Exercises 3

```haskell
module E3 where

import Data.List
```

Exercise 3.1: `IO` and `read`

Consider Hello World 2.0 from the lecture:

```haskell
dialogue :: IO ()
dialogue = do
  putStrLn "Hello! Who are you?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
```

Extend this implementation such that it behaves as follows.

```
E3> dialogue
Hello! Who are you?
Bob -- user input
Nice to meet you, Bob!
How old are you?
94 -- user input
Ah, that is 6 years younger than me!
```

Hint: You might want a line like `let age = read ageString :: Int` within the `do` block.

Exercise 3.2: `Functor`

Look up the definition and the laws for `Functor` and `Applicative`. You can consult the slides of lecture 3 or the Typeclassopedia. You can also use "`:i Functor`" etc. in ghci to see the definition of any type class, but note that this does not show the laws.

Recall the definition of `UnOrdPair` from Exercises 2:

```haskell
newtype UnOrdPair a = UOP (a,a)
```

Make unordered tuples a functor:

```haskell
instance Functor UnOrdPair where
  fmap = undefined
```

Then prove (on paper, or as comments here) that your definition fulfills the two functor laws:

```
fmap id = id
fmap (f.g) == fmap f . fmap g
```

Similarly, look up `Applicative` and define the following instance:

```haskell
instance Applicative UnOrdPair where
  pure  = undefined
  (<*>) = undefined
```

Check that your definition fulfills this property:

```
fmap f x = pure f <*> x
```

Then prove that your definition fulfills the four applicative laws:

```
pure id <*> == id
pure (.) <*> f <*> g <*> x = f <*> (g <*> x)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

Obvious bonus question: Can you make `UnOrdPair` a Monad?

Exercise 3.3: Hilbert's Hotel

Let's implement the famous Hilbert Hotel with laziness in Haskell.

If you don't know it yet, watch https://youtu.be/Uj3_KqkI9Zo.

A room can be occupied by a guest (`Just "Jana"`) or empty (`Nothing`). A hotel is a list of rooms:

```
type Guest = String
type Room = Maybe Guest
newtype Hotel = Hot [Room]
```

Initially, the Hotel is full. Admittedly, the guests have boring names:

```
initialFullHotel :: Hotel
initialFullHotel = Hot [ Just $ "Guest" ++ show n | n <- [(1::Integer)..] ]
```

To be sure that we never try to print the whole infinite hotel, here is a `Show` instance which only shows the first 10 rooms:

```
instance Show Hotel where
  show (Hot rooms) = "Hot [" ++ substring ++ ", ... ]" where
    substring = intercalate ", " $ map show (take 10 rooms)
```

I promise that now you can safely type and evaluate `initialFullHotel` in ghci.

Accomodating a single person is easy, right?

```
accommodateSingle :: Hotel -> Guest -> Hotel
accommodateSingle (Hot h) newGuest = undefined
```

If you replaced `undefined` above correctly, then you should get this:

```
E3> accommodateSingle initialFullHotel "Bob"
Hot [ Just "Bob", Just "Guest1", Just "Guest2"
    , Just "Guest3", Just "Guest4", Just "Guest5"
    , Just "Guest6", Just "Guest7", Just "Guest8"
    , Just "Guest9", ... ]
```

Also accomodating a finite group should be easy:

```
accommodateFiniteGroup :: Hotel -> [Guest] -> Hotel
accommodateFiniteGroup (Hot h) group = undefined
```

But what if `group` is infinite?

```
accommodateGroup :: Hotel -> [Guest] -> Hotel
accommodateGroup (Hot h) group = undefined
```

And what if we have a finite number of groups of infinite length?

```
accommodateFinitelyManyGroups :: Hotel -> [[Guest]] -> Hotel
accommodateFinitelyManyGroups (Hot h) groups = undefined -- Hint: use a fold!
```

Finally, what if we have infinitely many groups of infinite length?

```
accommodateArbitraryGroups :: Hotel -> [[Guest]] -> Hotel
accommodateArbitraryGroups (Hot h) groups = undefined
```

You might want to look up and use *Szudzik's Elegant Pairing Function.* Click here for a presentation and click here for an example in JavaScript.

Exercise 3.4: Other Trees

Recall the definition of binary trees from Lecture 4. Note that we only have `a`-type values at the leafs.

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

Change the definition to also have values at each intermediate node.

Then adapt the instances below.

```
instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
   fmap f (Leaf x)            = Leaf (f x)
   fmap f (Branch left right) = Branch (fmap f left)
                                       (fmap f right)

instance Applicative Tree where
-- pure ::   a -> Tree a
   pure = Leaf
-- (<*>) :: Tree (a -> b) -> Tree a -> Tree b
   (<*>) ftree (Leaf x)       = fmap ($ x) ftree
   (<*>) ftree (Branch xl xr) = Branch (ftree <*> xl)
                                       (ftree <*> xr)

instance Foldable Tree where
-- foldr :: (a -> b -> b) -> b -> Tree a -> b
  foldr f y (Leaf x)     = f x y
  foldr f y (Branch l r) = foldr f (foldr f y l) r

instance Traversable Tree where
-- traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse g (Leaf x)     = Leaf <$> g x
  traverse g (Branch l r) = Branch <$> traverse g l <*> traverse g r
```