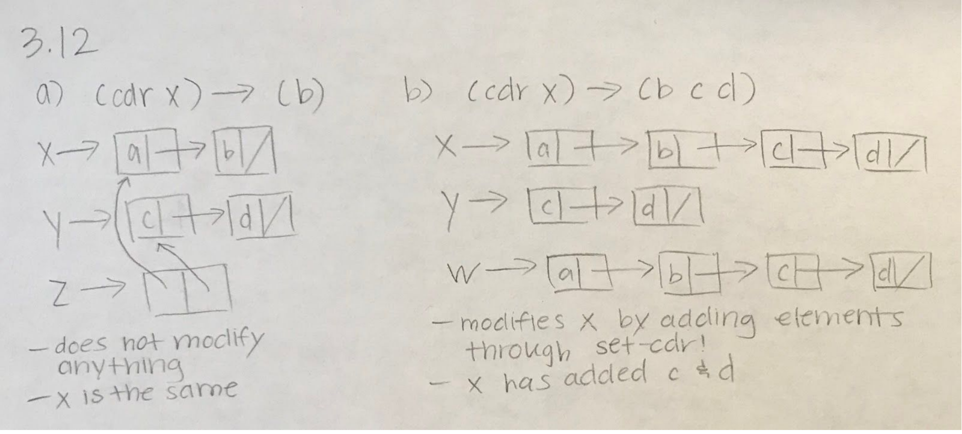
**Hannah Zhang**
**3.15-3.22**

Before Test 3

**3.12**



3.12

a) (cdr x) → (b)

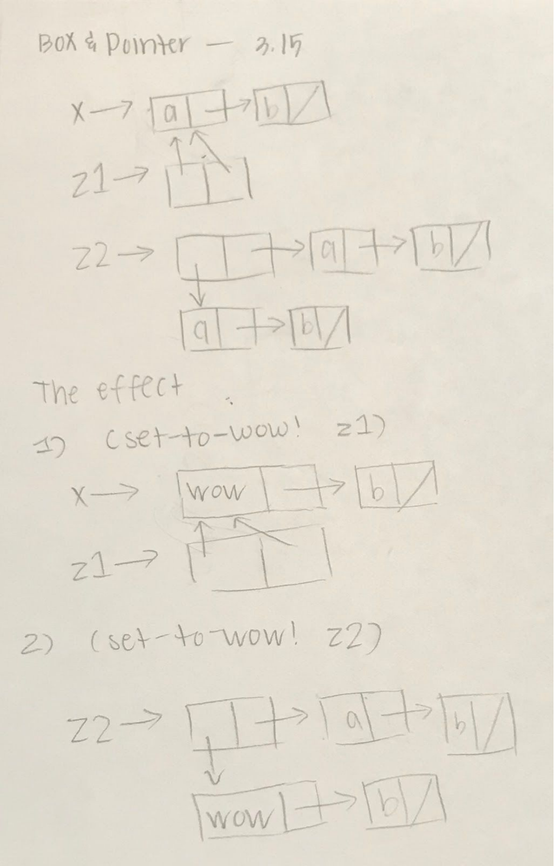
- does not modify anything
- x is the same

b) (cdr x) → (b c d)

- modifies x by adding elements through set-cdr!
- x has added c & d

**3.15**

Draw box-and-pointer diagrams to explain the effect of set-to-wow! on the structures z1 and z2 above.



Box & Pointer — 3.15

The effect:

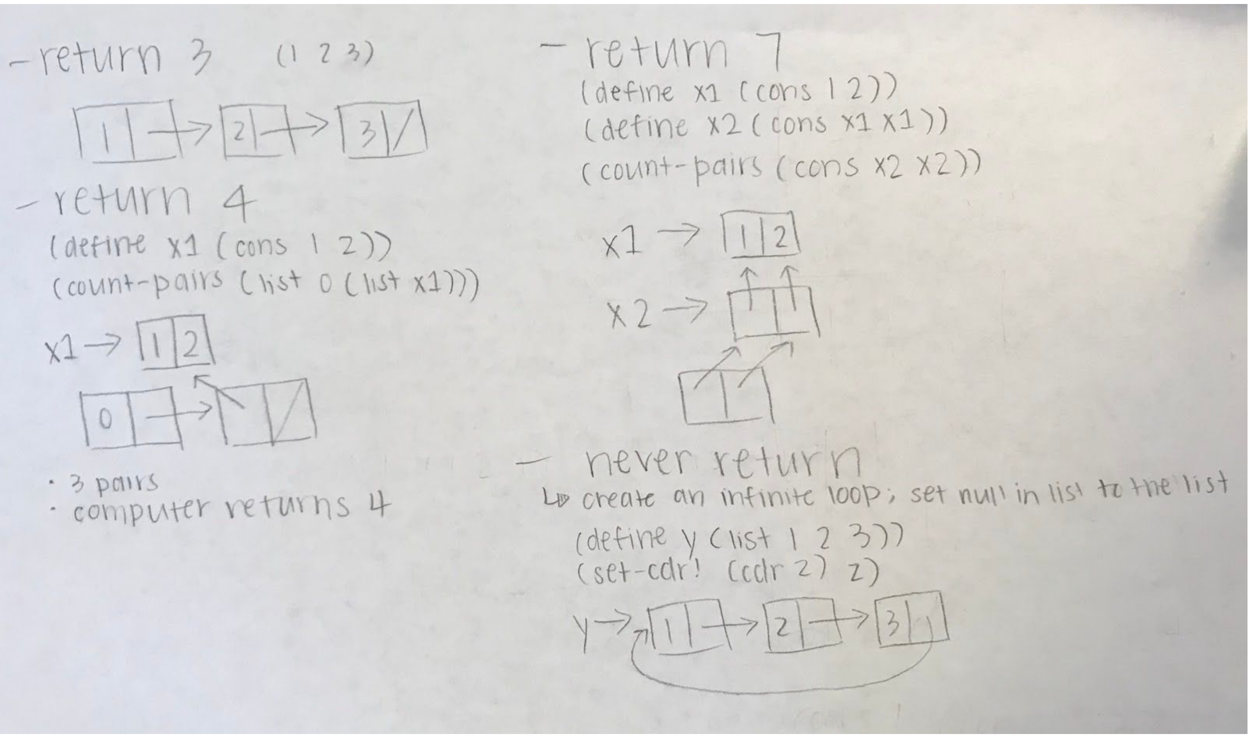
1) (set-to-wow! z1)

2) (set-to-wow! z2)

**3.16**

In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben's procedure would return 3; return 4; return 7; never return at all.

; the key to this problem is to create a variable that refers to a place in memory and then call this variable. This program counts it again when in reality, it is not counted again.



- return 3   (1 2 3)

- return 4

```
(define x1 (cons 1 2))
(count-pairs (list 0 (list x1)))
```

· 3 pairs
· computer returns 4

- return 7

```
(define x1 (cons 1 2))
(define x2 (cons x1 x1))
(count-pairs (cons x2 x2))
```

- never return

↳ create an infinite loop; set null in list to the list

```
(define y (list 1 2 3))
(set-cdr! (cddr z) z)
```

; sharing is undetectable if we operate on lists using only cons, car, and cdr

**; note that for return four, I drew it wrong**

**Should be (cons 0 x2)**

**3.22**

; using starter code and class notes, fill in the blanks to write make-queue

```
(define (make-queue)
  (let ((front-ptr '())
        (rear-ptr '()))

    ; empty-queue? is written to align with the way front-ptr
    ; and rear-ptr were given, above
    (define (empty-queue?)
      (null? front-ptr))

    ; peek returns the datum at the front of the queue
    ; peek returns #f if the queue is empty
    (define (peek)
      (cond ((empty-queue?) (error "Empty queue.  :-("))
            (else (car front-ptr))))

    ; insert-queue! plays out differently depending on whether
the queue
    ; is currently empty or not
    (define (insert-queue! datum)
      (let ((new-node (cons datum '())))
        (cond ((empty-queue?)
               (set! front-ptr new-node)
               (set! rear-ptr new-node))
              (else (set-cdr! rear-ptr new-node)
                    (set! rear-ptr new-node)))))

    ; delete-queue! has three possibilities:
    ; * empty queue
    ; * one element in queue
    ; * more than one element in queue
    (define (delete-queue!)
      (cond ((empty-queue?) (error "Empty queue.  :-("))
            (else
                    ; store the datum at the head of the queue
                  (let ((return-value (peek)))
                    ; update the front pointer
                    (set! front-ptr (cdr front-ptr))
                    ; If there was only one thing in the queue,
then the
                    ; rear-ptr will need to be set to nil
                    (if (null? front-ptr) (set! rear-ptr null))
                    ; Now return the element of the queue (or
#f)
                    return-value))))
    (define (dispatch message)
      (cond ((eq? message 'insert-queue!) insert-queue!)
            ((eq? message 'delete-queue!) delete-queue!)
            ((eq? message 'peek) peek)
            ((eq? message 'empty?) empty-queue?)))
    dispatch))

; to test
(define q (make-queue))
((q 'insert-queue!) 'a)
((q 'insert-queue!) 'b)
((q 'peek))
((q 'delete-queue!))
((q 'peek))
```

After Test 3

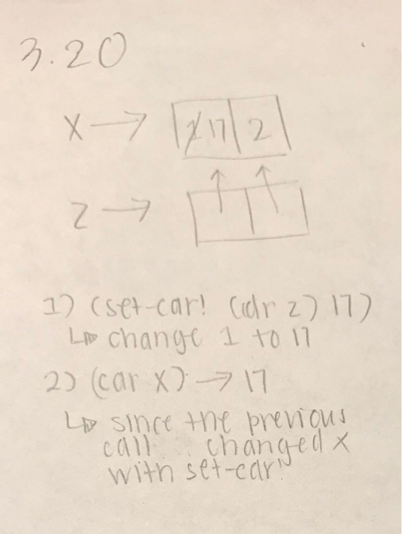
**3.17**

Devise a correct version of the count-pairs procedure of exercise 3.16 that returns the number of distinct pairs in any structure.

```
; must keep track of the pairs that have already been seen
; use eq? for same memory location
; using starter code and notes in starter code
(define (cp pair)
  (let ((visited '()))
    (define (visited? pair vl)
      (if (null? vl)
          #f
          (if (eq? pair (car vl))
              #t
              (visited? pair (cdr vl)))))
    (define (cp2 pair)
      (cond ((not (pair? pair)) 0)
            ((visited? pair visited) 0)
            (else (set! visited (cons pair visited))
                  (+ (cp2 (car pair))
                     (cp2 (cdr pair))
                     1))))
    (cp2 pair)))
```

**3.20**

Draw environment diagrams to illustrate the evaluation of the sequence of expressions



1) (set-car! (cdr z) 17)
↳ change 1 to 17

2) (car x) → 17
↳ since the previous call changed x with set-car!

**3.25**

Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys and different values may be stored under different numbers of keys. The lookup and insert! procedures should take as input a list of keys used to access the table.

```
; treat key as a list
; use let to create local-table, put all functions into one big
function, remove table from parameters and replace with
local-table
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (assoc key records)
      (cond ((null? records) #f)
            ((equal? key (caar records)) (car records))
            (else (assoc key (cdr records)))))
    (define (lookup key)
      (let ((record (assoc key (cdr local-table))))
        (if record
            (cdr record)
            false)))
    (define (insert! key value)
      (let ((record (assoc key (cdr local-table))))
        (if record
            (set-cdr! record value)
            (set-cdr! local-table
                      (cons (cons key value) (cdr
local-table)))))
      'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup) lookup)
            ((eq? m 'insert!) insert!)
            (else "i dont know what you are talking about")))
    dispatch))
```

**3.27**

optional