Exercise File 3

```haskell
module E3 where

import Data.List
```

Exercise 3.1

Suppose we want to have unordered pairs, for which $(4, 5) == (5, 4)$. For example, think about a round of a game with two players. This is an exercise to define instances of the `Show` and `Eq` type classes.

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

Implement a `Show` and an `Eq` instance such that we get:

```
> show (UOP (1,4))
UOP (1,4)
> show (UOP (4,1))
UOP (1,4)
> UOP (1,4) == UOP (4,1)
True
```

```haskell
instance (Show a, Ord a) => Show (UnOrdPair a) where
  show (UOP (x,y)) = undefined
```

Hint: start by distinguishing whether we have `x < y` or not.

```haskell
instance Ord a => Eq (UnOrdPair a) where
  (==) (UOP (x1,y1)) (UOP (x2,y2)) = undefined
```

Hint: Use `||` and describe the two cases in which the pairs should be equal.

Exercise 3.2

Consider Hello World 2.0 from the lectures:

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

```
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!
```

Exercise 3.3

Recall the Modal Logic implementation:

```haskell
type World = Integer
type Universe = [World]
type Proposition = Int
type Valuation = World -> [Proposition]
type Relation = [(World,World)]
data KripkeModel = KrM Universe Valuation Relation

data ModForm = Prp Proposition
             | Not ModForm
             | Con ModForm ModForm
             | Box ModForm
             deriving (Eq,Ord,Show)

makesTrue :: (KripkeModel,World) -> ModForm -> Bool
makesTrue (KrM _ v _, w) (Prp k)   = k `elem` v w
makesTrue (m,w)          (Not f)   = not (makesTrue (m,w) f)
makesTrue (m,w)          (Con f g) =
  makesTrue (m,w) f && makesTrue (m,w) g
makesTrue (KrM u v r, w) (Box f)   =
  all (\w' -> makesTrue (KrM u v r,w') f) ws where
    ws = [ y | y <- u, (w,y) `elem` r ]
```

In this exercise you should extend this implementation in various ways.

Add a function to check for truth in a whole model:

```haskell
trueEverywhere :: KripkeModel -> ModForm -> Bool
trueEverywhere = undefined
```

Add diamonds, the dual of boxes. You can either add a new constructor `Dia` to the line `data ModForm = ...` above or define diamonds as an abbreviation in terms of `Not` and `Box`.

```haskell
dia :: ModForm -> ModForm
dia = undefined
```

Think about when we call two Kripke models equal? For example, the universe should be the same when viewed as a list, but the order of worlds should not matter. Uncomment this and implement an `instance Eq KripkeModel`:

```haskell
-- instance Eq KripkeModel where
--   (==) = undefined
```

You should know what a bisimulation is. If not, see the relevant part of the BRV book. Write a function that checks a given bisimulation:

```haskell
type Bisimulation = [(World,World)]

checkBisim :: KripkeModel -> KripkeModel -> Bisimulation -> Bool
checkBisim = undefined
```

Kripke models where all relations are equivalence relations are often used in epistemic logic to model a strong/hard notion of knowledge. Because of the axioms that characterize axiomatize the logic of such models, they are also called `S5 models`.

Representing equivalence relations with `Relation = [(World,World)]` is a big waste of space. For example, the equivalence relation

`[(0,0),(0,1),(1,0),(1,1),(2,2)]`

can also be represented much shorter as a list of lists: `[[0,1],[2]]`.

Implement semantics in this way:

```
type EquiRel = [[World]]

data KripkeModelS5 = KrMS5 Universe Valuation EquiRel

makesTrueS5 :: KripkeModelS5 -> ModForm -> Bool
makesTrueS5 = undefined
```

It is annoying that we have to rename `makesTrue` for S5 models. We can in fact also use the same name. If you are curious how, look up how to define a new type class!

Some more ideas what you could do:

- Write a function that takes a formula and outputs nice LaTeX code.
- Visualize Kripke models by writing a function that takes a model and returns code for the `dot` program from https://www.graphviz.org/. (You can also use the `graphviz` library from Hackage, but note that it is not included in `lts-11.11`, so you might have to use and older snapshot and older version of GHC.)
- Use QuickCheck to investigate Modal Logic: First, implement `instance Arbitrary KripkeModel` and `instance Arbitrary ModForm`. Then check some modal formulas. Note that random testing will never allow you to show validity, but it *can* refute it.

Exercise 3.4

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)
```

Try this out by typing `initialFullHotel` in ghci now.

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> 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 = undefined
```

But what if `group` is infinite?

```
accommodateGroup :: Hotel -> [Guest] -> Hotel
accommodateGroup = undefined
```

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

```
accommodateFinitelyManyGroups :: Hotel -> [[Guest]] -> Hotel
accommodateFinitelyManyGroups = undefined -- Hint: use a fold!
```

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

```
accommodateArbitraryGroups :: Hotel -> [[Guest]] -> Hotel
accommodateArbitraryGroups = undefined
```

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