Advanced Programming IT712A
Assignment 02

SERGEY REDYUK

University of Skövde
Sweden
b16serre@student.his.se

1 Exercise

Consider the definition of an algebraic data type (binary) tree. The nodes of the tree are operations (at least two are possible "+" and "*") and the leaves are numbers (see Figure 1). Define the data type as polymorphic so that you can use at least two types of numbers (e.g., a real and a complex). Then, define the following functions.

1. A function preorder that given a tree makes a traversal in preorder and return a list of the elements. Recall that in a preorder, we first visit/display the root and then the two subtrees.
2. A function inorder that given a tree makes a traversal in inorder and return a list of the elements. You may consider adding parenthesis to avoid ambiguities. Recall that traversal in inorder means that first we visit/display the first subtree, then the root and then the other subtree.
3. A function evaluate that given a tree evaluates it and return its corresponding value.

For example, a possible definition for the tree in Figure 1 as well as the application of these functions is given below. In the outcome when an integer is printed we prefix it with "i".

```
val example:Tree[Integer] = Add(mult,
        Add(add, Leaf(new Integer(4)),Leaf(new Integer(7))),
        Add(mult, Add(add,Leaf(new Integer(2)), Leaf(new Integer(1))),
         Leaf(new Integer(5))))

scala> inorder(example)
res10: List[Any] = List(i4, add, i7, mult, i2, add, i1, mult, i5)

scala> preorder(example)
res12: List[Any] = List(mult, add, i4, i7, mult, add, i2, i1, i5)

scala> evaluate(example)
res21: Integer = i165
```

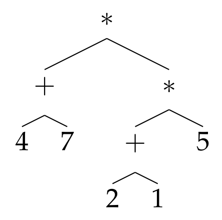

Figure 1: *Binary tree representing the expression* $(4 + 7) * ((2 + 1) * 5)$

Optional exercise.

1. Consider a redefinition of the algebraic data type so that not only integers but also some variables are allowed. Define a function evaluateWithVariables that given a tree and a list of associations (variable $\rightarrow$ value) evaluates the tree and returns its value.

Solution

First of all, the $Tree[+A]$ trait was created with 4 case objects – *EmptyTree* object to describe empty tree, *add*, *mult* and *divide* objects for algebraic operations – and 3 case classes – *Var* to define a variable, $Leaf$ to define a $tree\ leaf$ and *Add* which is used as a "*tree constructor*".

Secondly, the *Complex* class was defined to operate with complex numbers: *toString* method was overridden to print given complex number correctly; methods $+$, $-$, $*$ and $/$ were defined to implement bacis operations on complex numbers.

Then, *preorder* and *inorder* functions were implemented. Code is based on the recursive tree traversal algorithm: in *preorder* function the parent node comes first, then the left child, then the right one. As this binary tree should be represented as a $List[Any]$, the root node and the results of two recursive calls of *preorder* function are then concatenated.

The *inorder* function is based on the same algorithm, but in this case the left child comes first, then the root node, then the right child. The base case for the recursion is when the type of tree is *Var* or $Leaf$ – both options show that this node is a leaf. Pattern matching is used to map case object into *add*, *mult* or *divide* element of a list correspondingly.

Then, the *evaluate* and *evaluateWithVariables* functions were implemented. The main idea is similar to the recursive traversal of a tree. The key difference is that the algebraic operations are applied to the results of the recurrent calls instead of concatenation method. The *evaluateWithVariables* function uses the same algorithm. In addition, it uses the Map[String, A] data structure to map the *Var* to its corresponding value.

As the *go* inner functions should apply algebraic operations to values of generic type a *Num* trait was developed as well as a *NumOps* class (which adds basic algebraic operations to any object of type A that implements Num[A]) and its implementations for three types which are handled in this solution – *Integer*, *Double* and *Complex*. This approach is based on the tutorial "An Intro to Generic Numeric Programming with Spire" and is taken to solve the problem that "$+$ *operator* can not be applied to generic type A" (otherwise scala compiler raises an exception because these algebraic operations are not defined for every data type in Scala).

Finally, the examples are shown.

The case of using function *evaluateWithVariables* without passing the Map of values as a parameter is considered.

Solution is based on lectures notes, "Scala Lecture Notes" handout and "An Intro to Generic Numeric Programming with Spire" web page[1] — *algebraic data types, OOP, traits, classes, inheritance, recursion, pattern matching, lists, lazy and eager evaluation*.

[1] http://typelevel.org/blog/2013/07/07/generic-numeric-programming.html, 2016-10-07