EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summer Term 2025
Prof. Torsten Grust, Björn Bamberg
WSI — Database Systems Research Group

Assignment 3

Hand in this assignment until **Tuesday, May 27, 12pm** at the latest.

🤔 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—maybe you'll find just the help you need.

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

Task 1: Minesweeper (List Processing)

The objective of Minesweeper is to clear a rectangular field which contains hidden ***mines***. For each field cell the player steps on, a hint is given about the number of mines in the direct neighborhood.

We want to implement a function to compute these numbers for a given field with visible mines:

```haskell
minesweep :: [[Char]] -> [[Int]]
```

Example:

```haskell
field = [ "        "
        , "***     "
        , "* *     "
        , "***     "
        ]

Main> minesweep field
[ [ 2, 3, 2, 1, 0, 0, 0, 0 ]
, [ 3, 5, 3, 2, 0, 0, 0, 0 ]
, [ 5, 8, 5, 3, 0, 0, 0, 0 ]
, [ 3, 5, 3, 2, 0, 0, 0, 0 ]
]
```

Follow these steps to solve the problem:

(a). The actual algorithm shall be formulated as a combination of helper functions, you have to implement in advance:

   i. Write a function `num :: Char -> Int` which takes a field cell and returns 1, if it is mined (`'*'`), and 0, if it is safe (`' '`).

   ii. Write a function `shiftL :: a -> [a] -> [a]`. `shiftL x xs` shifts all elements of a list `xs` to the left, the original head is dropped, the new rightmost element is `x`.

   Note
   The length of list `xs` before and after shifting is identical.

   iii. Write a function `shiftR :: a -> [a] -> [a]` which shifts a given list of numbers to the right.

   iv. Write a function

   ```haskell
   zipWith3' :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
   ```

It takes a function `f :: (a -> b -> c -> d)`, as well as three lists of identical length and returns a list where each element is the combination (using `f`) of the input lists' elements at the same position.[1] **Example:**

```
zipWith3' (\a b c -> a * b + c) [1,2] [3,4] [5,6] ≡ [8,14]
```

v. Use functions `shiftL`, `shiftR` and `zipWith3'` to implement a function

```
addNeighbours :: [Int] -> [Int]
```

For each element of a given list of integers, the function returns the sum of the element itself, together with its left and right neighbour. The first (last) element has no left (right) neighbour. **Example:**

```
addNeighbours [1,1,1,0,1] ≡ [2,3,2,2,1]
```

vi. Write a function which transposes the rows and columns of a given matrix (list of lists of identical length):[2]

```
transpose :: [[a]] -> [[a]]
```

Example:

```
transpose [[1,2,3],[4,5,6]] ≡ [[1,4],[2,5],[3,6]]
```

(b). Implement `minesweep` as a combination of `Prelude` function `map` and the helper functions `num`, `addNeighbours` and `transpose` defined in step (a).

Task 2: Algebraic Data Types (Sum & Product Types) Ⓔ

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.

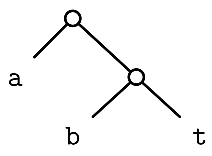
💡 Hint
Define auxiliary algebraic data types that help you to define the types `T` above.

[1]Obviously you shall implement this function on your own and must not use the `Prelude` function `zipWith3`.
[2]You must not use the `Data.List` function `transpose`, but implement the function on your own.

Task 3: Encoding and Decoding of Huffman Codes

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:

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

Conversely, *decoding* also follows 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 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

```haskell
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

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

to decode a given bit-sequence.

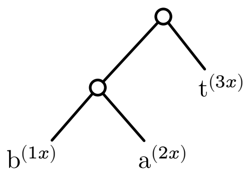
D In the lecture, we've discussed the identity `(fromList . toList) xs == xs`. A similar identity holds regarding the functions `encodeMessage` and `decodeMessage`. Formulate that identity.

Task 4: Optimal Huffman Trees

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 bis sequence `LLLRRRLRR`, 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 $\mapsto$ frequency tuples, sorted[3] in order of increasing frequency:

```
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.

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

[3]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).