Homework 2

The purpose of this assignment is to get familiar with algebraic data types and to implement a first "baby" version of our JavaScript interpreter.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamilar to you.

Finally, make sure that your solution compiles and runs. A program that does not compile will *not* be graded.

For submission instructions and the due date, please see the 'README.md' file.

Problem 1 Data Structures Review: Binary Search Trees. (24 Points)

In this exercise, we will review implementing operations on binary search trees from Data Structures. Balanced binary search trees are common in standard libraries to implement collections, such as sets or maps. For example, the Google Closure library for JavaScript has `goog.structs.AvlTree`. For simplicity, we will not worry about balancing in this question. Trees are important structures in developing interpreters, so this question is also critical practice in implementing tree manipulations.

A binary search tree is a binary tree whose nodes satisfy an ordering invariant on their data values. Let t be any node in the binary search tree and let d be the data value stored in the node t. Further, let t's left child be l, and its right child be r. The ordering invariant states that all of the data values in the subtree rooted at l must be strictly smaller than d, and all of the data values in the subtree rooted at r must be strictly greater than d.

We will represent binary trees containing integer data using the following Scala case classes and case objects:

```
sealed abstract class Tree
case object Empty extends Tree
case class Node(left: Tree, data: Int, right: Tree) extends Tree
```

A `Tree` is either `Empty` or a `Node(l,d,r)` with left child `l`, data value `d`, and right child `r`. For this exercise, we will implement the following four functions:

(a) The function `repOk`

```
def repOk(t: Tree): Boolean
```

checks that an instance of `Tree` is a valid binary search tree. In other words, it checks using a traversal of the tree the ordering invariant. This function is useful for testing your implementation. A skeleton of this function has been provided for you in the template. **(4 Points)**

(b) The function `insert`

```
def insert(t: Tree, n: Int): Tree
```

inserts an integer into the binary search tree. Observe that the return type of `insert` is a `Tree`. This choice suggests a functional style where we construct and return a new output tree that is the input tree `t` with the additional integer `n` as opposed to destructively updating the input tree. If the value `n` is already present in the tree `t`, then `insert` should return a tree that is structurally equal to the input tree `t`. **(8 Points)**

(c) The function `deleteMin`

```
def deleteMin(t: Tree): (Tree, Int)
```

deletes the smallest data element in the search tree (i.e., the leftmost node). It returns both the updated tree and the data value of the deleted node. This function is intended as a helper function for the delete function. Most of this function is provided in the template. **(4 Points)**

(d) The function delete

```
def delete(t: Tree, n: Int): Tree
```

removes the first node with data value equal to `n`. If no such node exists, the tree should be returned unmodified. This function is trickier than `insert` because what should be done depends on whether the node to be deleted has children or not. We advise that you take advantage of pattern matching to organize the cases. **(8 Points)**

Problem 2 JakartaScript Interpreter: Arithmetic Expressions (16 Points)

JavaScript is a complex language and thus difficult to build an interpreter for it all at once. In our interpreter implementation, we will make some simplifications. We consider subsets of JavaScript and incrementally examine more and more complex subsets during the course of the semester. For clarity, let us call the language that we implement in this course JakartaScript. For the moment, let us define JakartaScript to be a proper subset of JavaScript. That is, we may choose to omit complex behavior in JavaScript, but we want any programs that we admit in JakartaScript to behave in the same way as in JavaScript.

In actuality, there is not one language called JavaScript but a set of closely related languages that may have slightly different semantics. In deciding how a JakartaScript program should behave, we will consult a reference implementation that we fix to be Google's V8 JavaScript Engine. We will run V8 via Node.js, and thus, we will often need to write little test JavaScript programs and run them through Node.js to see how the test should behave.

In this homework, we consider an arithmetic sub-language of JavaScript (i.e., an extremely basic calculator). The first thing we have to consider is how to represent a JakartaScript program as data in Scala, that is, we need to be able to represent a program in our object/source language JakartaScript as data in our meta/implementation language Scala.

To a JakartaScript programmer, a JakartaScript program is a text file–a string of characters. Such a representation is quite cumbersome to work with as a language implementer. Instead, language implementations typically work with trees called abstract syntax trees (ASTs). What strings are considered JakartaScript programs is called the concrete syntax of JakartaScript, while the trees (or terms) that are JakartaScript programs are called the abstract syntax of JakartaScript. The process of converting a program in concrete syntax (i.e., represented as a string) to a program in abstract syntax (i.e., represented as a tree) is called parsing.

For this homework, a parser is provided for you that reads in a JakartaScript program-as-a-string and converts into an abstract syntax tree. We will represent abstract syntax trees in Scala using **`case class`** and **`case object`** declarations. The correspondence between the concrete syntax and the abstract syntax representation is shown in Figure 1. You can find the relevant code in the file `src/main/scala/popl/js/ast.scala`.

(a) **Interpreter 1.** Implement the eval function

```
def eval(e: Expr): Double
```

that evaluates a JakartaScript expression `e` to the Scala double-precision floating point number corresponding to the value of `e`. **(16 Points)**

Consider a JakartaScript program `p`; imagine `p` stands for the concrete syntax or text of the JakartaScript program. This text is parsed into a JakartaScript AST `e`, that is, a Scala value of type `Expr`. Then, the result of `eval` is a Scala number of type `Double` and should match the interpretation of `e` as a JavaScript expression. These distinctions can be subtle but learning to distinguish between them will go a long way in making sense of programming languages.

At this point, you have implemented your first language interpreter!

```
sealed abstract class Expr extends Positional
/* Literals and Values */
sealed abstract class Val extends Expr
case class Num(n: Double) extends Val
Num(n)   n
/* Unary and Binary Operators */
case class UnOp(op: Uop, e1: Expr) extends Expr
UnOp(uop,e1)   uop e1
case class BinOp(op: Bop, e1: Expr, e2: Expr) extends Expr
BinOp(bop,e1,e2)   e1 bop e2

sealed abstract class Uop
case object UMinus extends Uop
Uminus  −

sealed abstract class Bop
case object Plus extends Bop
Plus  +
case object Minus extends Bop
Minus  −
case object Times extends Bop
Times  ∗
case object Div extends Bop
Div  /
```

Figure 1: Representing in Scala the abstract syntax of JakartaScript. After each **`case class`** or **`case object`**, we show the correspondence between the representation and the concrete syntax.