Exercise 10: Polymorphism in Haskell

Starter code

Download these files from Quercus files.

- `Exercises/ex10/Ex10.hs`

Submission instructions

Submit the files `Ex10.hs` to MarkUs. But before you submit, check that:

- Your code does not have any syntax error. If you only completed part of the exercise, replace any possibly syntactically incorrect code with the starter code.
- Do not modify the module export code in your Haskell file.

Task 1: Practice with Maybe [5pt]

Implement the following functions according to their type signatures. Note that each of the first four has only one "reasonable" implementation that can typecheck. Part of your thought process here should be to determine what each function does, and explain it briefly *in English*!

1. `mapMaybes :: (a -> b) -> [Maybe a] -> [Maybe b]`
2. `composeMaybe :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)`
3. `foldMaybe :: (b -> a -> Maybe b) -> b -> [a] -> Maybe b`
4. `applyBinaryMaybe :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c`.
5. `collectMaybes :: [Maybe a] -> Maybe [a]`. Return a `Just` if the original list contains *only* `Just` values, and no `Nothing`s. This is probably the hardest one, and we strongly recommend using `applyBinaryMaybe` and/or writing helper function(s) with good type signatures here!

This question assumes that you have a good understanding of type signatures. If the type signatures don't make sense, start by reviewing week 4 materials.

Task 2: The Eq and Functor Typeclass [5pt]

In this task, you'll gain some practice with the `Eq` and `Functor` typeclasses. In particular, we'll work with `Functor` in a more abstract setting. Here's the (simplified) interface for the `Eq` and `Functor` typeclasses.

```haskell
class Eq a where
  -- A generalization of list `==`
  (==) :: a -> a -> Bool

class Functor t where
  -- A generalization of list `map`
  fmap :: (a -> b) -> f a -> f b
```

Complete the following tasks:

1. Make the data type `Pet` an instance of the Eq type class, so that two pets are equal if they are the same animal and have the same name. The two animals don't have to have the same age. All ants are the same.
2. Write code so that the type constructor `Organization` is an instance of the type class `Functor`, where a given function would be mapped over every `p` in the `Organization`.
3. Consider the function `robotize`, which replaces a `Person` with a `Robot`. This function is written for you. Use a call to `fmap` to turn the example value `company` (defined in the starter code) into an organization with the same structure, but populated entirely by robots. Save the result in the variable called `robotCompany`.
4. Write the generalization of the `mapMaybes` function that would work for any functors, not just `Maybe`. The type signature for this function is: 'mapF :: Functor f => (a -> b) -> [f a] -> [f b]