Exercises 2

```haskell
module E2 where

import Data.List
```

Exercise 2.1: `Show` and `Eq`

This is an exercise to define instances of the `Show` and `Eq` type classes. We want to have a data type for unordered pairs, where $(4, 5) == (5, 4)$.

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

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

```
GHCi> show (UOP (1,4))
UOP (1,4)
GHCi> show (UOP (4,1))
UOP (1,4)
GHCi> 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 `||` for "or" and describe the two cases in which the pairs should be equal.

Exercise 2.2

The Luhn Algorithm is a formula for validating credit card numbers. Give an implementation in Haskell. The type declaration should run:

```haskell
luhn :: Integer -> Bool
luhn = undefined
```

This function should check whether an input number satisfies the Luhn formula. You might want to use the following function. (Look up `read` on hoogle!)

```haskell
digits :: Integer -> [Integer]
digits n = map (\x -> read [x]) (show n)
```

Next, use luhn to write functions for checking whether an input number is a valid American Express Card, Master Card, or Visa Card number. Consult Wikipedia for the relevant properties.

```haskell
isAmericanExpress, isMaster, isVisa :: Integer -> Bool
isAmericanExpress = undefined
isMaster = undefined
isVisa = undefined
```

Bonus question: Write a function that generates (random?) credit card numbers!

Exercise 2.3

(If you prefer Modal Logic over Search Puzzles, jump ahead to 2.4.)

A farmer is on one side of a river. He has a wolf, a goat and a cabbage:

```haskell
data Item =  Wolf | Goat | Cabbage | Farmer deriving (Eq,Show)
data Position = L | R deriving (Eq,Show)
type State = ([Item], [Item])

start :: State
start = ([Wolf,Goat,Cabbage,Farmer], [])
```

He can move to the other side of the river and may carry an animal with him:

```haskell
type Move = (Position, Maybe Item)
```

Implement this (look up the `++` and `\\` functions):

```haskell
move :: State -> Move -> State
move (l,r) (L, Just  a) = (l ++ [Farmer,a], r \\ [Farmer,a])
move (l,r) _            = undefined -- what are the other cases?
```

For example, we should have:

```
*E2> move start (R, Just Cabbage)
([Wolf,Goat], [Cabbage,Farmer])
```

But this particular move would be a bad idea. Because whenever the farmer is not there, the wolf will eat the goat and the goat will eat the cabbage! Implement this:

```haskell
someoneGetsEaten ::[Item] -> Bool
someoneGetsEaten xs = undefined
```

We want to avoid states where someone gets eaten and we are done if everyone is on the right side:

```haskell
isBad, isSolved :: State -> Bool
isBad    (l,r) = someoneGetsEaten l || someoneGetsEaten r
isSolved (l,_) = null l
```

Your goal now is to implement a search algorithm to find a solution. First, given a state, what can the farmer do?

```haskell
availableMoves :: State -> [Move]
availableMoves (l,r) = undefined
```

We now do depth-first search. To prevent infinite loops, `prev` tracks previous states.

```haskell
solve :: [State] -> State -> [[Move]]
solve prev s | isSolved s = [ [] ]
             | otherwise  = [ m : nexts | m <- availableMoves s
                                        , undefined -- TODO do not move into "prev"
                                        , undefined -- TODO avoid if state isBad
                                        , nexts <- solve (s:prev) (move s m) ]

allSolutions :: [[Move]]
allSolutions = solve [] start

firstSolution :: [Move]
firstSolution = head allSolutions
```

Can you also find an optimal solution, with the fewest moves? Hint: Look up the functions `minimumBy` and `Data.Function.on`.

See also https://malv.in/posts/2021-01-09-depth-first-and-breadth-first-search-in-haskell.html.

Exercise 2.4

Recall the Modal Logic implementation:

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

```haskell
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
```

When should we call two Kripke models equal? For example, the universe should be the same set, but the order of worlds should not matter. Uncomment the following and implement an `instance Eq KripkeModel`:

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

You should know what a bisimulation is. If not, see Section 2.2 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. After the the axioms of 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

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

can also be represented much shorter as a list of lists: `[[0,1],[2]]`. You can also thinkg of this as a *partition*: each of the inner lists is an equivalence class.

Implement semantics in this way:

```haskell
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 your own new type class and a polymorphic `makesTrue`.

Some more ideas what you could do:

- 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.

  Note: to make QuickCheck available to GHC and therefore to make "`import Test.QuickCheck`" work you might first have to run "`cabal install --lib QuickCheck`", but this depends on how you installed GHC and other tools.

- Write a function that takes a formula and outputs nice LaTeX code.
- (Warning: this is more than a simple question.) Visualize Kripke models by writing a function that takes a model and returns code for the `dot` program from https://www.graphviz.org/. Some old code for this by Malvin: https://w4eg.de/malvin/illc/kripkevis/. A better way is to use the `graphviz` Haskell library.

Note: Enjoy all the errors!

Besides errors and type checking, GHC can help you with *warnings*. You should start it with `-Wall` like this:

```
ghci -Wall E2.lhs
```

Another great tool to improve your Haskell code is `hlint`. Install it with `stack install hlint` and then run `hlint Bla.lhs` to check a file. (Your editor might already show you those hints.)

For this exercise, reload your `E1.lhs` and `E2.lhs` files with all warnings enabled and fix any warnings you get. Also run `hlint` on both files, try to understand the suggestions and follow them. (Later we will aim for zero warnings and zero hlint sugggestions!)