1. **Feedback**. Complete the survey linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.

2. **Warm-Up: Continuations** To implement our regular expression parser, we introduce the idea of a continuation that captures "What to do next." Consider again the binary search trees from Labs 1 and 4:

   (a) First, let us reimplement *foldLeft* to take an additional parameter *sc* call the *success continuation*
   Completed in the code

   (b) Then, we implement *dfs* for a depth-first search of the tree
   Also completed in the code

3. **Regular Expressions.** Consider the syntax for a language of regular expressions shown in Figure 1. We note the corresponding Scala **case class** or **case object** used to construct abstract syntax trees of type RegExpr (shown below the grammar in full).

   (a) **Regular Expression Matcher: Continuations.** For this exercise, we will implement a basic backtracking regular expression matcher.
   Completed in the code

   (b) **Regular Expression Parser: Recursive Descent Parser**

      i. **Exercise.** In your write-up, give a refactored version of the *re* grammar from Figure 1 that eliminates ambiguity in BNF (not EBNF). Use the following template for the new non-terminal names:

$$
\begin{aligned}
re &::= union && (1)\\
union &::= intersect\{'|'intersect\} && (2)\\
intersect &::= concat(\qquad\qquad concat)\ldots && (3)\\
concat &::= not(not)\ldots && (4)\\
not &::= \ not|star\ldots && (5)\\
star &::= atom(*|+|?)\ldots && (6)\\
atom &::=!|\#|c|.|(re)\ldots && (7)\\
& && (8)
\end{aligned}
$$

help from https://github.com/untra/lab6/blob/master/src/main/scala/Lab6.scala

ii. **Exercise.** Explain briefly why a descent parser following your grammar with left recursion would go into an infinite loop.
My answer here

iii. **Exercise.** In your write-up, give a refactored version of the *re* grammar that replaces left-associative binary operators with n-ary versions using EBNF using the following template:

$$
\begin{aligned}
re &::= union && (9)\\
union &::= intersect\{'|'intersect\} && (10)\\
intersect &::= concat(\qquad concat)\ldots && (11)\\
concat &::= not(not)\ldots && (12)\\
not &::= \ not|star\ldots && (13)\\
star &::= atom(*|+|?)\ldots && (14)\\
atom &::=!|\#|c|.|(re)\ldots && (15)\\
& && (16)
\end{aligned}
$$

iv. **Exercise.** In your write-up, give the full refactored grammar in BNF without left-recursion and new non-terminals like *unions* for lists of symbols. You will need to introduce new terminals for *intersects* and so forth.

(c) **Regular Expression Literals in JavaScripty.** Let's extend our Lab 5 interpreter with regular expression literals and regular expression matching. We extend JAVASCRIPTY as follows.

i. **Exercise.** In your write-up, give typing and small-step operational semantic rules for regular expression literals and regular expression tests based on the informal specification given above. Clearly and concisely explain how your rules enforce the constraints given above and any additional decisions you made.

Your answer here

ii. *Optional.* Extend your Lab 5 interpreter to include regular expressions. The discussion above enables you to extend your Lab 5 interpreter so that you can have your own full JAVASCRIPTY interpreter!