Warmup: List Accessors

What does the evaluator print for each of the following expressions?

1. (car (cons 1 'A))

2. (car (list 1 'A))

3. (cdr (cons 1 'A))

4. (cdr (list 1 'A))

5. (cdr (cdr (cons 7 (cons 'B 2)))))

6. (cddr (cons 7 (cons 'B 2))))

7. (car (cdr (list 1 'B 3)))

8. (cadr (list 1 'B 3))

9. (second (list 1 'B 3))

10. (cdr (car (list 1 'A 3)))

11. (fourth (list 1 2 'C))
Quoting and Symbols

- ’foo is a way to get the symbol foo rather than the value of the thing named foo.
- The quote syntax ’<expr> can be thought of as syntactic sugar for (quote <expr>.
- String literals such as "spin wheel" and "foo" are not symbols. Rather, they are self-evaluating expressions (just as numbers are self-evaluating).
- The object returned via quote is a constant, and cannot (should not) be modified.
- The primitive eq? checks for symbol equality (case insensitive): (eq? ’a ’A) returns #t.
- If one argument to eq? is a symbol but the other is not, eq? returns #f. If neither argument is a symbol, the result is implementation dependent, except (eq? nil nil) which returns #t.

Draw box-and-pointer diagrams and give the printed representations for:

(cons ’a ’(b))

(cons ’(a) ’(b))

'(a b)

(list ’(a) ’(b))

(append ’(a) ’(b))
Trees as Nested Lists

A conventional representation of trees is achieved using a nested list structure. Each node in the tree is represented as a list of the children of that node, where a child may be either another tree or a leaf node. A leaf node is anything that is not a pair (e.g., a symbol or a self-evaluating value). Draw a box-and-pointer structure for the following tree using this convention.

```
  1
 /|
/ 5
 |
|
  4
 |
/|
/ 2
 |
/|
/ 3
```

Write a procedure `double-tree` that returns a new tree (in the list representation) with double the value of all leaf nodes.

An advantage of representing trees as lists is that we can use list procedures. Write the `double-tree` procedure using the `map` procedure.

Generalize this to a `map-tree` procedure which will perform some operation on all the leaf nodes of a tree, e.g., `(map-tree square mytree)`. 
Write a procedure `fringe` that returns a list of all the leaf values in a tree. For example `(fringe '((1) (2 (3 (((5)))) 4)))` should return `(1 2 3 5 4)`.

Once again, we can capture a common pattern in tree accumulation. Complete the following.

```
(define (accumulate-tree termproc combiner init tree)
  (cond ((null? tree) )
      ((not (pair? tree)) (tree))
      (else (accumulate-tree termproc combiner init
                       (accumulate-tree termproc combiner init
                                        (tree))))))
```