



Assignment 3

Hand in this assignment until **Friday, 19. May 2023, 10:00** at the latest.

Exam-style Exercises

Exercises marked with **E** are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Running out of ideas?

Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the [#forum](#) channel on our Discord server and check the tag for this assignment—maybe you'll find just the help you need.

Task 1: Algebraic Data Types (Sum & Product types) **E**

(1 credit)

Use **data** declarations to define algebraic data types **T** such that the number of possible values of type **T** is

- (a) 1
- (b) 7
- (c) 1200
- (d) 79.

Examples: Type **data Bool = True | False** has two values while type **Maybe Bool** has three possible values.

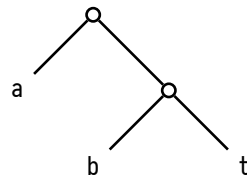
Hint

Define auxiliary algebraic data types that help you to define the types **T** above.

Task 2: Encoding and Decoding of Huffman Codes

(1 credit)

Billions of electronic messages are sent every second. Finding a compact binary coding for such messages is a major task to save network traffic. This exercise is about *Huffman Codes*, a classic approach to finding an optimal compact translation of messages (sequences of characters) into *bit* sequences. Messages can be *encoded* and *decoded* using binary trees. For example, consider the following tree:



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

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

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

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

Conversely, *decoding* also follows the coding given by the tree. Assume that we receive an encoded bit sequence RLLRRRLRR. 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 the bits. Finally, we have a decoded message "battat".

- (a) Define an algebraic data type `HuffTree` to represent trees which can be used for Huffman coding.

Hint

Your `HuffTree` data type will have *two* constructors.

- (b) Write a function

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

which encodes a message of type `String` using the coding of a given tree of type `HuffTree`. Recall: `String` is a type synonym for `[Char]`.

- (c) Also write a function

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

to decode a given bit-sequence.

- (d) 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.

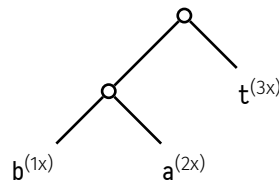
Task 3: Optimal Huffman Trees

(1 credit)

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

The solution is to consider the *frequency* of each character in the plain text message. Based on this, 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 map to shorter bit sequences, while less frequent characters require more.

With this approach the following Huffman tree is derived for the plain text word **"battat"** (the frequency of characters is noted in brackets):



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

The tree is built in two steps:

- (a) Write a function `frequencies` that counts the frequency of each character in a given plain text message and returns an association list of character→frequency tuples, sorted¹ in order of increasing frequency:

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

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

- (b) Write a function `codeOf` that builds the Huffman tree for a given message. Build the tree bottom-up from subtrees: Start with one tree for each character (so far, each tree has only 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 its subtrees. Repeat the last step until all trees are merged into one—the Huffman tree.

```
1 | codeOf :: String -> HuffTree
```

¹Have a look at function `sortBy :: 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).