Assignment 8
Due 4:00pm on Tuesday, November 29

For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch. Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include reference to the parameter names of your functions. The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources. Test data for all questions will always satisfy all conditions stated for consumed data.

Please read the course Web page for more information on assignment policies and how to organize and submit your work. Be sure to download the interface file from the course Web page and to follow all the instructions listed in the style guide (on the Web page). Specifically, your solutions should be placed in files a8qY.rkt, where Y is a value from 1 to 4. For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe, including the definition of constants and helper functions where appropriate.

Language Level: Beginning Student with List Abbreviations

Coverage: Module 9

Note: We strongly recommend the careful use of templates on this assignment in particular.

The following structure and data definitions are required for the assignment. You will also need to use the data and structure definitions from the `compound` teachpack.

```
(define-struct ae (fn args))
;; An arithmetic expression (aexp) is either a number, or a
;; structure (make-ae f alist) where f is a symbol and alist is
;; an aexplist.
;;
;; An aexplist is either empty or (cons a alist) where a is an
;; aexp and alist is an aexplist.

(define-struct t-node (label children))
;; A general tree (gen-tree) is either a string or a structure
;; (make-t-node l c), where l is a string and c is a tree-list.
;; A tree-list is either (cons t empty) or (cons t tlist) where
;; t is a gen-tree and tlist is a tree-list.

;; A leaf-labelled tree (llt) is one of the following:
;; empty
;; (cons l1 l2) where l1 is a non-empty llt and l2 is a llt
;; (cons v l) where v is an integer and l is a llt
```

1. The height of a `gen-tree` is 0 if it is a string. The height of a `gen-tree` of the form `(make-t-node l c)` is one more than the maximum height over all trees in `c`. For example, the height of `mytree` as defined below is 3.

```
(define mytree
  (make-t-node "a"
               (list "b"
                     (make-t-node "c"
                                  (list "e"
                                        (make-t-node "f"
                                                     (list "h" "i" "j"))))
                     (make-t-node "d"
                                  (list "g")))))
```

   Use structural recursion to complete a function `height` that consumes a `gen-tree` and produces its height.

2. Use structural recursion to complete a Scheme function `is-organic?` that consumes a compound and produces `true` if the `compound` contains the element `carbon` (a constant defined in the `compound` teachpack) and `false` otherwise. Two examples are below.

```
(is-organic? cinnamaldehyde) => true
(define neon
  (make-compound 'neon (list (make-part 1 (make-element 'N 20.18)))))
(is-organic? neon) => false
```

   Add the `compound` teachpack to your solution. However, do **not** copy any definitions from the teachpack (there will be errors if you do). You may use the data defined in the teachpack for testing.

3. Use structural recursion to complete a Scheme function `double-odds` that consumes a `llt` (named `allt`) and produces an `llt` that is like `allt` except that all odd integers in `allt` have been doubled. Two examples are below.

```
(double-odds '(2 1)) => (list 2 2)
(double-odds '((4 2) 3 (4 1 6))) => '((4 2) 6 (4 2 6)))
```

4. The function `eval` in Module 9 evaluates a general arithmetic expression. To produce a final result, an intermediate result is produced every time the built-in functions for multiplication and addition are applied. Use structural recursion to complete a Scheme function named `num-0-results` that consumes a general arithmetic expression and produces the number of times 0 is the result of the application of the built-in function `+` or `*` during the execution of `eval` on the expression. Some examples follow.

```
(num-0-results 0) => 0
(num-0-results (make-ae '+ (list 0))) => 1
(num-0-results (make-ae '+ empty)) => 0
(define myex1 (make-ae '+ (list 0 2 -2 3 1 -4 0)))
(num-0-results myex1) => 4
(define myex2 (make-ae '* (list 3 0 2 7)))
(num-0-results myex2) =>  2
(define myex3 (make-ae '* (list myex1 3 0 2 7)))
(num-0-results myex3) => 7
```

*Hint: You might want to write a helper function that returns both the result of an expression and the number of times 0 is the result of applying + or *. Since Scheme functions only produce single values, you will need to bundle these two numbers together.*