**Quiz 9B**

Name:

Login: cs61as-\_\_\_

Date:

1) (3 points) Say what Scheme prints, **and** draw a box-and-pointer diagram. In the case of an error, say “error” and don’t draw the box-and-pointer diagram. If there is a cycle, which would make it print infinitely, write “cycle” (and still draw the box-and-pointer diagram).

> (define x (list (cons 1 2) (cons 3 4)))

> (set-car! x (cadr x))

> (set-cdr! (cdr x) (car x))

> x

\_((3 . 4) (3 . 4) 3 . 4)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

> (let ((x (list 'dogs 'eat 'homework))

(y (list 'let 'dogs 'out)))

(set-cdr! (cdr y) (cdr x))

(set-cdr! x y)

x)

(dogs let dogs eat homework)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

> (define x (cons 1 2))

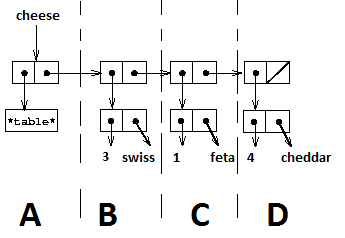
> (define y (cons 3 4))

> (set-cdr! x y)

> (set-car! (cdr y) x)

x

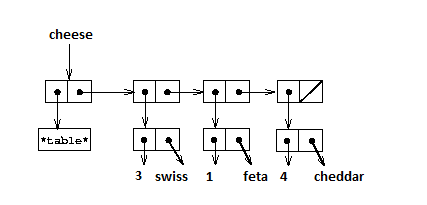
#error\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



2) (4 points total) Tom needs help! Jerry asked him to write a procedure *flatten!* that takes in a 1 dimensional table and mutate it so that it returns a list of values and keys (i.e. no nested lists). **Don’t create any new pairs. Only change their pointers.**

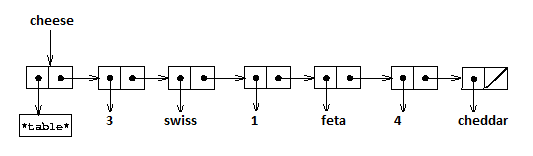
>cheese

(\*table\* (3 . swiss) (1 . feta) (4 .cheddar))



>(flatten! cheese)

(\*table\* 3 swiss 1 feta 4 cheddar)



a) (1 point)

Tom notices that except for the first chunk, marked A in this diagram, everything else (B,C,D, etc) have the same structure. If he knows how to swap the elements in B, he can do the same algorithm on thte rest.

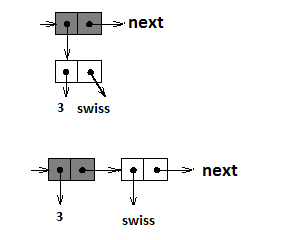
Tom wants to use recursion to make his job easier. What should be his base case? What should he return when he reach the base-case?

**1 point : null? return nil**

**0.5 point: just null? OR just nil OR (null? (cdr x)) return “do the operation”**

b) (1 point)

Tom now focuses on just B



He wants to write a function, flatten-helper that accepts a pair with a structure like B, and change it into something like the bottom structure (the boxes are shaded so that you know which one is which)

His attempt is as follows (along with his comments):

(define (flatten-helper lst)

(if <base case condition>

<base case return value>

(let ((next (cdr lst))

(set-car! lst (caar lst)) ;set car of gray box to 3

(set-cdr! lst (car lst)) ;set cdr of gray box to white box

(set-car! (car lst) (cdar lst)) ;set car of white box to ‘swiss

(flatten-helper next)))) ;Do the same thing with the rest

What is wrong with his code? (You can explain in words or draw a box-pointer on what will happen if we call flatten-helper on B)

**Any statement about losing the reference to the original white-pair, or referring to the wrong element. box-pointer is fine too**

**1 point: correct**

**0.5 point:** **Either vague explanation, or wrong box-pointer**

c) (2 point) Tom gave up. He asks you to write flatten! Feel free to fix Tom’s helper function and use it. If you don’t like Tom’s helper function and want to write your own, feel free to do so.

\*Hint: After you finish your code, try calling it with the example above and update the pointers as you go. It’s a good way to debug.

**(define (flatten! lst)**

**(define (helper lst)**

**(if (null? lst)**

**lst**

**(let ((key-value (car lst))**

**(next (cdr lst)))**

**(set-car! lst (car key-value))**

**(set-cdr! lst key-value)**

**(set-car! key-value (cdr key-value))**

**(set-cdr! key-value (helper next))**

**lst)))**

**(set-cdr! lst (helper (cdr lst)))**

**lst)**

**– 0.5 points if they need a return value (like returning the structure) for the recursion to work**

**–0.5 points for creating new pairs, but still using mutation in some way**

**- 0.5 for not keeping a pointer to the key-value pair (if flatten! Still works flawlessly, exempt from this deduction)**

**- 0.5 point if solution is correct, but the first pair being \*table\* messes it**

3) (3 points) Heath Ledger wants to create a function, *pairwise-vect* that accepts 2 vectors as arguments and return a vector that contains all possible pairing between their elements.

>(define ranks (vector 1 2 3 4))

#(1 2 3 4)

>(define suits (vector ‘a ‘b ‘c))

#(a b c)

>(pairwise-vect ranks suits)

#(1a 2a 3a 4a 1b 2b 3b 4b 1c 2c 3c 4c)

**\*Don’t use list->vector or vector->list**

**\*The order in which they appear doesn’t matter. So, this (and other permutations of it) is equally valid:**

>(pairwise-vect ranks suits)  
#(1a 1b 1c 2a 2b 2c 3a 3b 3c 4a 4b 4c)

**(define (pairwise-vect suits ranks)**

**(define (loop newvec i s r)**

**(if (= i (vector-length newvec))**

**newvec**

**(let ((next-s (if (= (+ 1 s) (vector-length suits)) 0 (+ s 1)))**

**(next-r (if (= (+ 1 s) (vector-length suits)) (+ r 1) r)))**

**(vector-set! newvec i (word (vector-ref suits s) (vector-ref ranks r)))**

**(loop newvec (+ i 1) next-s next-r))))**

**(loop (make-vector (\* (vector-length suits) (vector-length ranks))) 0 0 0))**

**-0.5 for each off by 1 error (usually in base case or list-ref)**

**-0.5 if syntax error for not using let or begin in an if**

**- 1 point if doesn’t return a new vector of the right length**

**- 1 point if returned vector traverses only 1 of the list completely, and returns a vector with duplicates/empty entries**