# DM509 Programming Languages

# Fall 2013 Project (Part 2)

Department of Mathematics and Computer Science
University of Southern Denmark

October 9, 2013

# Introduction

The purpose of the project for DM509 is to try in practice the use of logic and functional programming for small but non-trivial examples. The project consists of two parts. The first deals with logic programming and the second part with functional programming.

Please make sure to read this entire note before starting your work on this part of the project. Pay close attention to the sections on deadlines, deliverables, and exam rules.

# Exam Rules

This second part of the project is a part of the final exam. Both parts of the project have to be passed to pass the course.

Thus, the project must be done individually, and no cooperation is allowed beyond what is explicitly stated in this document.

## Deliverables

There is one deliverable for this first part of the project: A short project report in PDF format (2-5 pages without front page and appendix) has to be delivered. This report should contain the following 7 sections:

- front page
- specification
- design
- implementation
- testing
- conclusion
- appendix including all source code

The report has to be delivered as a single PDF file electronically using Blackboard's SDU Assignment functionality.

## Deadline

Wednesday October 30, 2013, 12:00

# Restrictions

Use of the following is **forbidden** for this project:

- The keyword **`var`**
- Anything from the package **`scala.collection.mutable`**

Please make sure that you do not directly use any of the above in your solution as it leads to non-functional programming. If you violate this rule, your project will not pass.

# The Problem

Your task in this part of the project is to produce a purely functional implementation of a Huffman encoder in `Scala`. A Huffman encoding is an optimal single-character *prefix encoding*, i.e. no codeword is a prefix of some other codeword and each character is mapped to exactly one keyword.

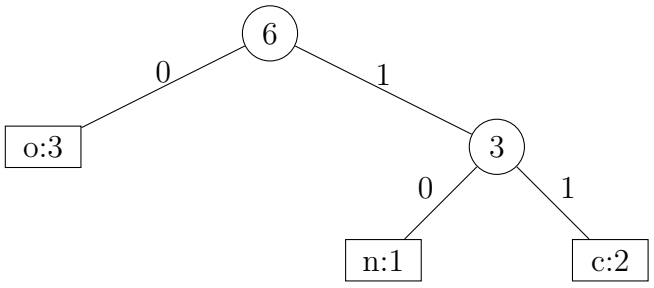
An optimal Huffman code tree is given by a *full* binary tree in which all internal nodes have 2 children. Given an alphabet, $C$, the tree has $|C|$ leaves and exactly $|C| - 1$ internal nodes. For a Huffman Tree, $T$, denote the frequency of occurrence of a character $c$, $f(c)$ and the depth of $c$'s leaf (equivalent to its codeword length) by $d_T(c)$. The bit length of the entire encoding is then

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Leaves in a Huffman Tree are defined as tuples of characters and frequencies and internal nodes as 3-tuples of a weight along with left and right subtrees.

A very minimal example of a Huffman tree is depicted in Figure 1:

Figure 1: The Huffman Tree created from the string "cocoon"



### Encoding

The encoding of a single character is given by the path from the root to the leaf containing that character, i.e. each time we go right, we get a 1, and each time we go left, we get a 0. From Figure 1 we get the following code-table:

$$\{o : 0, n : 10, c : 11\}$$

### Decoding

Decoding a string of bits is done by reading one character at a time and traversing the Huffman Tree going right whenever we see a 1 and left whenever we see a 0. When reaching a leaf, we output the character on that leaf and restart at the root of our tree. Given the bit-string `1, 1, 0, 1, 1, 0, 0, 1, 0` and the tree from Figure 1, we get the word "cocoon".

### Constructing the Tree

To construct the Huffman tree we use an optimal greedy algorithm. The original *imperative* construction algorithm relies on the mutation of a priority queue, constructing the Huffman tree from a list of frequency leaves as depicted in Algorithm 1.

**Algorithm 1** createCodeTree($PQ$)

```
1: while size(PQ) > 1 do
2:     left ←ExtractMin(PQ)          ▷ Mutates state
3:     right ←ExtractMin(PQ)         ▷ Mutates state
4:     tree ←MakeTree(left,right)
5:     Insert(PQ,tree)               ▷ Mutates state
6: end while
7: return ExtractMin(PQ)             ▷ Mutates state
```

Basically, we extract the two minimal elements from $PQ$, combine them and add them back into $PQ$. Assuming an alphabet of size $n$ and insertion and extraction in $\Theta(\log n)$, this algorithm runs in $\Theta(n \log n)$ time. There are two things about this algorithm that do not fit in the functional programming paradigm:

1. It relies on an imperative looping construct (the *while loop*)
2. state is mutated.

To combat these problems we use two tools, namely recursion and immutability. For this project, you must implement a functional version of the above algorithm. A sketch of a functional version of `createCodeTree` could look as follows:

```
createCodeTree(PQ) =
  if PQ has only 1 element
    head of PQ
  else
    next <- combination of 2 smallest elements from PQ
    newq <- drop 2 from PQ and insert 'next'
    createCodeTree( newq )
```

Whenever $PQ$ contains only a single element, that element is a Huffman Tree. One step in the loop can be modeled by first taking 2 minimal elements from an immutable priority queue, merging them and adding them to a version of the priority queue without the original two elements. With the above knowledge, you can model a recursive version with no mutation of state.

# The Implementation

There are a number of data structures in the standard `Scala` API that are of interest to the solution of this project. The following lists some immutable data structures along with interesting methods:

- `List[T]`
  - `xs ++ ys`, (append xs and ys)
  - `xs groupBy (λ)`, (create a map from $\lambda(x) \rightarrow x$)
  - `xs reduce (λ)`, (reduce list using binary associative operator $\lambda$)
  - `xs flatMap (λ)`, equivalent to a `map` followed by a `flatten`
- `Map[K,V]`
  - `a ++ b`, (combine maps a and b)
  - `Map('n' -> List(1,0))`, (construct an immutable hash map with (key,value) pair `'n'` to `List(1,0)`)
- `SortedSet[T]`

- `take i`, (get the first `i` elements in $\Theta(i \cdot \log n)$ time)
- `drop i`, (drop the first `i` elements in $\Theta(i \cdot \log n)$ time)
- `s + e`, (insert element `e` in `s` in $\Theta(\log n)$ time)

For this project, we use `Scala`'s `SortedSet[T]` in place of a priority queue since the standard library contains no immutable version of the priority queue. We see that `take` and `drop` can be used as a generalization of `ExtractMin` that insert and remove some number of elements respectively. Furthermore, `+` can be used in place of `Insert`. The following example shows some simple use of `SortedSet[Int]` from the REPL:

```
scala> val s = SortedSet[Int]() + 100 + 10 + 40 + 33 + 77
s: scala.collection.immutable.SortedSet[Int] = TreeSet(10, 33,
   40, 77, 100)

scala> s drop 2
res9: scala.collection.immutable.SortedSet[Int] = TreeSet(40,
   77, 100)

scala> s take 2
res10: scala.collection.immutable.SortedSet[Int] = TreeSet(10,
   33)
```

# The Input

The interface to your program will consist of 3 methods:

- `def createCodeTree(chars: List[Char]): HuffTree = ???`
- `def encode(tree: HuffTree)(text: List[Char]): List[Bit] = ???`
- `def decode(tree: HuffTree)(bits: List[Bit]): List[Char] = ???`

It is your responsibility to test these methods along with the ones you implemented "behind" the interface. To test your program, repeatedly implement and test smaller parts (essentially methods) in order to produce the final working program. Debugging your program will become increasingly difficult if you fail to test your methods thoroughly in a hierarchical fashion.

Your report should **as a minimum** list some thorough tests proving the functionality of the three interface methods above.

# The Output

Given the interface methods described in the last section, you should be able to infer input and output formats for the different methods. It is your responsibility to convince the reader that your program works as intended in order to pass the project. A very minimal example of a test case follows.

**Input:**

```
val str = "cocoon".toList
val tree = createCodeTree(str)
val enc = encode(tree)(str)
val equal = ( str == decode(tree)(enc) )

println(str)
println(tree)
println(enc)
println(equal)
```

**Output:**

```
List(c, o, c, o, o, n)
Node(Leaf(o,3),Node(Leaf(n,1),Leaf(c,2),3),6)
List(1, 1, 0, 1, 1, 0, 0, 1, 0)
true
```

# The Task

Using the template implementation given in the three files, `HuffTree.scala`, `Huffman.scala`, and `HuffTest.scala`, implement and test Huffman encoding and decoding in a purely functional style. You are allowed to implement additional methods and change the template, however, the 3 interface methods `createCodeTree`, `encode`, and `decode` should keep their function signatures.

1. Basics

   - `def weight(tree: HuffTree): Int`, calculates the sum of the weights of all leaves in `tree`
   - `def combine(left: HuffTree, right: HuffTree): HuffTree`, combines 2 Huffman Trees by creating a new internal node and

making them left and right subtrees respectively. The weight of the new node is the combined weight of its children.

2. Generating the Huffman Tree

   - `def frequencies(chars: List[Char]): List[(Char, Int)]`, converts a list of characters to a frequency table with tuples of $(c, f(c))$
   - `def combineTrees(trees: SortedSet[HuffTree]): HuffTree`, creates the Huffman Tree by repeatedly combining the 2 smallest elements (leftmost) from `trees` and reinserting the result into `trees` *without* its 2 smallest elements.
   - `def createCodeTree(chars: List[Char]): HuffTree`, combines the functionality of the other functions to create a Huffman Tree from a list of characters.

3. Encoding and Decoding

   - `def convert(tree: HuffTree): CodeTable`, converts a Huffman Tree into a `CodeTable`, i.e. an immutable `Map[Char,List[Bit]]`.
   - `def codeBits(table: CodeTable)(char: Char): List[Bit]`, looks up the list of bits associated with `char` in `table`
   - `def encode(table: CodeTable)(text: List[Char]): List[Bit]`, encodes `text` using a code table
   - `def getLetter(tree: HuffTree, bits: List[Bit]): (Char, List[Bit])`, given an encoded string, returns the first character along with the remaining bits of the encoded string.
   - `def decode(tree: HuffTree)(bits: List[Bit]): List[Char]`, decodes a string of bits with a given Huffman Tree using `getLetter`

To start working on this project, it is a good idea to use the `fsc` command ("Fast" Scala Compiler). To compile the 3 template files and run the `HuffTests` program, you can proceed as follows:

```
felix@felix-UX32VD:~$ fsc Huffman.scala HuffTests.scala HuffTree.scala
felix@felix-UX32VD:~$ scala HuffTests
java.lang.RuntimeException: Method not implemented...
at Huffman$.$qmark$qmark$qmark(Huffman.scala:7)
at Huffman$.createCodeTree(Huffman.scala:42)
...
```

Remember to recompile your program whenever you make changes. You are of course free to work with the implementation of this project with whatever tools you like best.

Finally, produce a report containing descriptions of your design decisions, implementation and tests as stated in the section on deliverables.

# Optional Tasks

1. **Make it Object Oriented**

   If you are feeling particularly adventurous, there are things we can do to make the program more `Scala`-esque. We have already looked at 3 data structures (usually, *collections* in `scala`) from the standard library, namely `List[T]`, `Map[K,V]`, and `SortedSet[T]`. Since `Scala` is a functional `object-oriented` language, these are implemented as classes. Thus, for all interactions with these types we have used *inifix notation*, i.e. the left-hand operand has been the *callee*. Our data type, `HuffTree`, is already a class-hierarchy divided between two subclasses, `Node` and `Leaf`. We could, however, move our functionality inside the HuffTree class itself to make it fit inside the paradigms of `Scala`.

   An example of the change could be the construction of a `HuffTree` from a list of chars:

```
val str = "cocoon".toList
val tree = Huffman.createCodeTree(str)
```

   Here, we use a method from a singleton object, `Huffman.createCodeTree(str)`. This could be changed to:

```
val str = "cocoon".toList
val tree = HuffTree(str)
```

   which is much more to the point. Furthermore, encoding and decoding can now be defined in the context of `tree` as:

```
val enc = tree encode str
val equal = (str == tree decode enc)
```

   The optonal task is to create an object oriented version (leave your original solution intact!) of your solution in which you move the functionality of `Huffman.scala` into methods on `HuffTree, Node, and Leaf`. Think carefully about ways to avoid code-duplication, e.g. what methods can you implement at the *bottom* of the class hierarchy inside `HuffTree`? How does the use of your program change with the object oriented implementation? What are the advantages/disadvantages of the object oriented solution?

2. **Make it Useful**

   Encoding strings with the type *Int* is hopelessly inefficient in terms of memory consumption. If we say that an `Int` is 32 bits and we use it to store 1 bit, we are wasting significant amounts of memory. Alternatively, we can model bits as `Boolean` to reduce runtime memory consumption.

   Luckily, it is not too hard to embed bits in, say, an integer. If we want to embed bit-strings of length 4, we can use the following code:

```
import math.pow

val N = 4

def toMaskedInt(xs:List[Boolean]) = {
  val z = (xs,0 until N).zipped
  val ints = z.map( (b,i) => if(b) pow(2,i).toInt else 0
     )
  ints.reduce(_ | _) //Reduce with logical 'or'
}

def fromMaskedInt(n:Int) = 0 until N map (i => (n &
   pow(2,i).toInt) != 0) toList


val bools = List(true,false,false,true)
println( fromMaskedInt(toMaskedInt(xs)) == xs )
```

   One could do a similar transformation into `Bytes`. The method `sliding` on lists could then come in handy:

```
scala> List(1,1,0,0,1,0,1,0).sliding(4,4).mkString(",")
res57: String = List(1, 1, 0, 0),List(1, 0, 1, 0)
```

   Try to make a useful command-line application that can compress/decompress files using your Huffman Encoding. Test your implementation and compare the compressed file size to the original.