SCALA LAB 2

1. Instructions

Download the files tester2.scala and lab2.scala (the file where you will put your code) into a directory. Put all of your code inside the Lab2 object in lab2.scala.

The tester2.scala file contains the entry point to your code and some tests. You can comment out tests for functions you have not implemented.

Upload only lab2.scala

2. Exceptions, Option, Either

Scala has many “wrapper” types that, when combined with pattern matching, allow you to do interesting things. The first of these allows you to handle exceptions.

2.1. **Handling exceptions.**

First, you will need to import Try, Success, and Failure from the util package. In order to catch an exception, put a Try{} around your code. The result will be either a Failure object or a Success object. For example:

```
val y = 1/0
val z = 2/1
```

Would throw an exception when defining y but defining z would not cause an error. To catch the exceptions, we could do the following:

```
import util.{Try, Success, Failure}
val y = Try{1/0}
val z = Try{2/1}
```

Here, when we defined y, we caught an exception, while in the definition of z, everything was normal.

You will notice that the value of y is now a Failure object, specifically `Failure(java.lang.ArithmeticException: / by zero)` and the value of z is the Success object `Success(2)`. In the case of Success, we want to get the result of our computation out of the object. There are several ways of doing this.

The first way, less preferred, uses the .isSuccess and .get methods (not there are no paranthesis following them):

```
val y = Try{1/0}
if(y.isSuccess) {
  println(y.get)
} else {
  println("exception thrown")
}
```

A more elegant approach uses pattern matching:

```
val z = Try{2/1}
z match {
   case Success(a) => println(a)
   case Failure(b) => println("exception thrown")
}
```

In many cases we need to catch exceptions when processing lists, vectors, etc. For example, the following code creates a list of strings and then tries to convert them into integers. It will fail with an exception because ”hello” cannot be converted to an integer:

```
val a = List("1", "2", "3", "hello")
val b = a.map{x => x.toInt}
```

Instead, we can wrap the conversion inside a Try, then use a filter to keep only the successes and then use .get to obtain the values:

```
val a = List("1", "2", "3", "hello")
val b = a.map{x => Try{x.toInt}}
val filtered = b.filter{x => x.isSuccess}
val result = filtered.map{x => x.get}
```

THe result is List(1,2,3).

2.2. **Option types.** Sometimes your function may want to return something and sometimes it might not. In this case, we would use option types. In the following code, we define a vals that stores an Option[Int] meaning that they could store an integer (the class Some) or could store nothing (the class None).

```
val y: Option[Int] = None //stores nothing
val z: Option[Int] = Some(3) //stores the number 3
```

we can access the value that is stored using pattern matching:

```
val y: Option[Int] = Some(3)
y match {
   case None => println("nothing")
   case Some(x) => println(x)
}
```

This code will print a 3.

2.3. **Either types.** Sometimes you may want a variable to have possibly two different types. For example, suppose we are expecting a number and we let the user either provide it directly as an integer or as a spelling of a number (e.g., myfunction(1) or myfunction("one"). We can use Either to specify the type as follows (currently the function doesn't do anything):

```
def myfunction(x: Either[Int, String]) = {}
```

if we want to puss in an integer (the left type), we call it as myfunction(Left(3)) and if we want to pass in a string (the right type) we call it as myfunction(Right("one")). We can extract the values using pattern matching. For example:

```
def myfunction(x: Either[Int, String]) = {
   x match {
      case Left(y) => y //y is the int
      case Right("one") => 1 //the value is 1
      case Right("two") => 2
      case Right(x) => 3 //all other values are converted to 3
   }
}
```

If we call this function as myfunction(Left(4)), it returns 4. If we call myfunction(Right("one")), it returns 1 and if we call myfunction(Right("seven")) it returns 3.

3. Questions

Question 1. *In the Lab2 object, write a function* ***upper*** *whose input is a paremeter called* ***mystrings*** *whose type is a vector of strings. It returns the vector of capitalized strings. For example*

```
val a = Vector("penn", "state")
val b = upper(a)
```

should result in b being Vector("PENN", "STATE"). Strings have a .toUpperCase method (without parentheses). Hint: use map. ***The body of this function should be 1 line***

Question 2. *In the Lab2 object, write a function* ***su*** *whose input parameter is called mystrings and is a vector of strings. The output should be a vector of those strings that contian "su" in them. For example su(Vector("psu", "usc")) should return Vector("psu"). Strings have a .contain method. Hint: use filter.* ***The body of this function should be 1 line***

Question 3. *In the Lab2 object, write a function* ***tot*** *whose input parameter is called mystrings and is a vector of strings. The function should return the total length of the strings in the input. For example, tot(Vector(”penn”, ”state”)) should return 9. Strings have a .size method (no parentheses). You must use foldLeft.* ***The body of this function should be 1 line.***

Question 4. *Arithmetic expressions can be represented as operator trees. For example, the expression (1+2) * (3+4) can be represented as the following tree:*

```
            *
           / \
          /   \
         /     \
        +       +
       / \     / \
      1   2   3   4
```

where the leaf nodes are numbers and the interior nodes specify an operation to perform. We will encode the nodes of the tree using a case class. This case class will store the operator (””, ”-”, ”/”, ”+”, or a Double), a left child, and a right child. The operator is one of ”*”, ”-”, ”/”, ”+” for internal nodes, but for leaves, the operator is simply a Double. Thus we need to use Either for operators. Since leaves do not have left or right children, we will need to use Option types for leaves. Our case class is then defined as follows*

```
case class Node(first: Option[Node], op: Either[String,Double], second: Option[Node])
```

first *refers to the left child,* ***op*** *is the operator and* ***second*** *is the right child.*

For example, the leftmost leaf in the tree above just contains the number 1. We can define its corresponding node as follows:

```
val leaf1 = Node(None, Right(1), None)
```

because it doesn't have any children (hence the None) and op field should be 1 (since Double is the Right type of Either[String, Double], we use Right(1))

Similarly, the leaf nodes 2,3,4 can be created as follows:

```
val leaf2 = Node(None, Right(2), None)
val leaf3 = Node(None, Right(3), None)
val leaf4 = Node(None, Right(4), None)
```

The two intermediate nodes that hold + can be created as:

```
val inner1 = Node(leaf1, Left("+"), leaf2)
val inner2 = Node(leaf3, Left("*"), leaf4)
```

and the root node:

```
val root = Node(inner1, Left("*"), inner2)
```

Your job is to write the function ***evaluate*** *whose input is a parameter called r and its type is Node. When we pass in a node of an operator tree, the function should evaluate the expression corresponding to that node and return the result. For example, for the nodes we defined, we should get the following results:*

- *evaluate(leaf1) should return 1.0*
- *evaluate(leaf3) should return 3.0*
- *evaluate(inner1) should return 3.0*
- *evaluate(root) should return 21.0*

This function should be recursive and should use pattern matching.