1. (10 points) Consider the define-struct given below:

\[
\text{(define-struct pos (x y))}
\]

(a) (3 points) Define pos struct variables named \(p_1\), \(p_2\), and \(p_3\) with any numbers you choose for the \(x,y\) fields in each pos.

(b) (2 points) Define a list of pos structs called \(\text{pos-list}\), that contains each of the three pos structs you defined in part (a).

(c) (5 points) Design the function \text{translate}. It consumes and produces a list of pos structs. For each pos with fields \(x\) and \(y\) in the input list, the output list should contain a pos with field \(x\) unchanged and the value in field \(y\) incremented by 1. Write a header comment with usage, input, and output statements; write 1 check-expect statement, using your list from part (b); and then define the function.
2. (10 points) Consider the function `average-list`, given below (assume the function works correctly and finds the average of all numbers in listy):

```scheme
;; Contract: (average-list listy) -> number
;; Input: listy = list of numbers
;; Purpose: Find the average of a list of numbers.
(define (average-list listy)
  (letrec
    [(len (length listy))
      (sum 0)
      (sum-acc
        (lambda (lis)
          (cond
            [(empty? lis)
              (/ sum len)]
            [else
              (set! sum (+ (first lis) sum))
              (sum-acc (rest lis))
            ])))]
    (sum-acc listy)))
```

Rewrite `average-list` below so that it uses iteration (i.e., a `while`, `dolist`, or `dotimes`) instead of a recursion.
3. (5 points) A left rotation of an ordered collection of items $a_0, a_1, \ldots, a_n$ produces the collection $a_1, a_2, \ldots, a_n, a_0$, where all items are shifted to the left and $a_0$ is put on the right end of the collection. Write a function \texttt{vleft!} that consumes a non-empty vector called \texttt{vec} and rotates the elements in positions 1...(sub1 (vector-length vec)) to the left, changing the value at the last position in the vector to hold the item originally in position 0 of the vector.

The contract, input, purpose, pre-function tests, and a helper function (\texttt{swap!}) are given for you.

```scheme
;; Name: swap!
;;------------------------------------------------------------------------
;; Usage: (swap! vecty p1 p2) -> vector of any valid type
;; Input: vecty = vector of any valid type
;; p1,p2 = numbers in the range 0...((vector-length vecty) - 1)
;; Output: mutated vector

;; Pre-function tests:
(check-expect (swap! (vector 1 2 3 4) 0 2) (vector 3 2 1 4))
(check-expect (swap! (vector 5 6) 0 1) (vector 6 5))

(define (swap! vecty p1 p2)
  (local
    [(define temp (vector-ref vecty p1))]
    (vector-set! vecty p1 (vector-ref vecty p2))
    (vector-set! vecty p2 temp)
    vecty))

Suppose you ran the swap! function as follows:

```(swap! #(2 3 4 5) 2 3)`

Why does this line return an error?
;; Name: vleft!
;;-----------------------------------------------------------------------
;; Usage: (vleft! vec) -> non-empty vector of any valid type
;; Input: vec = non-empty vector of any valid type
;; Output: A vector that contains the elements of vec rotated one
;; position to the left, moving item originally on the
;; left to the right end of vec.

;; Pre-function tests:
(check-expect (vleft! (vector 1 2 3 4)) (vector 2 3 4 1))
(check-expect (vleft! (vector 4 3 2 1)) (vector 3 2 1 4))

;; Function definition of vleft!
(define (vleft! vecty)