ASSIGNMENT 2

ALEXANDER CROLL

Advanced Programming, IT712A

Data Science, University of Skövde

Lecturer: Vicenç Torra

October 20, 2016

Table of Contents

1. Introduction

1.1. Purpose of this paper

This paper is meant to accompany the practical programming assignment (“Assignment 2”) in the course Advanced Programming.

While the exercises were implemented in Eclipse and are handed in as text file (“[Advanced Programming] Alexander Croll – Assignment 2”), this document will do the following:

- Discuss the implemented solution
- How the solution was build
- Provide references, where required

2. Exercise 1

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.

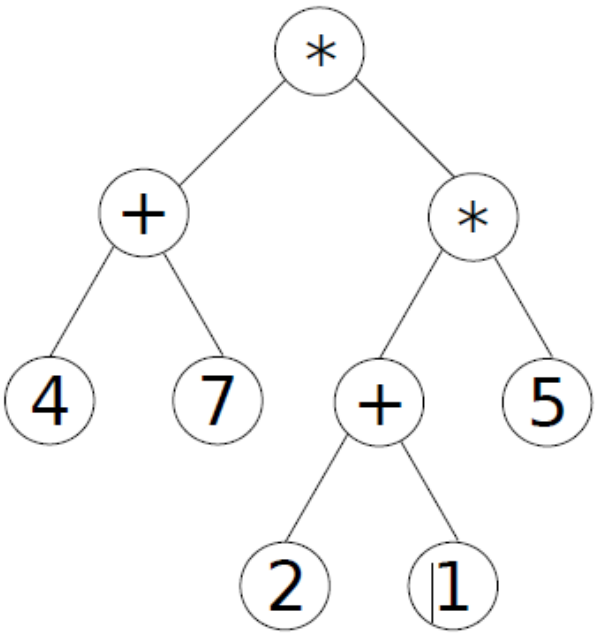

Fig. 1. Binary tree representing the expression $(4+7)*((2+1)*5)$.

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)

evaluate(example)

res21: Integer = i165

2.1. Binary Tree

The first step of the implementation is to define the binary tree as it is given in the exercise. This is done by the *trait Tree[+A]*. The arbitrary type *A* is used because the tree should not only use one data type, but numbers in general. These numbers should at least include real and complex numbers. Furthermore, the case objects *EmptyTree*, *add* and *mult* are the cases when the tree is empty, i.e. has no more branches, and the mathematical operations. As defined by the exercise, at least addition and multiplication should be possible, so these two are implemented here. One could also think about adding further operations such as division and subtraction if needed.

The case classes *Add[A]* and *Leaf[A]* are used to define instances of a binary tree. The class *Add[A]* describes a new branch of a tree, where the three parameters root, left and right stand for the top node and the two dependent sub-nodes. These dependent sub-nodes could be Leaves, as implemented in the case class *Leaf[A]*.

2.2. Data Types

The arbitrary type A already includes some real data types, e.g. Int or Double. However, a complex data type still needs to be implemented which is done by the class *Complex*. This class takes two parameters, numRe being the real number and numIm being the imaginary number. A complex number should be represented in the way of X + Y*i, which is why the .toString method has to be overridden (otherwise the output would simply be X, Y). The .toString method is overridden to achieve a representation of X + Y*i for numIm > 0 and X – Y*i for numIm < 0.

As a final step, the two required operations add and multiply have to be implemented. For complex numbers, additional is quite straight forward, but multiplying a bit more complex. The implementation has been done according to the following definitions:[1]

Addition of Complex Numbers:

$$(a+b\mathbf{i}) + (c+d\mathbf{i}) = (a+c) + (b+d)\mathbf{i}$$

Multiplication of Complex Numbers:

$$(a+b\mathbf{i})(c+d\mathbf{i}) = ac + ad\mathbf{i} + bc\mathbf{i} + bd\mathbf{i}^2 \quad | \textit{ basic rule}$$

$$(a+b\mathbf{i})(c+d\mathbf{i}) = (ac-bd) + (ad+bc)\mathbf{i} \quad | \textit{ quicker way}$$

Right now, the arbitrary type A could be one of many types, so for example a String. Because String does not have mathematical operations such as multiply or add, the type A also does not include those. However, because later on we would like to execute mathematical operations (add, mult) on our inputs (which will be numbers, but of different types), we need to implement these operations specifically for the possible data types.

Thus, as an alternative to classes, traits can help define methods and even a partial implementation of them. These methods can then be implemented in companion objects for specific instances. Here, this was done by trait DataTypes[A] and the companion objects for the real data types (Integer, Double) and the complex data type. All of these objects extend the trait DataTypes[A] and implement the functions *add* and *mult* specifically for their cases.[2] [3]

In a last step, the mathematical operations have to be assigned to their respective operators, so *add* -> + and *mult* -> *. [4] The implicit class does this for every type of DataType[A].

2.3. Function ‘preorder’

The function preorder should traverse the tree and return a list of its elements in preorder. When returning the preorder, the root is visited before the first subtree, and finally the second subtree.

The function preorder takes one input parameter, which is the Tree[A], and returns a List[Any], because there are both numbers and operations in the tree. Then, pattern matching is applied

[1] Math is fun (2016)
[2] Torra, V. (2016): pp. 61 ff.
[3] Westheide, D. (2013)
[4] Stackoverflow (2011)

on the tree, with three possible cases: Add, Leaf or Var (these are the elements used to construct a tree).

- As a tree definition always starts with Add(…), this is also the first case. In any case, the tree will need a root, which will consist of either of the two operations (add, mult). Thus, this case is again applying pattern matching to the root parameter. In either case, the output will be a List(add | mult) to which the function will then concatenate the left side of the tree and then the right side. The function calls itself recursively with preorder(left | right). When it is called again, the same pattern matching is applied as long as there is a new root.
- The goal of the function preorder is to reach the case of a Leaf. When a Leaf is reached, the value of this leaf is taken and appended to the list. As required by the exercise, in case of an Integer, there will also be an “i” added in front of the value.
- The function also has a third case Var, which is only implemented to provide exception handling in case that a tree with variables is attempted to be put in order with this function.

Example:

```
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))))

preorder(example)
```

In this sample evaluation, only the left part of the tree will be evaluated step by step due to simplicity reasons.

Evaluation		**Output**
tree match -> case Add		-
root match -> case mult		List(mult) ::: preorder(left) ::: preorder(right)
tree match -> case Add		List(mult) ::: preorder(left) ::: preorder(right)
root match -> case add		List(mult, add) ::: preorder(left) ::: preorder(right) ::: preorder(right)
tree match -> case Leaf	tree match -> case Leaf	List(mult, add, i4, i7) ::: preorder(right)

...preorder(right) would be evaluated in a similar fashion and concatenated to the list. Evaluation of left and right side of the binary tree will happen at the same time (as indicated by the split in the first column, last row).

2.4. Function ‘inorder’

The function inorder should traverse the tree and return a list of its elements in order. When returning the inorder, the first subtree is visited before the root, and finally the second subtree.

The function inorder also takes one input parameter, which is the Tree[A], and returns a List[Any], because there are both numbers and operations in the tree. Then, pattern matching is applied similarly as for the preorder function, also including three possible cases: Add, Leaf or Var.

- As a tree definition always starts with Add(…), this is also the first case. In any case, the tree will need a root, which will consist of either of the two operations (add, mult). Thus, this case is again applying pattern matching to the root parameter. In either case, the output will be a List(add | mult). In contrary to the preorder function, the inorder function does not concatenate left and right side to the List(add | mult), but instead, the left side comes first, and then List(add | mult) and the right side are concatenated to it. However, the function does also call itself recursively with inorder(left | right), applying the same pattern matching as long as there is a new root.
- The goal of the function inorder is to reach the case of a Leaf. When a Leaf is reached, the value of this leaf is taken and appended to the list. As required by the exercise, in case of an Integer, there will also be an “i” added in front of the value.
- The function also has a third case Var, which is only implemented to provide exception handling in case that a tree with variables is attempted to be put in order with this function.

Example:

```
val example2: Tree[Complex] = Add(add, Add(mult, Leaf(new Complex(2,3)), Leaf(new Complex(1,2))), Add(add, Leaf(new Complex(1,4)), Leaf(new Complex(3, 9))))

inorder(example)
```

In this sample evaluation, only the left part of the tree will be evaluated step by step due to simplicity reasons.

Evaluation		**Output**
tree match -> case Add		-
root match -> case add		inorder(left) ::: List(add) ::: inorder(right)
tree match -> case Add		inorder(left) ::: List(add) ::: inorder(right)
root match -> case add		inorder(left) ::: List(mult) ::: inorder(right) ::: List(add) ::: inorder(right)
tree match -> case Leaf	tree match -> case Leaf	List(2.0+3.0i, mult, 1.0+2.0i, add) ::: preorder(right)

...inorder(right) would be evaluated in a similar fashion and concatenated to the list. Evaluation of left and right side of the binary tree will happen at the same time (as indicated by the split in the first column, last row).

2.5. Function ‘evaluate’

The function evaluate is supposed to traverse the tree and evaluate the mathematical operations.

The function evaluate has one input parameter, Tree[A], and one output parameter, which is the result of all operations in form of a string. Within the function, there is another function, calc, that actually does the calculation of the result. This function has the same input parameter, but will return an arbitrary type A (depending on the type of tree). A first layer of pattern matching will determine on of the three cases Add(…), Leaf or Var (as already seen in the previous functions).

- In the case of Add(…), a second layer of pattern matching on the root is applied. Depending on the case (add, mult), the tree is split into left and right side and the appropriate operation is added in between (either adding or multiplying left and right side). As long as there is still a new root, the function calc is called recursively.
- The goal of the calc function (as for the previous functions) is to reach a Leaf case, which means that a value has been found that can be added or multiplied.

- The function also has a third case Var, which is only implemented to provide exception handling in case that a tree with variables is attempted to be put in order with this function.

Finally, the outcome of this calc function is checked using pattern matching for one of these two cases

- If the outcome is an Integer, then the result should be returned in the format “i” + value, so this case is adding an “i” in front of the result value.
- If the outcome is anything but an Integer, the outcome will be returned as a String by applying the .toString method.

Example:

```
val example3: Tree[Complex] = Add(add, Leaf(new Complex(3,2)), Leaf(new Complex(1,7)))
evaluate(example3)
```

Evaluation		**Output**
tree match -> case Add		-
root match -> case add		calc(left) + calc(right)
tree match -> case Leaf	tree match -> case Leaf	Complex(3, 2) + Complex (1, 7)
*(a+b**i**) + (c+d**i**) = (a+c) + (b+d)**i***		Complex(4.0, 9.0i)
calc(tree) match -> case any		any.toString
Complex(4.0, 9.0i).toString		String = 4.0 + 9.0i

3. Optional Exercise

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 → value) evaluates the tree and returns its value.

3.1. Solution

The function evaluateWithVariables was not implemented in this solution. Hence, the solution is not equipped to use variables within the tree and calculate results based on this. However, the class Var[A] was implemented as an extension of Tree[A], which allows the functions to also do pattern matching for Var[A]. In the case of a Var, the pattern matching will return an appropriate exception message stating that variables are not allowed in this context.

List of References

Math is Fun (2016): Complex Numbers, https://www.mathsisfun.com/numbers/complex-numbers.html, Date of Retrieval: October 19th, 2016

Stackoverflow (2011): Coding with Scala implicits in style, http://stackoverflow.com/questions/5984720/coding-with-scala-implicits-in-style, Date of Retrieval: October 19th, 2016

Torra, V. (2016): Scala – From a Functional Programming Perspective, Cham/Switzerland: Springer International Publishing

Westheide, D. (2013): The Neophyte's Guide to Scala, Part 12: Type Classes, http://danielwestheide.com/blog/2013/02/06/the-neophytes-guide-to-scala-part-12-type-classes.html, Date of Retrieval: October 19th, 2016