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.

(a) (1 point) Give the contract for the constructor function of a `robot`:

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

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

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

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

2. (10 points) Write a definition of a Racket function called \textbf{sum-square-diff}. This function takes a vector \textit{vnum} of numbers and a particular number \textit{k} as arguments. It subtracts \textit{k} from each member of \textit{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.

; Name:
;-----------------------------------------------------------------------
; Usage:
; Header:
; Output:

; 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:

; Function definition:
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 using an accumulator and tail recursion, an accumulator and iteration, or by using regular recursion based on the position number (an extra parameter).

```scheme
;; Name:
;; ----------------------------------------------
;; Usage:
;; Header:
;; Output:

;; Pre-function tests:
(check-expect (encode empty) empty)
(check-expect (encode (list 1 1 1 1)) (list 0 1 2 3))
(check-expect (encode (list 1 2 3 4)) (list 0 2 6 12))
(check-expect (encode (list 8 0 5 6)) (list 0 0 10 18))

;; Function definition:
```
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 header, input, and output. 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.

```lisp
; Name:
;===============================================================================
; Usage:
; Header:
; Output:

; Pre-function tests:
(check-expect (stutter "horse") "hhoorrssee")
(check-expect (stutter ") "")
(check-expect (stutter "cat") "ccaatt")
; WRITE A UNIQUE CHECK-EXPECT STATEMENT BELOW THIS LINE:

; Function definition:
```
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.

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

=> ___________________________________________________________
```

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

=> ___________________________________________________________
```

```scheme
(map (lambda (x y z) (+ (* x y) z))
     (list 1 2 3 4) (list 1 1 2 2) (list 4 3 2 1))

=> ___________________________________________________________
```

```scheme
(map (lambda (num) (abs num)) (list -10 2.5 3.14 -11.5 1.7))

=> ___________________________________________________________
```

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

=> ___________________________________________________________
```