Declaratieve Talen

Haskell 3

1 Tree Folds

1.1 Defining a tree

Given below is a definition for a binary tree. Be sure to include the `deriving (Show, Eq)` construct to generate the right typeclasses.

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
  deriving (Show, Eq)
```

- Just like lists we can define a fold over trees. Define a function `foldTree :: (a -> b) -> (b -> b -> b) -> (Tree a -> b)` that performs this folding.
- If given the right functions, `foldTree` can reconstruct the original tree. Define a function `idTree :: Tree a -> Tree a` that performs this reconstruction.

1.2 Folding trees

Using `foldTree`, define the following functions.

- A function `nrOfLeaves :: Tree a -> Int` that counts the number of leaves in a tree.
- A function `sumTree :: Tree Int -> Int` that sums the integers stored at the leafs of a tree.
- A function `depthOfTree :: Tree a -> Int` that calculates the maximum depth of the tree.
- A function `treeToList :: Tree a -> [a]` that converts a tree to a list.
- A function `minTree :: Tree Int -> Int` that returns the smallest integer stored at any of the leaves.
- A function `mirrorTree :: Tree a -> Tree a` that mirrors all subtrees.
- A function `addOne :: Tree a -> Tree Int` that adds one to each integer stored at the leaves of a tree.

Examples

```
Main> ( idTree (Leaf 1) )
Leaf 1
Main> ( idTree (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))) )
Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))

Main> ( nrOfLeaves (Leaf 1) )
1
Main> ( nrOfLeaves (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))) )
3

Main> ( sumTree (Leaf 1) )
1
Main> ( sumTree (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))) )
6

Main> ( depthOfTree (Leaf 1) )
1
Main> ( depthOfTree (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))) )
3

Main> ( treeToList (Leaf 1) )
[1]
Main> ( treeToList (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))) )
[1,2,3]

Main> ( minTree (Leaf 1) )
1
Main> ( minTree (Fork (Fork (Leaf 20) (Leaf 30)) (Leaf 10) ) )
10

Main> ( mirrorTree (Leaf 1) )
Leaf 1
Main> ( mirrorTree (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))) )
Fork (Fork (Leaf 3) (Leaf 2)) (Leaf 1)

Main> ( addOne (Leaf 1) )
Leaf 2
Main> ( addOne (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))) )
Fork (Leaf 2) (Fork (Leaf 3) (Leaf 4))
```

2 Squares and Triangles

In this exercise you will need to use IO to print a rectangles, squares, trapezoids and triangles.

Rectangles & Squares Implement a function `rectangle` that, given a width and a height prints a rectangle composed of `*` characters with the given dimensions.
For example:

```
> rectangle 4 2
****
****
> rectangle 10 5
**********
**********
**********
**********
**********
```

Also implement a function `square` that takes a single dimension and creates a square with edges of the given dimension.
For example (note: because the characters on a terminal are rectangular, the image on the screen will not be exactly square).

```
> square 2
**
**
> square 5
*****
*****
*****
*****
*****
```

Trapezoids & Triangles Implement a function `trapezoid` that, given the width w of the top edge and the height h prints a trapezoid composed of `*` characters with a top edge of length w, a height h, and a bottom edge parallel to the bottom edge of lenght $w + 2(h - 1)$. The midpoints of top and bottom edges must line up.
For example:

```
> trapezoid 4 3
  ****
 ******
********
```

Note: There should be no spaces on the right of the stars!

Also implement a function `triangle` that, given a height h prints an equilateral triangle with base $2(h-1)+1$.

For example:

```
> triangle 1
*
> triangle 2
 *
***
> triangle 4
   *
  ***
 *****
*******
```

Expense Balancer

In this assignment we solve a common problem: Imagine you are going on a trip with your friends. Naturally, this involves some expenses, e.g. air plane tickets, hotels, bars, restaurants, ... To simplify matters, you decide that each person takes care of a specific expense.

However, some expenses are larger than others, so not everyone will have spent an equal amount of money. In this exercise, we write a program that computes how much everyone should pay to everyone afterwards, to ensure that everyone contributes an equal amount.

Part I: Expense & Delta

An *Expense* is an amount of money that has been spent by a person. For implementing the balancing algorithm it is useful to know a person's *relative* expense. This is the difference between a person's expense and the average expenses (total expenses divided by the number of persons), we call this a *Delta*.

Exercise 1

1. Create a data type `Expense` that contains an amount of money (a `Double`) and the name of a person (a `String`), that made the expense Ensure that `Expense` also derives `Eq` and `Ord`.

   Important: `Expense` must contain these fields in the given order to ensure that the `Ord` instance is correct, i.e. it sorts first by amount, and only then by name.

2. Implement the function `mkExpense :: String -> Double -> Expense` that creates an `Expense` with the given name and amount.

3. Additionally, implement an instance of `Show` for `Expense`, that shows an expense as a name, followed by a colon (:), a space and an amount.

```
> mkExpense "Alex" 10.0
Alex: 10.0
> mkExpense "Alex" 200.05
Alex: 200.05
> mkExpense "Xander" 10 < mkExpense "Alex" 200.05
True
```

Exercise 2

1. Create a data type `Delta` that contains an `Expense`. The `Delta` data type is a wrapper around `Expense`, to indicate that we have switched from absolute expenses to relative expenses (with respect to the total average expenses). This means that negative amounts are also possible. Ensure that this data type also derives `Eq` and `Ord`.

2. Additionally, implement an instance of `Show` for `Delta`, that shows a `Delta` as a name, followed by a colon (:) a space and an amount.

3. Implement the function `fromExpense :: Double -> Expense -> Delta` that transforms an absolute a `Expense` into a relative `Delta`, given the total average expenses, by computing the difference of the absolute amount with the average expenses.

To improve the readability of the examples, a function `mkDelta` has been predefined as:

```
mkDelta name amount = fromExpense 0 (mkExpense name amount)
```

```
> fromExpense 0 (mkExpense "Alex" 10.0)
Alex: 10.0
> fromExpense 250.05 (mkExpense "Alex" 200.05)
Alex: -50.0
> mkDelta "Xander" 10 < mkDelta "Alex" 200.05
True
```

Exercise 3 Write a function `toDeltas :: [Expense] -> [Delta]` that takes a list of absolute expenses (`[Expense]`) and transforms it into a list of relative expenses (`[Delta]`). Assume that there is at most one `Expense` per person. For every `Expense` in the list, the amount in the corresponding `Delta` is the difference between the amount in the `Expense` and the average expenses (the sum of all absolute expenses divided by the number of people). This means that a person who has spent more than average has a positive `Delta`, and a person who has spent less than average has a negative `Delta`. The sum of all deltas should be zero.

Hint: You can use the function `fromIntegral` to convert an `Int` to a `Double`.

```
> toDeltas [mkExpense "Alex" 40, mkExpense "Gert-Jan" 200]
[Alex: -80.0,Gert-Jan: 80.0]
> toDeltas [mkExpense "Matthias" 11.5, mkExpense "Thomas" 100]
[Matthias: -44.25,Thomas: 44.25]
> toDeltas [mkExpense "Alex" 11.5, mkExpense "Gert-Jan" 100, mkExpense "Tom" 1000]
[Alex: -359.0,Gert-Jan: -270.5,Tom: 629.5]
```

Part II: Transferable Transfers

A *transfer* transfers money from one person to another person.

Note, this means that if person 1 transfers money to person 2, the expense of person 1 increases, and the expense of person 2 decreases. The same reasoning applies for deltas: transferring money from person 1 to person 2 increases the delta of person 1, and decreases delta of person 2.

The data type `Transfer` contains three fields: two `String`s: the payer and the payee, and a `Double`: the amount of money that is transferred. This data type is already defined for you (including a `Show` instance).

This part of the assignment requires you to implement instances of the `Transferable` class for `Expense` and `Delta`. The class has one method: `applyTransfer :: Transfer -> t -> t` which applies a `Transfer` to a `Transferable t`.

Exercise 4 Complete the `Transferable` instance for `Expense` such that `applyTransfer t e` increases or decreases the expense `e` with the right amount if the owner of `e` is the payer or the payee, respectively. Otherwise, `e` is left unchanged. If the payer and payee are identical, `e` must also be unchanged.

```
> applyTransfer (MkTransfer "Alex" "Tom" 100) (mkExpense "Alex" 125)
Alex: 225.0
> applyTransfer (MkTransfer "Alex" "Tom" 100) (mkExpense "Tom" 125)
Tom: 25.0
> applyTransfer (MkTransfer "Alex" "Tom" 100) (mkExpense "Thomas" 125)
Thomas: 125.0
> applyTransfer (MkTransfer "Tom" "Tom" 100) (mkExpense "Tom" 125)
Tom: 125.0
```

Exercise 5 Complete the `Transferable` instance for `Delta` such that `applyTransfer t d` increases or decreases the delta `d` with the right amount if the owner of `d` is the payer or the payee, respectively. Otherwise, `d` is left unchanged. If the payer and payee are identical, `d` must also be unchanged.

```
> applyTransfer (MkTransfer "Alex" "Tom" 100) (mkDelta "Alex" 125)
Alex: 225.0
> applyTransfer (MkTransfer "Alex" "Tom" 100) (mkDelta "Tom" 125)
Tom: 25.0
> applyTransfer (MkTransfer "Alex" "Tom" 100) (mkDelta  "Thomas" 125)
Thomas: 125.0
> applyTransfer (MkTransfer "Tom" "Tom" 100) (mkDelta  "Tom" 125)
Tom: 125.0
```

Exercise 6 Define a function `createTransfer :: Double -> Delta -> Delta -> Transfer` such that `createTransfer amount d1 d2` creates a `Transfer` from the owner of `d1` to the owner of `d2` with the given amount.

```
> createTransfer 100 (mkDelta "Alex" (-100)) (mkDelta "Tom" 200)
Alex -> Tom:100.0
> createTransfer 30 (mkDelta "Tom" (-100)) (mkDelta "Tom" 200)
Tom -> Tom:30.0
```

Part III: Balancing Expenses

In this part you write the functions to balance the expenses. Given a list of `Expense`s, the goal is to obtain a list of `Transfer`s, such that when they are applied to those `Expense`s, the resulting expenses are balanced. Two expenses are balanced if the absolute difference of their amounts is smaller than a small $\varepsilon > 0$.

More formally, we say that a list of expenses[1] $E \subseteq \mathbb{R}$ is ε-*balanced* if

$$\forall e_1, e_2 \in E : |e_1 - e_2| < \varepsilon$$

[1] For this definition you should understand "expenses" as an amount of money.

We use this definition since floating-point numbers are not infinitely precise, and amounts smaller than one euro cent are irrelevant in practice.

Exercise 7 Implement a function `balanced :: [Expense] -> Double -> Bool` such that `balanced exp e` returns `True` if and only the list `exps` is `e`-balanced.

Hint: You can use `abs` from the `Prelude`.

```
> balanced [mkExpense "Alex" 100,mkExpense "Matthias" 125.0] 0.01
False
> balanced [mkExpense "Alex" 100,mkExpense "Matthias" 100] 0.01
True
> balanced [mkExpense "Alex" 100.5,mkExpense "Matthias" 100] 1
True
```

Extra: The straightforward implementation of `balanced` that is immediately derivable from the definition requires $\mathcal{O}(N^2)$ time. When you have finished the other exercises, try to implement a version that has a linear time complexity ($\mathcal{O}(N)$).

Exercise 8 Implement the function `balanceDeltas :: [Delta] -> Double -> [Transfer]` that ε-balances a list of deltas (the definition of ε-balanced for `Delta` is identical to the definition for `Expense`).

`Hint`: Implement a greedy algorithm that in every step, tries to transfer from the smallest delta (the person with the least expenses) to the person with the largest delta (the person with the most expenses). The transferred amount *must not be larger than* the absolute values of smallest and the largest delta (transferring larger amounts does not balance the expenses).

`Hint`: To find the minimum and maximum you can use `minimum` and `maximum` from the `Prelude`.

```
> balanceDeltas [mkDelta "Alex" (-175),
                 mkDelta "Gert-Jan" (-275),
                 mkDelta "Tom" 425,
                 mkDelta "Thomas" 25] 0.01
[Gert-Jan -> Tom:275.0,Alex -> Tom:150.0,Thomas -> Tom:25.0]
```

Exercise 9 Implement a function `balance :: [Expense] -> Double -> [Transfer]` that ε-balances a list of `Expense`s (first transform the list into `Delta`s and then use `balanceDeltas`).

```
> let expenses = [mkExpense "Alex" 200,mkExpense "Gert-Jan" 40,
                  mkExpense "Tom" 1000,mkExpense "Thomas" 275.5]
> balance expenses 0.01
[Gert-Jan -> Tom:338.875,Alex -> Tom:178.875,Thomas -> Tom:103.375]
> -- the following commands only work right after the previous command
> -- applyTransfers has been predefined
> map (applyTransfers it) expenses
[Alex: 378.875,Gert-Jan: 378.875,Tom: 378.875,Thomas: 378.875]
> balanced it 0.01
True
```

Part IV: Balancer Application

This part contains a small terminal application that reads a number of expenses, balances them, and then prints the resulting transfers.

Exercise 10 Implement a function `getExpenses :: IO [Expense]` that asks the user to input expenses, and returns these as a list. The function first asks for a name and an amount. The `name` is preceded by a `Name:`-prompt, and the amount by an `Amount:`-prompt. If the amount is non-negative, the function asks for another name and amount, repeating until the entered amount becomes negative.

`Hint`: Implement a function `getExpense :: IO Expense` that reads a single `Expense`. The function `getExpenses` calls this function, and if the amount is non-negative, it places the `Expense` in a list, and calls *itself* again.

```
> getExpenses
Name: Alex
Amount: 200
Name: Gert-Jan
Amount: 0
Name:
Amount: -1
[Alex: 200.0, "Gert-Jan": 0.0]
```

Exercise 11 Implement a function `printTransfers :: [Transfer] -> IO ()` that prints a list of transfers, where every transfer appears on a separate line.

```
> printTransfers [MkTransfer "Alex" "Tom" 200,MkTransfer "Thomas" "Gert-Jan" 20]
Alex -> Tom:200.0
Thomas -> Gert-Jan:20.0
```

Exercise 12 Implement a function `balanceIO` that asks a user to input expenses, then balances these expenses and finally prints the transfers that balance these expenses (you may assume $\varepsilon = 0.01$). Leave a blank line between where the expenses are input, and the transfers are printed.

```
> balanceIO
Name: Alex
Amount: 200
Name: Gert-Jan
Amount: 40
Name: Tom
Amount: 1000
Name: Thomas
Amount: 0
Name:
```

```
Amount: -1

Thomas -> Tom:310.0
Gert-Jan -> Tom:270.0
Alex -> Tom:110.0
```

Good luck!