CS3101-2 Scala, Fall 2014: Problem Set 3

Daniel Bauer

Total points: 20
Due date: Nov 11, 11:59pm EST

Submission instructions:
Place the files for all problems in a directory named `[your_uni]_week[X]`, where X is the number of the problem set. For instance if your uni is xy1234 and you are submitting the problem set for the first week, the directory should be called `xy1234_week1`. Either zip or tar and gzip the directory (using `tar -c xy1234_week1 | gzip > xy1234_week1.tgz`) and upload it to your directory in the drop box for this class on Courseworks.

Part 1 - Function Composition

Please save your solution to the following problem in a file named `Part1.scala`.

a. (3 pt) **Composing functions:** Given two functions f and g, the composition f after g is defined to be the function $h(x) = f(g(x))$.

Define a Scala function `compose(f: Int=>Int, g: Int=>Int): Int=>Int` that implements composition. For example, `compose` should behave like this:

```
scala> val square = (x : Int) => x*x
square: Int => Int = <function1>

scala> val inc = (_ : Int) + 1
inc: Int => Int = <function1>

scala> val squareinc = compose(square, inc)
squareinc: Int => Int = <function1>

scala> squareinc(6)
49
```

b. (4 pt) **Repeated application:** Given a function f and a positive integer n, the n-th repeated application of f is the function $h(x) = f(f(\cdots(f(x))\cdots))$. For example, the 2nd repeated application of the function $f(x) = x + 1$ is $h(x) = f(f(x))$ and $h(2) = 3$.

Define a Scala function `repeat(f: Int=>Int, n: Int): Int=>Int` that returns the n-th repeated application of f. For example:

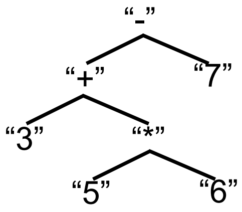
```
scala> val square = (x: Int) => x*x
square: Int => Int = <function1>

scala> val square4 = repeat(square, 4)
square4: Int => Int = <function1>

scala> square4(2)
res0 Int = 65536
```

Hint: It is convenient to use `compose` defined in part (a).

Part 2 - Object Oriented and Functional Programming

The file `Part2.scala` contains a class `Tree` that is used to represent simple binary trees. `Tree` is an abstract class that has two concrete subclasses `Node` and `Leaf`. All `Node` instances contains the attributes `left` and `right`, corresponding to the subtrees below this node. `Leafs` do not have any subtrees. All `Tree` instance also have a `content` attribute of some generic type `T`. The companion objects to `Leaf` and `Node` define apply methods that make it easy to construct binary trees like this:

```
val tree : Tree[String] =
         Node("-",
            Node("+",
               Leaf("3"),
               Node("*",
                  Leaf("5"),
                  Leaf("6"))),
            Leaf("7"))
```



A tree operation maps a `Tree` to some result. For instance, we would like to define a `toString` method that produces the following string for the tree above.

```
scala> val s = tree.toString
s: String = (- (+ 3 (* 5 6)) 7)
```

In this problem we will define sevaral tree operations as methods of `Tree`, making use of a shared functional abstraction. A general pattern underlying most tree operations is to traverse the tree depth first. At each node t we recursively compute a result for each subtrees and then combine these results with the content of t.

This pattern is implemented in the method `traverse[A](`*proc_node*`,` *proc_leaf*`)` which is defined for `Leaf`s and `Node`s.

On `Leaf` instances `traverse` calls *proc_leaf*, which is a method of type `T=>A`, where `T` is the type of the content of the tree (in the example `String`) and `A` is the type of the result of the tree operation (which is also `String` for the `toString` method).

On `Node` instances `traverse` calls *proc_node*, which is a method of type `(A,A,T)=>A`. The three parameters are 1) the result of recursively calling `traverse` on the left subtree 2) the result of calling `traverse` on the right subtree and 3) the content of the current node.

Solve the following problems by passing appropriate *proc_leaf* and *proc_node* function objects to `traverse`.

a. (4 pts) Preorder String Representation: Override the `toString` method on `Tree` to return a single `String` as shown in the example.

b. (4 pts) List of leaf values: Define a `leafs` method on `Tree` that returns a list of all leafs in the correct order. For instance, for the example tree above:

```
scala> tree.leafs
res2: List[String] = List(3, 5, 6, 7)
```

c. (5 pts) Evaluating Arithmetic Expressions: As shown in the example, binary tress can be used to encode the order of operations of arithmetic expressions like $(3 + (5 * 6)) - 7$. Such trees are also called Abstract Syntax Trees.

Define a method `evaluate` on `Tree` that evaluates the arithmetic expression encoded in the tree. Assume that all nodes of the tree are `String`s, but that they only describe integers or the operators "+", "-", and "*". Hint: You can use the `toInt` method on `String` instances to parse a `String` into an `Int`.