Functional Programming / Funkcinis programavimas

Exercise set 6 (optional)

Solutions to be sent until January 10th

Exercise 1. With the provided function and values, use `(<$>)` or `fmap` from Functor, `(<*>)` and `pure` from the Applicative type class to fill in missing bits of the broken code to make it work:

- `const Just "Hello" "World"`
- `(,,,) Just 90 <*> Just 10 Just "Tierness" [1, 2, 3]`

After "fixing", the first expression must evaluate to `Just "Hello"`, and the second expression – `Just (90,10,"Tierness",[1,2,3])` .

Exercise 2. Redefine the apply operator `(<*>)` of Applicative relying on Monad functions. In other words, define your own function of the type

```
::  (Monad f) => f (a->b) -> f a -> f b
```

using monad operations, which works exactly as `(<*>)` of Applicative. Provide two versions of the defined function, with and without the do notation.

Test your definitions on selected concrete structures, e.g., lists or Maybe values.

Exercise 3. Define functions for "lifting" a binary function to become applicable on given applicative functor or monad structures. To be exact, write functions

- `myLiftA ::  (Applicative f) => (a -> b -> c) -> (f a -> f b -> f c)`, relying on Functor and Applicative operations;
- `myLiftM ::  (Monad f) => (a -> b -> c) -> (f a -> f b -> f c)`, relying on the do notation for monads.

Modify the last solution to define a function:

```
myLiftM' ::  (Monad f) => (a -> b -> f c) -> (f a -> f b -> f c)
```

with a function parameter that introduces a monad structure as its result.

Test your definitions on selected concrete structures, e.g., lists or Maybe values.