls)) (link-first! (cdr ls)))
         (else (set-cdr! ls (cddr ls))
               (link-first! ls))))
```

4. Vectors

Exercise 5. Write a procedure `subvector` that takes three arguments: a vector `vec` and two non-negative integers `start` and `end` that are less than the length of `vec`. It should return a new vector containing only the elements of `vec` at positions between start and end inclusive. For example:

```
STk> (subvector #(we all live in a yellow submarine) 2 5)
  #(live in a yellow)
```

If `end` is less than `start`, return an empty vector.

```
(define (subvector vec start end)
   (let ((new-vec (make-vector (+ (- end start) 1))))
      (define (loop index)
         (if (= index (vector-length new-vec))
             new-vec
             (begin (vector-set! new-vec index (vector-ref vec (+ index start)))
                    (loop (+ index 1)))))
      (loop 0)))
```

5. Streams

Exercise 6. What are the first 10 elements of the following stream?

```
(define s
      (cons-stream 1
            (interleave (stream-cdr integers) s)))

Answer: (1 2 1 3 2 4 1 5 3 6 ...)
```

Exercise 7. Implicitly define the following stream without defining any procedures. You may use the stream `ones` and `integers`, and common stream operators.

```
(     ()    (())   ((()))   (((())))  ((((()))))    ((((((()))))) ...        )

(define s1 (cons-stream '()
                        (stream-map list s1)))

(  0  3  8  15  24  35  48  63  80  99  ...  )

(define s2 (stream-map - (stream-map * integers
                                      integers)
                         ones))
```

6. Concurrency

Exercise 8. Analyze the following code and write down all possible answers:

```
(define x 3)
(parallel-execute (lambda () (set! x (+ (* x x) 1)))
                  (lambda () (let ((y x))
                                   (set! x (+ y y))))

Answer: 20, 37, 10, 6, 19
```