Functional and Logic Programming

Home Assignment 4

Due: Saturday, 31.5.2018 - 23:55

Instructions

- Please create a source file called ***HW4.hs*** and put all the answers there.
  The file should start with a comment which contains your **full name** (in English) and **ID**
  (see also example: ***hwexample.hs*** in the Moodle)

```
-- John Doe
-- 654321987
```

- Make sure the file **is valid** by loading it into GHCi.
  A valid file will load without any errors or warnings.
- If you need a function but you don't know how to implement it - just write it's signature (name and type) and put `undefined` in the function's body.
  That way you'll be able to load the file even though it contains references to undefined names.
  (see also example: ***hwexample.hs*** in the Moodle)
- When writing a function - write both the **type** and the **body** of the function.
- Be sure to write functions with **exactly the specified name** (and **type signature** - if it is provided) for each exercise.
  You may create additional auxiliary/helper functions with whatever names and type signatures you wish.
- Try to write **small functions** which perform just **a single task**, and then **combine** them to create more complex functions.

Exercises

In this exercise we'll implement the **Huffman coding** compression algorithm.
The algorithm converts a string into a sequence of bits which take up much less space (in bits) than the original string.

`https://en.wikipedia.org/wiki/Huffman_coding`
`https://www.youtube.com/watch?v=umTbivyJoiI`

We'll use the following data types:

```
data Bit = O | I
    deriving (Eq, Show)

data HuffmanTree = Leaf Char Int | Node Int HuffmanTree HuffmanTree
    deriving (Eq, Show)
```

And the following type synonyms:

```
type Code = [Bit]

type FreqTable = [(Char, Int)]

type Dictionary = [(Char, Code)]

type Encoder = Char -> Code
```

The final goal is to create 2 functions:

```
encode :: String -> (HuffmanTree, Code)
```

which takes a string and returns a pair which contains:

1. An Huffman tree which could be used to decode compressed codes.
2. A compressed code which represents the string.

```
decode :: HuffmanTree -> Code -> String
```

which takes an Huffman tree and a compressed code and translates it back into a string by using the tree as a decoder.

1. **Create frequencies table:**

   a) Create a function **`insert`** with the following signature:

   ```
   insert :: Char -> FreqTable -> FreqTable
   ```

   which takes a character and a frequencies table and adds `1` to the item in the frequencies table which corresponds to this character, if it exists; otherwise - it adds a new item to the frequencies table with this character in the 1st component and `1` in the 2nd component. You may assume there is only 1 item in the list which corresponds to the given character.

   ```
   insert 'x' [('a',5),('x',3),('b',6)] = [('a',5),('x',4),('b',6)]
   insert 'x' [('a',5),('b',6)]         = [('a',5),('b',6),('x',1)]
   insert 'x' []                        = [('x',1)]
   ```

   b) Create a function **`count`** with the following signature:

   ```
   count :: String -> FreqTable
   ```

   The function will take a list and return a frequencies table where for each element of the list there is a pair in which the first component is this element and the second component is the number of times this element occurrs in the list.
   The order of the items in the list doesn't matter, but for each character in the input string there should be only 1 item within the table that counts the number of occurrences of that character in the string.
   (*hint:* you can use the **`insert`** function you've just created)

   ```
   count "this is a test" = [('t',3),('h',1),('i',2),('s',3),(' ',3),('a',1),('e',1)]
   ```

2. **Convert a frequencies table into a list of Huffman leaves:**
   Create a function **`initLeaves`** with the following signature:

   ```
   initLeaves :: FreqTable -> [HuffmanTree]
   ```

   The function will initialize a list of Huffman trees by taking a frequencies table and turning each pair of (character, frequency) into a leaf of an Huffman tree.

   ```
   initLeaves (count "this is a test") =
       [Leaf 't' 3, Leaf 'h' 1, Leaf 'i' 2, Leaf 's' 3, Leaf ' ' 3, Leaf 'a' 1, Leaf 'e' 1]
   ```

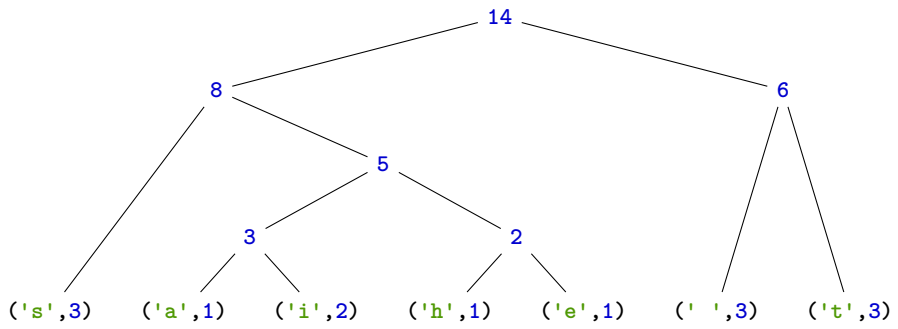
3. **Build an Huffman tree from a list of Huffman leaves:**
   Create a function **`buildTree`** with the following signature:

   ```
   buildTree :: [HuffmanTree] -> HuffmanTree
   ```

   The function will take a list of Huffman trees and recursively merge them into a single Huffman tree by picking at each step the 2 trees which have the smallest value within their root and replace them with a `Node` which contains the sum of their values and has both of them as

children.
The order of the leaves doesn't matter, but at the end result leaves with small numeric values (which correspond to rare characters) should have a long path from the root, and leaves with large numeric values (which correspond to common characters) should have a short path from the root.
You may assume the input list is non-empty and contains at least 2 leaves.

```
buildTree (initLeaves (count "this is a test")) =
    Node 14
        (Node 8
            (Leaf 's' 3)
            (Node 5
                (Node 3 (Leaf 'a' 1) (Leaf 'i' 2))
                (Node 2 (Leaf 'h' 1) (Leaf 'e' 1))
            )
        )
        (Node 6 (Leaf ' ' 3) (Leaf 't' 3)
        )
```



4. **Create dictionary from an Huffman tree:**
Create a function:

`createDictionary :: HuffmanTree -> Dictionary`

which converts a `HuffmanTree` into a `Dictionary`.
The `Dictionary` will hold for each character which appears in the leaves of the tree the path from the root to that leaf.
The path will be given as a list of `Bit`s - `O` for a left child, and `I` for a right child.
(*hint:* see the **`paths`** function in class exercise 5)

```
createDictionary (buildTree (initLeaves (count "this is a test"))) =
    [('s',[O,O]),('a',[O,I,O,O]),('i',[O,I,O,I]),('h',[O,I,I,O]),('e',[O,I,I,I]),(' ',[I,O]),('t',[I,I])]

createDictionary (buildTree (initLeaves (count "xoxo"))) =
    [('x',[O]),('o',[I])]
```

```
createDictionary (buildTree (initLeaves (count "aaaabc")))
    [('b',[0,0]),('c',[0,1]),('a',[1])]
```

5. **Create encoder from dictionary:**
   Create a function:

   `createEncoder :: Dictionary -> Encoder`

   which turns a dictionary into an encoder.
   The encoder is a function which takes a **Char** as input, and if it has a corresponding code in the **Dictionary** it will returns that code; otherwise - it will return the empty list.

```
createEncoder (createDictionary (buildTree (initLeaves (count "this is a test")))) 'e' = [0,1,1,1]
createEncoder (createDictionary (buildTree (initLeaves (count "this is a test")))) 'q' = []
```

6. **Encode:**
   Create a function:

   `encode :: String -> (HuffmanTree, Code)`

   which when given a **String** returns both the **HuffmanTree** derived from it and the **Code** which represents it.
   You may assume the string contains at least 2 different characters.
   (*hint:* use all the functions you have created so far, and maybe some other helper functions)

```
encode "this is a test" =
    ( Node 14
        (Node 8
            (Leaf 's' 3)
            (Node 5
                (Node 3 (Leaf 'a' 1) (Leaf 'i' 2))
                (Node 2 (Leaf 'h' 1) (Leaf 'e' 1))
            )
        )
        (Node 6 (Leaf ' ' 3) (Leaf 't' 3))
    , [1,1,0,1,1,0,0,1,0,1,0,0,1,0,0,1,0,1,0,0,1,0,0,1,0,0,1,0,1,1,0,1,1,1,0,0,1,1]
    )

encode "beekeeper" =
    ( Node 9
        (Node 4
            (Node 2 (Leaf 'p' 1) (Leaf 'k' 1))
            (Node 2 (Leaf 'b' 1) (Leaf 'r' 1)))
        (Leaf 'e' 5)
    , [0,1,0,1,1,0,0,1,1,1,0,0,0,1,0,1,1]
    )
```

7. **Decode:**
   Create a function:

   ```
   decode :: HuffmanTree -> Code -> String
   ```

   which takes a `HuffmanTree` and a `Code` and decodes by using the `HuffmanTree` the `Code` into a `String`.

   It should obey the following law:

   ```
   uncurry decode (encode s) == s
   ```

   ```
   decode (Node 9
       (Node 4 (Node 2 (Leaf 'p' 1) (Leaf 'k' 1)) (Node 2 (Leaf 'b' 1) (Leaf 'r' 1)))
       (Leaf 'e' 5))
       [0,1,0,1,1,0,0,1,1,1,0,0,0,1,0,1,1]
       =
       "beekeeper"
   ```