1 Your own types

It's time to start making your own types, here are some ideas for some types you can make, but you are welcome to try something else if you want. Remember to go read in LYH if you forgot something.

1.1 Shapes

Go through the shapes example in LYH.

A shape is either a cirle or a rectangle.

- Circles are defined by x and y coordinates and a radius.
- Rectangles are defined by their lower left corner and upper right corner.
- Update so points are represented by their own type.
- Make functions that calculate the surface area of shapes.
- Make functions that move the shapes around.
- Write at least one more function that has at least one argument of type `Shape`. E.g.:
- `boundingBox :: Shape -> Shape` that given a circle returns the smallest rectangle that contains it.
- `center :: Shape -> Point` that given a shape returns its center point.
- `moveToOrigin :: Shape -> Shape` that given a shape, centers it on the origin.
- `scale :: Shape -> Float -> Shape` that given a shape and a float, scales it by the float. Optional: make it keep its center (potentially hard).
- Optional: Expand your shape type with a square type, defined by the lower left corner and the side-length.

1.2 Students

A student has: name, exam number, list of enrolled courses, list of grades.

- Make the value constructor.
- Make functions that can add or remove courses from the enrolled list.
- Make a function to add a grade.
  - Optional: Only allow grading for courses already enrolled.

1.2.1 Persons

Rather than students directly, make a type for persons.

A person has a name, gender and age.

- Make gender a type similar to `Bool`, choose whatever genders you like.
- Replace name with person in student, and update the program accordingly.
- Make some functions for persons, e.g. greetings based on name, gender and/or age.

- Update the constructors for students and persons with record syntax
- Write at least one additional function that has students and/or persons as an argument. E.g.:
  - Given a list of persons, return a list of persons with voting rights.
  - Given a list of students, return a list of students with at least some average grade.
    * Make courses into a type as well, where courses have a name and a weight. Update the average to take the weight into account.

1.3 Trees

A tree is a recursive datastructure. A tree can either be the empty tree `Nil` (or `EmptyTree` from LYH), or be a `Node` and have a key and two sub-trees (it is a binary tree). To make it a binary search tree, everything in the left sub-tree must have a key smaller or equal to the key in the current node, and everything in the right sub-tree must be strictly larger (this is different from LYH, where duplicates are removed).

- Make the value constructors.
- Make a function that takes a value and a tree, and inserts a node with the value as key into the tree.
- Make a function that takes a value and a tree, and returns whether that value is the key of a node in the tree.
- Make a function `takeT` that takes a value `h` (height) and a tree, and returns the first `h` layers of the tree.
- Make a function `repeatT` that takes a value, and creates an infinite tree where the value is the key of every node.
- Make a function `replicateT` that takes two values `k` and `h`, and creates a tree with `h` layers where all nodes have `k` as key.

We can make fold for trees like so:

```
data Tree a = Nil | Node a (Tree a) (Tree a)

foldT _ z Nil = z
foldT f z (Node k l r) = f k (foldT f z l) (foldT f z r)
```

- Write the type signature for `foldT`. Try to understand why the function was defined this way.
- Define `f` such that `sizeT = foldT f 0` returns the number of nodes in the tree.
- Define `f` such that `sumT = foldT f 0` returns the sum of nodes in the tree.
- Define `f` such that `heightT = foldT f 0` returns the height of the tree.
- Define `f` such that `flattenT = foldT f []` returns the tree as a sorted list.

2 Playing games

Here are some suggestions for games you can make, more or less with what you know at this point.

2.1 1D Lights out

Lights out is a game where you want to "turn off" all the lights, but when you toggle one light, you also toggle the neighbouring lights.

A simple version of this works in just one dimension, and could be represented with a list of `Bool`.

You can implement it however you want, but I think of it something like this:

```
ghci> Lights [False,True,False]
Lights [False,True,False]
ghci> toggle (Lights [False,True,False]) 3
Lights [False,False,True]
```

2.2 Tic-tac-toe

Tic-tac-toe is a bit larger of a project. I recommend looking up `Data.Array`, as a help to representing the board.

Once you are far enough I also recommend making a custom implementation of `show`, rather than just deriving it.

In my implementation I had the player pieces as a type with three possible values, where one was a "dummy" piece if no piece was at that spot. It could also be implemented with `Maybe` instead.

3 Your own typeclass

Try to make your own typeclass, if you need inspiration you can try to make a `CharLike`, `Game` or `PlayerSet` typeclass.

3.1 CharLike

Instances of `CharLike` should be convertible into a `Char`, e.g. if `Bool` is an instance, then `True` could be converted into `'T'` and `False` into `'F'`.

Make some instances of `CharLike`. As a start make `Char` an instance.

Add a function `similar` to the typeclass, that should see if the character representation of two `CharLike` things is the same.

3.2 Game

A very simple `Game` typeclass might look something like this:

```
class Game g where
  isEnd :: g -> bool
```

To make the typeclass more useful you should probably add more functions, e.g. `move` or `getPlayers`. Note that you may have to restructure your previous game(s) if you want it/them to fulfill the requirements of your new typeclass.

3.3 PlayerSet

A `PlayerSet` typeclass may look like this.

```
class PlayerSet s where
  players :: s a -> [a]
```

Some instances could be `PlayerList`, `PlayerTuple` and `SinglePlayer`, that you must define before you can define the instances.

4 Functors

Try to make some instances of the `Functor` typeclass. Here are some suggestions:

- `MyMaybe`, where you make your own implementation of `Maybe` and then make it an instance of `Functor`.
- `Tree`, following the example in book, both for making the tree and making it an instance of `Functor`.
- `Box`, a dummy type that simply holds a single value of "unknown" type.
- `PlayerList`, `PlayerTuple` or `SinglePlayer` from the previous exercise.

Now use `fmap` with a function that changes the type of what you implemented. For example, if you have `MyMaybe True`, that has type `MyMaybe Bool`, use `fmap` with a function so it changes its type into `MyMaybe Char`.