CS 505
Autumn 1997
Assignment 1 — Scheme
Due Oct 8

1. Write and test a recursive Scheme function range that takes two arguments: start and stop. It should return the list of integers between start and stop inclusive. Examples:

(range 1 5)	evaluates to	(1 2 3 4 5)
(range 0 0)	evaluates to	(0)
(range 5 4)	evaluates to	()

   You don't need to check for non-numeric arguments.

2. Consider the following Scheme procedures.

```
;;; sum the numbers in a list of numbers
(define (sum s)
   (if (null? s) 0
       (+ (car s) (sum (cdr s)))))

;;; sum the numbers at the fringes of a tree
(define (sum-tree s)
   (cond ((null? s) 0)
         ((number? s) s)
         ((pair? s) (+ (sum-tree (car s)) (sum-tree (cdr s))))
         (else (error "non-numeric element in tree"))))
```

   For sum and sum-tree, can you write a tail recursive version? If so, write and test it; if not, explain why not.

3. Is your range procedure from Question 1 tail recursive? If not, can you write a version that is tail recursive? If you can, show it working; if not, explain why one can't write a tail recursive version.

4. This question is about lexical scoping. Consider the following definitions:

```
(define (test x y)
   (let ((x 100))
      (+ x y)))

(define (mollusc n p)
   (define (crab m)
      (+ m n))
   (if (< n 3) (mollusc (+ n 1) crab) (p 100)))
```

   What do the following expressions evaluate to? Explain how Scheme produces these answers. (You can try these if you want.)

   (a) `(test 1 2)`

   (b) `(mollusc 1 (lambda (x) (error "I should never be called")))`

5. The lecture notes include a `reduce` function as an example of a built-in higher order function. This version of `reduce` requires a base value. However, many of Scheme's functions that you would feed to `reduce`, such as `+` and `*`, return the identity element for that function when called with no arguments. Write a function, called say `my-reduce`, that takes just two arguments: a function and a list, and that works with lists of any length. It should assume that the function can be called with no arguments, and if so will return the identity element for that function. Thus:

   ```
   (define (sum lst) (my-reduce + lst))
   ```

   would be a correct definition of the `sum` function.

   (Note that this example is a bit contrived, since `+` can actually take an indefinite number of arguments and there is thus less motivation to use `reduce` — the point of this question is to use higher-order functions. Note also that the answer to this question is much shorter than the question ...)

6. Write a `tree-reduce` function that works on trees rather than just lists. Given this function you should be able to define the `tree-sum` function from Question 2 as:

   ```
   (define (tree-sum s) (tree-reduce + s))
   ```