Worksheet 13: Michael Collins

1. **Occurrences**

Write the function `occurrences` that takes a number and a tree and counts the number of times that number appears in the tree. For example,

```
(define tree (list 1 2 (list (list 3 1) (list 1 2) 1)))
(occurrences 1 tree) ==> 4
```

```
(define (occurrences elt tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) (if (= elt tree) 1 0))
        (else (+ (occurrences elt (car tree))
                 (occurrences elt (cdr tree))))))
```

2. **Flatten**

Write the function `flatten` that takes a tree structure and returns a flat list of the leaves of the tree. For example

```
(fringe (list 1 (list 2) 3)) ==> (1 2 3)
```

```
(define (flatten x)
  (cond ((null? x) nil)
        ((not (pair? x)) (list x))
        (else (append (flatten (car x))
                      (flatten (cdr x))))))
```

3. **Deep Reverse**

So far, we've been working on lists, while we've ignored the elements of the list. What does the following return? `(reverse (list 1 (list 2 3) (list 4 5 6)))`

Write a function `deep-reverse` that when called on the above tree will reverse all the elements.

```
(deep-reverse (list 1 (list 2 3) (list 4 5 6))) ==> ((6 5 4) (3 2) 1)
```

(Hint: try using “map” and “reverse”)

```
(define (deep-reverse x)
  (if (not (pair? x))
      x
      (map deep-reverse (reverse x))))
```

4. **General Tree Manipulation**

We can generalize tree operations with the following procedure:

```
(define (tree-manip leaf-proc init merge-proc tree)
  (cond ((null? tree) init)
        ((leaf? tree) (leaf-proc tree))
        (else (merge-proc
               (tree-manip leaf-proc init merge-proc (car tree))
               (tree-manip leaf-proc init merge-proc (cdr tree))))))
```

This is similar to the list folding procedures that we have seen a while ago – but here ‘init’ is used for many nulls, and we use two input procedures for merging values and for operating on leaves. For the following assume this binding:

```
(define test-tree '(1 (2 (3 (4) 5) 6) 7))
```

4.1. Write a ‘count-leaves’ procedure using ‘tree-manip’, for example:

```
(count-leaves test-tree) => 7
```
4.2. Write a ‘flatten-tree’ procedure using ‘tree-manip’, for example:

```
(flatten-tree test-tree) => (1 2 3 4 5 6 7)
```

4.3. Write ‘sum-tree’ that sums the values of a tree of numbers, for example:

```
(tree-sum test-tree) => 28
```

4.4. Write ‘filter-tree’ that receives a predicate procedure and a tree, and returns a tree of the same structure, except that all values that the predicate does not hold for are removed. For example:

```
(filter-tree even? test-tree) => ((2 ((4)) 6))
(filter-tree odd? test-tree) => (1 ((3 () 5)) 7)
```

5. More practice with vectors. We’re now going to create a set of matrix operations, which are analogous to vector operations:

```
(make-matrix m n val) ;; makes a matrix of dimension m*n
(matrix-set! x i j val) ;; sets x[i,j] to be val
(matrix-ref x i j) ;; returns value of x[i,j]
```

5.1 First, write a procedure for make-matrix. Hint: you should try to create a vector x of m elements, where each element is a vector of size n.

5.2 Now write versions of matrix-set! and matrix-ref. Also write a function (size x) which returns a pair (cons m n) where m and n are the dimensions of the matrix.