Quiz 8b

1. (4 points)

a. (2 points) Draw the environment diagram and find what the last statement evaluates to.

(define (foo f x)

(lambda (y) (f x y)))

(define fisher (lambda (y x) (/ y x)))

(define bar (foo fisher 4))

> (bar 8)

\_\_.5\_\_\_

**1. Start with G and have the foo bubble pair pointing into foo inside G and foo pointing to the bubble pair**

**2. Then have a fisher variable in G whose value is a lambda. This lambda points to G and fisher points to the bubble pair representing that lambda**

**3. Create an E1 that points to G where f points to fisher’s bubble pair and x = 4. Bar should be in G and point to a lambda bubble pair that points to E1**

**4. Create E2 that points to E1 with y = 8**

**5. Create E3 that points to G with y = 4 and x = 8**

**returns 0.5**

**.5 points for right answer**

**1.5 points for env diagram**

b. (2 points) Draw the environment diagram and find what the last statement evaluates to.

> (define (foo)

(define y (cons 1 2))

(lambda (y z) (set! x (cons y z))))

> (define x 1)

> ((foo) x 3)

> x

\_\_(1 . 3)\_\_

**1. G has foo pointing to its bubble pair and that pair is pointing back to G**

**2. G has x with value 1**

**3. E1 points to G and has y equal to (cons 1 2) and has a bubble pair with the lambda body pointing to E1**

**4. E2 points to E1 and has y = 1 and z = 3**

**5. x is now set to (1 . 3)**

**x returns (1 . 3)**

**0.5 points for right answer**

**1.5 points for env diagram**

1. (2 points) Write a procedure mr-worm that returns a “worm” procedure. Whenever this “worm” procedure is called, it appends a 1 to a list inside the procedure. This list is initially 1 element long. Whenever a new mr-worm procedure is called another worm with the same length as the most recent worm is created.

>(define worm1 (mr-worm))

worm1 ;right now worm1 is (1)

>(worm1)

(1 1)

>(worm1)

(1 1 1)

>(define worm2 (mr-worm))

worm2 ;right now worm2 is (1 1 1)

>(worm2)

(1 1 1 1)

>(define worm3 (mr-worm))

worm3 ;right now worm3 is (1 1 1 1)

**(define mr-worm**

**(let ((last-length (list 1)))**

**(lambda ()**

**(let ((my-length last-length))**

**(lambda () (set! my-length (cons 1 my-length))**

**(set! last-length my-length)**

**my-length)))))**

1. (4 points) To get a better understanding of how classes work under the line, we will write the equivalent of a class **without using define-class**.

We give you a book class. You can assume it works like a normal class. It has two methods title and contents which returns the title and contents respectively.

>(define book1 (instantiate book ‘(CS61as is awesome) ‘(A very short book) ))

book1

>(ask book1 ‘title)

(CS61as is awesome)

>(ask book1 ‘contents)

(A very short book)

Now write code that will simulate a shelve class with a list of books. It should be able to add books when given a book and remove a book given its title. It should output false when finding or removing a book the shelf doesn’t have. Assume there will be no books with the same title. Remember write this in below the line. **You can assume book works above the line though.**

>(define shelf1 (make-shelf)

shelf1

>((shelf1 ‘find) ‘(The Meaning Of Life))

#f

>((shelf1 ‘add) book1)

okay

>(ask ((shelf1 ‘find) ‘(CS61as is awesome)) ‘contents)

(A very short book)

Here is the above line version:

(define-class (shelf)

(instance-vars (books nil))

(method (add-book title) #code to add a book to books#)

(method (find-book title) #code to find a book in books#))

**(define (make-shelf)**

**(let ((books ‘()))**

**(define (add-book book)**

**(set! books (cons book books)))**

**(define (find-book title)**

**(define (helper title finding)**

**(cond ((empty? finding) #f)**

**((eq? title (ask (car finding) ‘title)) (car lst))**

**(else (helper title (cdr finding)))))**

**(helper title books)))**

**(define (dispatch m)**

**(cond ((eq? m ‘find) find-book)**

**((eq? m ‘add) add-book)**

**(else (error “unrecognizable command”))))**

**dispatch))**

**3 points: should probably grade leniently given how long the problem is, but they should have. If they did a lambda version that’s fine too, and they do not need an error check.**

* **All functions inside the first define**
* **Semi functional code in each “class command”, including set! to change the books**
* **Some sort of dispatch function for message passing**
* **A books “instance variable”**

**ignore whatever is below:**

**(define (remove-book title)**

**(define (helper title finding parsed)**

**(cond ((empty? finding) #f)**

**((eq? title (ask (car finding) ‘title))**

**(set! books (cons parsed (cdr finding)))**

**(else**

**(helper title (cdr finding) (cons (car finding) parsed))))**

**(helper title books ‘()))**