Assignment: 8

Due: Monday, July 20 at 4:00 pm
Language level: Beginning Student with List Abbreviations
Coverage: Modules 1–9

For this and all subsequent assignments, to receive full marks you are required to use the design recipe for every function you write. You may include the examples given in the assignment in your submissions, but they will be ignored by the markers—you must develop a complete suite of examples and tests on your own. For your convenience, an interface file which contains the headers of the required functions is available on the course webpage.

Do not send any code files to course staff; they will not be accepted. Submissions must be made via MarkUs as described on the course webpage. After submission, check your basic test results to ensure your files were properly submitted. Solutions that do not pass the basic tests are unlikely to receive any correctness marks.

Remember, the solutions you submit must be **entirely your own work**.

1. This question uses the following data definition:

```
;; A BooleanExpression is one of:
;; * a Bool
;; * a (list (anyof 'even? 'odd?) Int)
;; * a (list (anyof '> '< '=) Int Int)
;; * a (list 'not BooleanExpression)
;; * a (cons (anyof 'and 'or) (listof BooleanExpression))
```

   (a) Write a template for a function which consumes a *BooleanExpression*. Because the definition of *BooleanExpression* depends on (*listof BooleanExpression*), you should also include a template for a helper function which processes a (*listof BooleanExpression*). Include the template as comments at the top of the file you submit to MarkUs.

   (b) Write a Racket function *eval* which consumes a *BooleanExpression* and produces either *true* or *false*, following the rules given in Module 3 for how predicates and Boolean expressions are evaluated. The evaluation of a *Bool* should be the Boolean itself; otherwise, the symbol which occurs at the start of the *BooleanExpression* denotes which function should be applied, and the elements in the rest of the *BooleanExpression* denote the arguments to apply the function to.

   For example, (*eval true*), (*eval* (*list* 'even? 2)), (*eval* (*list* '> 2 1)), (*eval* (*list* 'not *false*)), (*eval* (*list* 'or *false true*)), and (*eval* (*list* 'and *true true*)) should all produce *true*.

   It is recommended to first complete part (a) to help understand the structure that the body of *eval* should follow before actually writing it.

2. Computer files are typically organized by storing them inside *directories*, with related files stored in the same directory. Directories may also contain other directories, leading to a hierarchy known as a *directory tree*. This question uses the following structure and data definitions to simulate a directory tree:

```racket
(define-struct directory (name contents))
;; A Directory is a (make-directory Str (listof File))
;; A File is one of:
;; * a Str (containing the name of the file; nonempty and does not contain "/")
;; * a Directory
```

As an example, the directory "cs115" given by the definitions

```racket
(define a1 (make-directory "a1" (list "a1.pdf" "a1q1.rkt" "a1q2.rkt" "a1q3.rkt")))
(define a2 (make-directory "a2" (list "a2.pdf")))
(define assns (make-directory "assns" (list a1 a2)))
(define labs (make-directory "labs" (list "lab01.pdf" "lab02.pdf")))
(define final (make-directory "final" empty))
(define cs115 (make-directory "cs115" (list assns labs final "style.pdf" "survival.pdf")))
```

can be concisely represented diagrammatically as follows:

```
cs115
    assns
        a1
            a1.pdf
            a1q1.rkt
            a1q2.rkt
            a1q3.rkt
        a2
            a2.pdf
    labs
        lab01.pdf
        lab02.pdf
    final
    style.pdf
    survival.pdf
```

Write a Racket function *find* which consumes a *Directory* to search and a *Str* representing the name of a file to search for and produces *false* if the file cannot be found and otherwise produces the complete path of the file (a string containing the name of every directory that the file is contained in starting from the root of the directory tree, with names separated by "/").

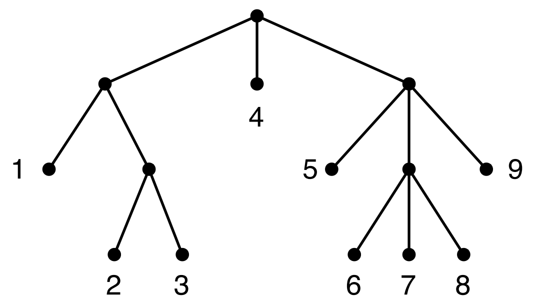
For example, (*find cs115* "final") should produce "cs115/final", (*find cs115* "a2.pdf") should produce "cs115/assns/a2/a2.pdf", and (*find cs115* "a3") should produce *false*. You can assume that all filenames in the given directory tree are unique (so there will be at most one match), and you do not need to worry about efficiency concerns.

3. Recall the definition of a leaf-labelled tree (*LLT*) as given on slide 25 of module 9:

   ;; An LLT is one of:
   ;; * empty
   ;; * (cons Num LLT)
   ;; * (cons LLT LLT), where the first LLT is nonempty

   Write a Racket function *max-children* which consumes an *LLT* and produces the maximum number of children that any node in the tree has.

   For example, say the following tree (from slide 23 in module 9) has been defined as *tree*:



(**define** *tree* (*list* (*list* 1 (*list* 2 3)) 4 (*list* 5 (*list* 6 7 8) 9)))

   Then (*max-children tree*) should produce 3 and (*max-children* (*first tree*)) should produce 2. Your function should produce 0 when given the *empty* tree.