Homework Assignment #1

Due Wednesday 10/7

(All assignments will be turned in at start of lecture.)

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)    → (1 2 3 4 5)
(range 0 0)    → (0)
(range 5 4)    → ()
```

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

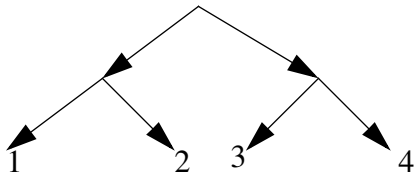
2. Is your `range` procedure tail recursive? If not, can you write a version that is tail recursive? If you can, show it working; if not, explain why one can't write a tail recursive version.

3. In Scheme, binary trees can be represented using cons cells for internal nodes to hold pointers to the left subtree (in `car`) and right subtree (in `cdr`); empty subtrees can be represented with nil, leaf nodes with non-list values (like an integer). The following expression, for example, constructs a simple tree with four leaf nodes:

```
(cons (cons 1 2) (cons 3 4))    → ((1 . 2) 3 . 4))
```

(Scheme tries to print it as a list which it isn't.)

The tree looks like this:



The following predicates are used to work with trees represented this way:

`(pair? s)`	true if `s` is an internal node
`(null? s)`	true if `s` is an empty subtree
`(leaf? s)`	true if `s` is a leaf; it should be defined as: `(define (leaf? s) (not (or (pair? s) (null? s))))`

Consider the following Scheme procedure:

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

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

4. Write a `tree-reduce` function that works on trees rather than lists. It should have four arguments: the reduction function; its identity element, returned in case of an empty tree; the leaf-value function which, given a leaf of the tree, returns the value to be reduced; and the tree itself. Given this function you should be able to define the `tree-sum` function from the previous question as:

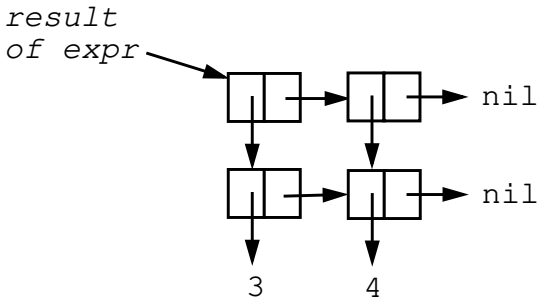
```
(define (sum-tree s) (tree-reduce + 0 (lambda (leaf) leaf) s))
```

5. Consider the following code:

```
(define (add-n n)
   (lambda (x) (+ x n)))
(define f1 (add-n 3))
```

When computing `(f1 5)`, what is the value of `n` in the expression `(+ x n)`, and where is it set? What would evaluating `(f1 5)` do if Scheme were dynamically scoped, and why?

6. For the following box-and-pointer diagram, give a **single** Scheme expression that evaluates to the data structure. Do not use side-effects. Hint: exploit `let`.



7. For the following sequence of Scheme expressions, illustrate using a box-and-pointer diagram the final data structure(s) resulting from evaluating the sequence, showing where the variables `a` and `b` point into the final data structure(s).

```
(define a '(hi (there) bob))
(define b (car (cdr a)))
(set! a (cdr a))
(set-car! b 'new)
(set-cdr! b (car a))
```

8. Why are lists better suited than arrays for manipulation by recursive programs?