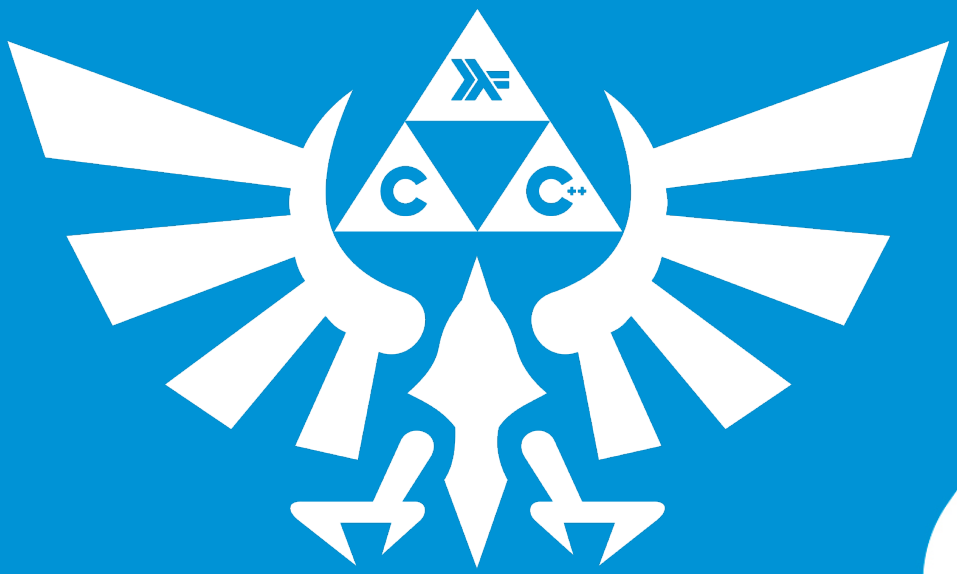
{EPITECH}

DAY 03

A GENTLE INTRODUCTION TO FUNCTIONAL PROGRAMMING



DAY 03

binary name: Game.hs, Tree.hs
language: haskell
compilation: via Makefile, including re, clean and fclean rules

- ✓ The totality of your source files, except all useless files (binary, temp files, obj files,…), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.

Today's goal is to discover how to:

- ✓ create custom types
- ✓ declare data structures
- ✓ handle algebraic data types

Step 0 - Yet another RPG

You've been asked to implement the part of a role playing video-game handling hostile non-playing characters (Mobs).

Some of these mobs are able to hold an item in their hands. There are 3 different items and they are defined as:

- ✓ Sword
- ✓ Bow
- ✓ MagicWand

Task 01

Write the data structure to describe these items. The name of the data-structure must be Item, and each item can be created using its name:

Terminal

```
*Game> :t Sword
Sword :: Item
*Game> :t Bow
Bow :: Item
*Game> :i Item
[...]
```

Task 02

Item must be deriving from **Eq**, so you should be able to compare two items.

Terminal

```
*Game> Sword == Bow
False
*Game> Sword == Sword
True
*Game> Sword /= MagicWand
True
```

Task 03

Now we want items to be displayed in a specific manner when we print them. To achieve this we want **Item** to be an instance of **Show** . Provide a custom implementation to achieve this behavior:

Terminal

```
*Game> Sword
sword
*Game> Bow
bow
*Game> MagicWand
magic wand
```

Step 1 - the Mobs

Now that we have well defined items, we're ready to implement the mobs.

Task 4

There are 3 kinds of mobs in this game: **Mummy**, **Skeleton** and **Witch**

- ✓ All the mummies come bare handed.
- ✓ The skeleton always holds an item.
- ✓ A witch can either **Just** hold an item or **Nothing**.

Create a data structure to represent them.

There's not a lot of ways to do that properly…

Your data structure **Mob** must be deriving from **Eq** and **Show**

Task 5

But some also come in different varieties, therefore you must implement the following functions to create each specific type of mob:

```
createMummy :: Mob      -- a Mummy
createArcher :: Mob     -- a Skeleton holding a Bow
createKnight :: Mob     -- a Skeleton holding a Sword
createWitch :: Mob      -- a Witch holding Nothing
createSorceress :: Mob -- a Witch holding a MagicWand
```

Task 6

We also want a single function to create any kind of mob depending on a string. As this function can fail if the string doesn't correspond to any mob, it returns a Maybe Mob.

```
create :: String -> Maybe Mob
```

The string coresponding to each type of mob are respectively:

- ✓ "mummy"
- ✓ "doomed archer"
- ✓ "dead knight"
- ✓ "witch"
- ✓ "sorceress"

Task 7

We also want to be able to equip Skeletons and Witch with any Item, replacing the one they already hold. As mummies can't hold anything, this function can also fail…

```
equip :: Item -> Mob -> Maybe Mob
```

Task 8

As with items, we want to provide a custom way to print mobs.

If you provide a custom implementation you have to remove **Show** from the list of type classes your data type "automaticaly" derives from…

- ✓ a Mummy must be shown as "mummy"
- ✓ a Skeleton holding a Bow must be shown as "doomed archer"
- ✓ a Skeleton holding a Sword must be shown as "dead knight"
- ✓ a Skeleton holding anything else as "skeleton holding a " followed by the said item
- ✓ a Witch holding nothing must be shown as "witch"
- ✓ a Witch holding a MagicWand must be shown as "sorceress"
- ✓ and a Witch holding another item as "witch holding a " followed by the said item.

Task 9

Now it is time to create our own type class. We want to have a way to know if a Mob, NPCs, Player, or any other character in the game currently holds an Item or not.

To achieve this you have to declare a new type class called **HasItem**.

This type class has two functions:

- ✓ getItem which takes an object as argument and returns a Maybe Item.
- ✓ hasItem which takes an object as argument and returns a Bool.

The minimal definition for this class is to provide an implementation for getItem, a generic hasItem implementation must be provided by the type class definition.

Task 10

Make **Mob** an instance of **HasItem**

Step 2 - Binary Tree

In this last part, we're going to do something quite diffrent and more generic: you're going to create you own "container" data structure which will be a simple binary tree.

For these exercices, create a new source file called **Tree.hs**

Task 11

Define a new data type called **Tree** which takes a type **"a"** :

```
data Tree a = ...
```

A Tree can either be constructed with **Empty** as a constructor, or with the constructor **Node** which contains, in order, another Tree of the same type, a value of type **a**, and a last Tree. your Tree must be an instance of **Show**

At this point you should be able to create trees *"by hand"* such as:

Terminal

```
> Empty
Empty
> Node Empty 42 Empty
Node Empty 42 Empty
> Node (Node Empty 31 Empty) 42 (Node Empty 53 Empty)
Node (Node Empty 31 Empty) 42 (Node Empty 53 Empty)
```

You should have a look on the definition of **Maybe** or lists...

Task 12

We now want a simple function to add a new value value in an existing tree. To add a new value in the right place in our tree, we need to be able to compare the values between each other, so for this function **“a”** must be an instance of **Ord**.

```
addInTree :: Ord a => a -> Tree a -> Tree a
```

This function takes a value, a tree, and returns a new tree with this value added in the right spot.

The rules to add a new value are as follow:

- ✓ If the tree is **Empty**, create a **Node** containing the value, with **Empty** as its right and left branch.
- ✓ If the tree is a **Node** :
  - if the value to be inserted is stricly lower than the value stored in this **Node**, the value must be added to its left branch.
  - if it’s greater or equal, the value must be added to its right branch.

Example:

Terminal

```
> addInTree 42 Empty
Node Empty 42 Empty
> addInTree 31 $ addInTree 42 Empty
Node (Node Empty 31 Empty) 42 Empty
> addInTree 53 $ addInTree 31 $ addInTree 42 Empty
Node (Node Empty 31 Empty) 42 (Node Empty 53 Empty)
```

Task 13

We want to be able to apply a function on all the values stored in our **Tree**. To achieve this, make your tree an instance of the **Functor** type class.

Terminal

```
> fmap (*2) $ addInTree 53 $ addInTree 31 $ addInTree 42 Empty
Node (Node Empty 62 Empty) 84 (Node Empty 106 Empty)
```

Task 14

We want to be able to convert a list of values to a **Tree**. Write the function **listToTree** which takes a list of **“a”** as argument and returns a Tree build using addInTree.

Task 15

Likewise, we want to be able to concert a **Tree** to a list. Write a function **treeToList** which takes a tree of **“a”** as argument and returns a **[a]**.

Task 16

Using your existing function, write the function treeSort, which takes a list of **“a”** and returns a sorted list of **“a”**.

Task 17

Make your **Tree** an instance of **Foldable**.

It should look like generalised version of a function you have alread written

Once you’ve done it, you should be able to re-write some functions in a shorter manner

v 1.1