1. (10 points) This question refers to the define-struct statement below:

```
(define-struct robot (dof type proto))
```

where `dof` is a number representing the degrees of freedom of the robot, `type` is a string describing the type of robot (e.g. "humanoid", "aerial", "grasping", or "modular") and `proto` is a boolean that is true if the robot has been prototyped and false otherwise.

Assume this define-struct statement has been executed as part of a Racket program. When asked to write a contract, remember this includes the name of the function, the type of inputs to the function, and the type of output produced. NOTE: When it asks for the contract, giving the usage is sufficient.

(a) (1 point) Give the usage, input, and output for the constructor function of a robot:

- **Name:** make-robot
- **Usage:** `(make-robot d t p) --> robot`
- **Input:** `D` is a number, `T` is a string, and `P` is a boolean
- **Output:** robot struct

(b) (2 points) Define a named variable of robot type:

```
(define r1 (make-robot 6 "humanoid" true))
```

(c) (3 points) List the contracts for the accessor functions of a robot:

- `(robot-dof r) -> number; r is a robot struct`
- `(robot-type r) -> string; r is a robot struct`
- `(robot,proto r) -> boolean; r is a robot struct`

(d) (3 points) List the contracts for the mutator functions of a robot:

- `(set-robot-dof! r n) -> void; r is a robot struct and n is a number`
- `(set-robot-type! r s) -> void; r is a robot struct and s is a string`
- `(set-robot,proto! r b) -> boolean; r is a robot struct and b is a boolean`

(e) (1 point) Give the contract for the type-checker function of a robot:

```
(robot? a) --> boolean; a is any valid data type
```

2. (10 points) Write a definition of a Racket procedure called `sum-square-diff`. This function takes a vector `vnum` of numbers and a particular number `k` as arguments. It subtracts `k` from each member of `vnum`, squares each difference, and returns the sum of the squares.

Complete the function header comment (contract) by looking at the function tests, input, and output. Carefully read over the pre-function tests provided and write an extra test case. Lastly, write the function. Pay close attention to the test statements when writing the function.

```
(require test-engine/racket-tests)

; Name:    sum-square-diff
;---------
; Usage:   (sum-square-diff vnum k) --> number
; Input:   VNUM, a vector of numbers
```
; K, a number
; Output: subtracts K from each element of VNUM, squares each difference,
; and returns the sum of the squares

; Pre-function tests:
(check-expect (sum-square-diff (vector) 17) 0)
(check-expect (sum-square-diff (vector) 0) 0)
(check-expect (sum-square-diff (vector 2 4 6) 4) 8)
(check-expect (sum-square-diff (vector 1 2) -1) 13)
; WRITE A UNIQUE CHECK-EXPECT STATEMENT BELOW THIS LINE:
(check-expect (sum-square-diff (vector 2 2) -1) 18)

; Function definition: (using tail-recursion)
(define sum-square-diff
  (lambda (vnum k)
    (letrec [(vlen (vector-length vnum))
              (s-s-acc
               (lambda (index sum-acc )
                 (cond
                   [(= vlen index) sum-acc]
                   [else
                    (let [(curr (vector-ref vnum index))]
                      (s-s-acc (add1 index)
                       (+ sum-acc (* (- curr k) (- curr k))))))]))
      (s-s-acc 0 0))))

; Name: sum-square-diff-it
;--------------------------------------------------------
; Usage: (sum-square-diff-it vnum k) --> number
; Input: VNUM, a vector of numbers
;        K, a number
; Output: subtracts K from each element of VNUM, squares each difference,
;        and returns the sum of the squares

; Pre-function tests:
(check-expect (sum-square-diff-it (vector) 17) 0)
(check-expect (sum-square-diff-it (vector) 0) 0)
(check-expect (sum-square-diff-it (vector 2 4 6) 4) 8)
(check-expect (sum-square-diff-it (vector 1 2) -1) 13)
(check-expect (sum-square-diff-it (vector 2 2) -1) 18)

; Function definition: (using iteration)
(define sum-square-diff-it
  (lambda (vnum k)
    (letrec ((vlen (vector-length vnum))
              (sum 0)
              (sqr
               (lambda (x)
                 (* x x))))
      (do ((i vlen)
           (let ((curr (vector-ref vnum i)))
            (set! sum (+ sum (sqr (- curr k))))))
          sum)))

;; Higher order version uses map and then apply
(check-expect (sum-square-diff-ho (vector) 17) 0)
(check-expect (sum-square-diff-ho (vector) 0) 0)
(check-expect (sum-square-diff-ho (vector 2 4 6) 4) 8)
(check-expect (sum-square-diff-ho (vector 1 2) -1) 13)

;; Higher order version uses map and then apply:
(define sum-square-diff-ho
  (lambda (vnum k)
    (let* [(lnum (vector->list vnum))]
      (apply + (map (lambda (x) (* (- x k) (- x k))) lnum)))))

3. (10 points) Finish the definition of the function encode that takes a vector of numbers as input and returns a vector of the same length. Each value in the output vector is the result of multiplying the corresponding element from the input vector by its position in the input list. This function can be solved many different ways, including using a dotimes and mutation or by using regular recursion over the index number with mutation.

;; Name: encode (non-destructive with iteration)
;--------------------------------------------------------
;; Usage: (encode vnum) --> vector
;; Input: VNUM, a vector of numbers
;; Output: A vector of numbers in which each element of VNUM
;; is multiplied by its indexed position

(check-expect (encode (vector)) #())
(check-expect (encode (vector 1 1 1 1)) #(0 1 2 3))
(check-expect (encode (vector 1 2 4 2)) #(0 2 8 6))

(define encode ;; dotimes version with mutation of internal vector
  (lambda (vnum)
    (let* [(vlen (vector-length vnum))]
      (dotimes (i vlen)
        (vector-set! i (* i (vector-ref vnum i))))
      v1)))

;; function definition with internal recursion and mutation:
(define encode! ;; destructive with recursion
  (lambda (v1)
    (letrec
      ((help-encode!
        (lambda (pos)
          (cond
            [(= pos (vector-length v1)) v1]
            [else
             (vector-set! pos (* pos (vector-ref v1 pos)))
             (help-encode! (add1 pos)))]))
        (help-encode! 0))))

4. (10 points) Write the function stutter that takes a string as input and returns a string of twice the length, in which each letter is repeated.

Complete the contract (2 points), header (1 point), and purpose (1 point). Carefully read over the pre-function tests provided and write an extra test case (1 point). Lastly, write the function (5 points). Pay close attention to the test statements when writing the function and include comments.
; Contract: (stutter str) -> string
; Input: STR is a string
; Output: A string with all characters in STR repeated twice, sequentially.

; Pre-function tests:
(check-expect (stutter "horse") "hhoorrssee")
(check-expect (stutter "") "")
(check-expect (stutter "cat") "ccaatt")
(check-expect (stutter "polo") "ppoolloo")
(check-expect (stutter "madam") "mmaaddaamm") ;; still a palindrome! :)

; Function definition:
(define stutter
  (lambda (str)
    (letrec
      ;; convert str into list of characters
      ((lsstr (string->list str))
       ;; internal function will do recursion over lsstr
       (stut
        (lambda (lst chacc)
          (cond
            ;; base case: return reverse of chacc
            [(empty? lst) (reverse chacc)]
            ;; recursive case: cons first of lst twice onto the chacc
            [else
             (stut (rest lst)
                 (cons (first lst) (cons (first lst) chacc)))])
          )
        ;; initial call to inner function
        (list->string (stut lsstr empty))))))

5. (10 points) For each of the higher order function invocations below, write the result they will return when evaluated by DrRacket. You can assume all function invocations are syntactically legal; i.e., they will not cause an error. Keep in mind that map can consume an x-argument function if it also consumes x lists of equal length.

(filter (lambda (n) (and (even? n) (> n 200))) (list 280 90 120 543 666))
=> '(280 666)

(filter (lambda (p) (<= (posn-x p) (posn-y p)))
  (list (make-posn 5 5) (make-posn 4 2) (make-posn 3 8)))
=> '((make-posn 5 5) (make-posn 3 8))

(map (lambda (x y z) (+ (* x y) z))
  (list 1 2 3 4) (list 1 1 2 2) (list 4 3 2 1))
=> '(5 5 8 9)

(map (lambda (num) (abs num)) (list -10 2.5 3.14 -11.5 1.7))
=> '(10 2.5 3.14 11.5 1.7)

(apply + (map (lambda (z) (if (> 100 z) z 0))
  (list 543 12 57 101 99.55)))
=> 168.55