Exercise File 2

```haskell
module E2 where

import Data.List
```

Exercise 2.1

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 2.2

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:

```
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:

```
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?

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

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

```
solve :: [State] -> State -> [[Move]]
solve done s | isSolved s = [ [] ]
             | otherwise  = [ m : nexts | m <- availableMoves s
                                        , undefined -- TODO do not move into "done"
                                        , undefined -- TODO do not go to a bad state
                                        , nexts <- solve (s:done) (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`.

Exercise 2.3

Besides the default type checking, GHC can help you with *warnings*. You should start it with `-Wall` to enable them. To do this with stack, use this full command:

```
stack exec 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.

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.