Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming

WS 19/20

Benjamin Dietrich, Denis Hirn

Assignment #3

Submission Deadline: Thu, 14.11.2019

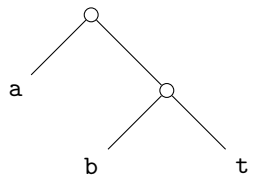
Please note the following format guidelines for all your future submissions:

- The only acceptable file format for your solutions is **plain text Haskell files** (`*.hs`). Unless explicitly stated differently, files of other formats – in particular PDF, DOCX, etc. – will not be graded in future.
- All textual answers shall be written as **Haskell comments**: `--...`, `{-...-}`
- Please try to submit **one file for each exercise**.
- **All code you submit must compile.** Code that does not compile – in particular, does not typecheck – might not be graded in future.
- Please submit comprehensive code that is nicely and consistently formatted and well-documented. To have an idea of "nicely formatted code", you can find a short *Haskell* style guide here: https://github.com/tibbe/haskell-style-guide/blob/master/haskell-style.md

Exercise 1: Encoding and Decoding of Huffman Codes **(10 Points)**

Every second, billion of electronic messages are transmitted. To find a compact binary coding for such messages is a major task in order to save network traffic.

This exercise is about *Huffman Codes*, a classic approach to find an optimal compact translation of messages (sequences of characters) to *bit* sequences. Messages can be *encoded* and *decoded* using binary trees. Take for example the following tree:



The tree represents a coding for the three characters `'a'`, `'b'` and `'t'`. Each character can be described by the by the *path* from the tree's root to the leaf with the character. This path is recorded by the decision to go either left (`L`) or right (`R`) at each node of the path:

Character	*Code*
'a'	L
'b'	RL
't'	RR

We *encode* a message by concatenation of its characters' path-codes. For example, message `"abba"` is encoded to the bit sequence `LRLRLL`. In our program, this will be represented by a list `[L,R,L,R,L,L] :: [Bit]`. `L` and `R` are constructors of a simple sum type `Bit`, defined as follows:

```
data Bit = L | R
           deriving (Eq, Show)
```

In the opposite direction, *decoding* also proceeds along the coding given by the tree. Assume that we receive an encoded bit sequence `RLLRRRRLRR`. In order to decode it, we follow the given path until we find a leaf (i.e. a character) – which is `'b'` in this example. Restarting from the root, we repeat this procedure, to get a textual interpretation for the rest of all bits. Finally, we have a decoded cleartext message `"battat"`.

1. Define an algebraic data type `HuffTree` to represent trees which can be used for Huffman coding.
2. Write a function

   ```
   encodeMessage :: HuffTree -> String -> [Bit]
   ```

   which encodes a message of type `String` using the coding of a given tree of type `HuffTree`.
3. Also write a function

   ```
   decodeMessage :: HuffTree -> [Bit] -> String
   ```

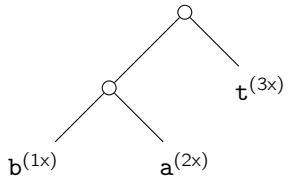
   to decode a given bit-sequence.
4. In the lecture, we've discussed the identity `(fromList . toList) xs == xs`. A similar identity holds regarding the functions `encodeMessage` and `decodeMessage`. Formulate that identity.

Exercise 2: Optimal Huffman Trees **(10 Points)**

In the previous exercise we encoded and decoded messages along given tress (of type `HuffTree`). A further interesting question is how to construct an *optimal* coding (i.e. tree) for a given cleartext message. A coding is *optimal* with respect to a message `m`, if the encoded bit sequence `encodeMessage m` is of minimal length.

The solution is to consider the *frequency* of each character in the cleartext message. Based on that it is possible to build a tree – called *Huffman tree* – which contains the most frequent character next to the root and all other characters, in descending frequency order, further down in the tree. Thus, more frequent characters need less space for coding, while less frequent need more.

With this approach the following Huffman tree is derived for the cleartext word `"battat"` (character frequency annotated in in parentheses):



Encoding `"battat"` with this tree returns the bis sequence `LLLRRRLRR` which is one bit shorter than the sequence given in the previous exercise.

The tree is built in two steps:

1. Write a function `frequencies` witch counts the frequency of each character in a given cleartext message and returns an association list of character↦frequency tuples, sorted[1] in increasing frequency order:

   ```
   frequencies :: String -> [(Char, Integer)]
   ```

   Example: `frequencies "battat"` ≡ `[('b',1),('a',2),('t',3)]`.

2. Write a function `codeOf` witch builds the Huffman tree for a given message. Built the tree bottom-up out of subtrees: Start with one tree for each character (so far, each tree consists only of one leaf). Take the two trees with the lowest frequency and merge them into a new tree. The frequency of a tree is the sum of the frequencies of his subtrees. Repeat the last step until all trees are merged into one – the Huffman tree.

   ```
   codeOf :: String -> HuffTree
   ```

[1] Have a look at function `sortOn :: Ord b => (a -> b) -> [a] -> [a]` in `Data.List` to sort the list based on the second value of each tuple (i.e. the frequency).