Lab Exam Software Specification and Testing

October 19, 2015

```
> module LabExam where
>
> import Data.List
> import Data.Ord
> import Test.QuickCheck
```

There are 6 problems plus a bonus problem. If you manage to solve the bonus problem you earn compensation for errors in your answers to the regular problems.

Problem 1

Codes are lists of zeros and ones. For convenience we use the datatype `[Int]` for codes.

```
> type Code = [Int]
```

Examples are `[1,0]` and `[0,1]` and `[1,0,1,1]`. Think of these as lists of bits.

Now assume that these codes represent non-negative integers in binary, with the bits in reverse order of significance: least significant bit first. Thus `[1,0]` represents 1, `[0,1]` represents 2, and `[1,0,1,1]` represents 13.

Write translation functions `bits2integer :: [Int] -> Integer` and `integer2bits :: Integer -> [Int]` that efficiently translate bit lists to the corresponding (non-negative) integers, and back.

Next, test your code for correctness with QuickCheck (the two translations have to be inverses of each other) and for efficiency (the code has to be fast; you have to devise your own method for establishing this).

Problem 2

A function $f : \mathbb{R} \to \mathbb{R}$ (a function on the real numbers) is called *even* if it holds for all $x \in \mathbb{R}$ that $f(x) = f(-x)$. An example of an even function is the squaring function (the function $\lambda x \mapsto x^2$).

Please note that this notion of *even* is different from the well-known property of integer numbers. We are talking about a property of *functions* here.

- Give one more example of an even function on the real numbers.
- Explain why it is impossible to write an algorithm that checks whether a function of type `Float -> Float` is even.
- Implement a test `maybeEven` for evenness of functions, and explain why the outcome `False` of this

test is more reliable than the outcome `True`.

maybeEven :: (Float -> Float) -> Bool maybeEven f = ...

Problem 3

Give code for constructing a frequency list from a string. The input is a string, the output is a list of pairs (c, i) where c is a character and i gives the number of occurrences of c in the input string. The type is:

```
freqList :: String -> [(Char,Int)]
freqList s = [] -- replace by your own code
```

This should give, e.g.:

```
*LabExam> freqList "Lab Exam"
[(' ',1),('E',1),('L',1),('a',2),('b',1),('m',1),('x',1)]
```

Include a test for checking that your implementation is correct.

Problem 4

A *Huffman tree* is a binary tree with characters at the leaf nodes, and weight information (given by an integer) at every node.

```
> data HTree  = Leaf Char Int
>             | Fork HTree HTree Int
>             deriving (Show)
```

The weight of a tree is given by:

```
> weight :: HTree -> Int
> weight (Leaf _ w)    = w
> weight (Fork _ _ w)  = w
```

Call the following property the *Huffman property*:

```
> prop_huffman :: HTree -> Bool
> prop_huffman (Leaf _ _) = True
> prop_huffman (Fork t1 t2 w) = prop_huffman t1 && prop_huffman t2
>                        && weight t1 + weight t2 == w
```

The following tree merge function can be used to build Huffman trees.

```
> merge t1 t2 = Fork t1 t2 (weight t1 + weight t2)
```

Consider the following function that builds trees from frequency tables:

```
> createTree :: [(Char,Int)] -> HTree
> createTree [(c,i)] = Leaf c i
> createTree ((c,i):t) = merge (Leaf c i) (createTree t)
```

Test with QuickCheck whether for any frequency list `xs` it is the case that `createTree xs` has the Huffman property.

Problem 5

Huffman code is used in compression algorithms. It gives efficient encodings of characters as bit strings, in the sense that more frequent characters get mapped to shorter bit strings. Huffman trees are built from frequency lists, as follows:

```
> buildTree :: [(Char, Int)] -> HTree
> buildTree table = let
>     sortedTable = sortBy (comparing snd) table
>     trees = map (\ (c,i) -> Leaf c i) sortedTable
>     bld :: [HTree] -> HTree
>     bld ([t])     = t
>     bld (a:b:cs) = bld $
>         insertBy (comparing weight) (merge a b) cs
>   in bld trees
```

Explain how this algorithm works by inspection of the code.

Next, test with QuickCheck whether all Huffman trees built by means of `buildTree` have the Huffman property?

Problem 6

```
> type CodeTable = [(Char,Code)]
```

Building a code table from a Huffman tree:

```
> tree2table :: HTree -> CodeTable
> tree2table (Leaf c _)     = [(c, [])]
> tree2table (Fork l r _)  =
>   map (addBit 0) (tree2table l) ++ map (addBit 1) (tree2table r)
>      where addBit b = \ (c,xs) -> (c,b:xs)
```

Building a code table from a string: first use the string to build a frequency list, next map that to a Huffman tree, next use the Huffman tree for constructing the table.

```
> string2table :: String -> CodeTable
> string2table = tree2table . buildTree . freqList
```

This uses your code for `freqList` from an earlier question. Example:

```
 *LabExam> string2table "the quick brown fox jumps over the lazy dog"
 [('s',[0,0,0,0,0]),('v',[0,0,0,0,1]),('p',[0,0,0,1,0]),('q',[0,0,0,1,1]),
  ('y',[0,0,1,0,0]),('z',[0,0,1,0,1]),('w',[0,0,1,1,0]),('x',[0,0,1,1,1]),
  ('i',[0,1,0,0,0]),('j',[0,1,0,0,1]),('f',[0,1,0,1,0]),('g',[0,1,0,1,1]),
  ('m',[0,1,1,0,0]),('n',[0,1,1,0,1]),('k',[0,1,1,1,0]),('l',[0,1,1,1,1]),
  ('o',[1,0,0]),('e',[1,0,1,0]),('t',[1,0,1,1,0]),('u',[1,0,1,1,1]),
  ('h',[1,1,0,0,0]),('r',[1,1,0,0,1]),('c',[1,1,0,1,0,0]),
  ('d',[1,1,0,1,0,1]),('a',[1,1,0,1,1,0]),('b',[1,1,0,1,1,1]),
  (' ',[1,1,1])]
```

Notice that the codes for 'o' and ' ' (space) are the shortest, corresponding to the fact that these are the characters that occur most often in the text.

Encode a string using a code table:

```
> ecd :: CodeTable -> String -> Code
> ecd m = concatMap f where
>    f x = maybe (myundefined x) id $ lookup x m
>    myundefined x = (error $ "No '"++[x]++"' in the code table")
```

The codetable is computed from a string, so we can view encoding as a function that takes a first string for the frequencies, and uses these to encode a second string:

```
> encode :: String -> String -> Code
> encode fqs message = ecd (string2table fqs) message
```

Decode using a frequency string:

```
> decode :: String -> Code -> String
> decode s = dcd tree
>     where  tree = buildTree $ freqList s
>            dcd :: HTree -> Code -> String
>            dcd (Leaf c _) []         = [c]
>            dcd (Leaf c _) bs         = c : dcd tree bs
>            dcd (Fork l r _) (b:bs)   = dcd (if b == 0 then l else r) bs
```

Your task is to test this code for correctness with QuickCheck. Some care should be taken to ensure that the Huffman tree has weight information for all characters. Also, you may assume that the string that is used to compute the frequency table is non-trivial: it should assign weights to at least two characters.

Deliverables: some suitable QuickCheck properties, plus a test report. If your testing reveals an error, please indicate how it could be corrected. If your testing indicates that the code is correct, please explain why you are confident that you did run enough tests.

Bonus Problem

[If you manage to solve this bonus problem you earn compensation for errors in your answers to the regular problems above.]

Once you have convinced yourself that all code you developed and tested in this exam session is correct and fast, you can use it to compress texts for which you have a frequency table.

A frequency table for English you can create by taking some sample text from a corpus. Take for instance, the source code of this Lab Exam, or some fragment from the ASCII version of the novel *Lord Jim* by *Joseph Conrad*. Next, use the frequency table you have computed to test your compression software. Count each character in the text you are compressing as 8 bits. Next count the numbers of bits in the Huffman encoding, and compute the compression factor you have achieved.

To make this into a practical method, you should find a way to represent bit lists in a compact way. Check if you can use `bits2integer` for this. But maybe you need hexadecimal representation. Have a look at

ByteString Bits and Pieces for some inspiration.

Implement your method, compute the compression factor you have achieved, and test whether your software works correctly by decompressing and comparing with the original input. Can you use QuickCheck for this?
