Be aware that the section on I/O differs a lot between the PDF and the web version, especially the part on exception handling. The PDF is the version we are officially following. If you have read in the web version, a quick heads up, the `catch` function from `System.IO.Error` is deprecated, use the and `catch` functions from `Control.Exception` instead. The examples from the web version should work if you simply change the import.

1 Hello, World!

- Write a Hello World program that prints some text.
- Write a program that asks for your name, reads it from stdin and greets that person.
- Write a program that counts the number of lines read from stdin
- Write a program that transforms text from stdin to upper case.
- Rewrite one of the previous programs to take their input from a file, with the filename given as a command line argument.

2 Scoreboard

Make a program that calculates the scores of some players in the following way. Input is given in a file, starting with a line stating which players were playing, each represented as an uppercase letter. The next line shows, the order in which players gained or lost points as a string, where an uppercase letter means the corresponding player gained a point and a lowercase letter means that player lost a point.

As an example two players `A` and `B` played a short game:

```
AB
AABaB
```

First `A` won two rounds, then `B` won, then `A` lost, and finally `B` won again. The final result is `A` has a score of `1` and `B` a score of `2`.

You should write the result to a new file, in a reasonable way. For example each player on its own line:

```
A: 1
B: 2
```

But initially you should probably just print/show the object, as I prefer you thinking about how to solve the problem, rather than details of how to present it in a pretty way.

Hint: Start with a simple solution that works according to some (or a lot) assumptions, and later refactor your solution to be better.

Hint: You might find that `Data.Map.Strict` supplies a helpful data structure and helper functions.

MORE EXAMPLES

Input:

```
ABC
ABCABCbaa
```

Output:

```
A: 0
B: 1
C: 2
```

Input:

```
quickfox
QFOqUQFIIOoxXCckxxoQ
```

Output:

```
C: 0
F: 2
I: 2
K: -1
O: 0
Q: 2
U: 1
X: -2
```

2.1 Some improvements

After you made your first version, improve it, so it works from fewer assumptions. This may be fast if you made a great solution initially.

Things you may consider improving:

- If you assumed the players came in ordered like `['A'..]`, you could change it so it works for any order of players, such as `"BXM"`, or even `"bXm"`.
- Improve the printing of the result, so it shows more like the example.

3 Lazy IO and Bracket

3.1 Lazy IO

Make two programs, one that gets the contents of a file, and prints the first character, and another one that prints the last character of that file.

Use some kind of time tool (such as `time` on Linux), and see the difference in time when running the programs on a large file, such as the `lipsum.txt` file uploaded here.

3.2 Bracket

Write a program that uses `bracket` to open and close a file, and for the "main" part counts the number of lines in the file. - Add "debug" information to the opening and closing of the file so you can follow what happens, i.e. when opening the file it outputs something like "Opened a file". - What happens when an error occurs in the "main" part?

Try changing the `bracket` to `bracketOnError` - What happens now when there is no error in the main part? - What happens when there is an error in the main part? - What happens when there is an error in the first part, e.g. the file does not exist?

3.3 Try

Use the `try` or `tryJust` function from `Control.Exception` (https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Exception.html#g:7) to recover from an error, e.g. an expected command line argument, or a file not existing.

4 Another game: Nim

The game Nim exists in many variants, usually between two players. One of the simple variants is played with a pile of items, say, 15, and each round the current player must take a number of items in some range, say, 1-4 items. The goal is to not take the last item, so the range does not include 0.

You should implement this game, you can decide on the rules to play by, i.e. how many items in the pile, whether you win or lose when you take the last item, etc. The program should write the number of items left after each player has had their turn.

To save some time, you don't need to think about which players turn it is, the people playing it can keep track of that themselves.

4.1 Improvements

Ideas for additions to the game:

- The program actually terminates when the game is finished (number of items == 0), with a small message.
- As mentioned, the number of items and the range is decided by input at the beginning of the program.
- Make it so there are multiple piles each with their own number of items, so the player first chooses which pile to draw from, then the number of items to draw.
- Make a datastructure for the game, and make it a custom instantiation of the `Show` typeclass. Use this to show the game, rather than simply the number of items left in the pile.
- Make it part of the `Game` typeclass, from the typeclass exercise from sheet 3, if you made it.

5 Working towards monads

It is important you understand types, typeclasses and functors to a certain degree before tackling monads, so here are some more exercises on that.

5.1 Types and typeclasses

If you haven't already, read and follow along the section "A yes-no typeclass" in chapter 7. Feel free to make your own type and make it an instance as well.

Also make sure you've done exercise 5 on sheet 3.

Make a `Describable` typeclass. Instances must implement a function `describe` that takes something describable and returns a tuple of the description and the thing, `(String, a)`. Make some types instances of `Describable`, e.g. `Bool`, `Char`, some custom type.

```
ghci> describe True
("One of my favorite booleans: True",True)
ghci> describe 'Y'
("This one has character: 'Y'",'Y')
```

5.2 Functors

Again, make sure you've made exercises 6 on sheet 3, which includes following the example from "The Functor typeclass" in chapter 7.

Remember, we only have to implement `fmap` for a type to become members of the `Functor` typeclass. If we think of functors as boxes that hold some value, then `fmap` takes a function that transforms the held value and creates a new box with the new value inside.

Make a type `Described a` that holds any type, but has a description (`String`) as well. You can use record syntax to define it and/or make extra functions to retrieve or change the description.

Make `Described` an instance of the `Functor` typeclass, by making `fmap` function on the value, and not changing the description.

Consider, how you would take any simple value and make it into a `Described` (if you are not given a description). Also consider how you would handle the case where you have a function in a described, along with a value in a described and you would like to apply the function to the value and keep the result in a described. In "pseudo" Haskell:

```
(Described (+3) "Plus three") `apply` (Described 39 "A good number") = (Described 42 ?)
```

Once you have decided how you want to handle that, you can make your `Described` into an `Applicative` functor.

6 Monads

Note that LYH says that you do not need to make your type an instance of `Applicative` to make an instance of `Monad`, but THIS IS NOT TRUE - in recent versions of GHC this is a requirement!

Make sure you've read and followed along the example in "Walk the line" in chapter 11 which shows one way to utilize the `Maybe` monad. Also make sure to try to "desugar" the do notation in the examples throughout the "do notation" section.

In Java you can make objects representing computations that might finish in the future, and sometimes you'd like to make more computations on whatever the result may be. You can encapsulate the new result in a new future, to further postpone the computations. This is actually usually the case in Haskell because it is lazy, but we continue with the example anyways.

In the following example try to make `Future` an instance of `Monad`. For `>>=` it should just unwrap the value inside the `Future` and apply the function to it.

```
data Future a = Future a deriving Show

instance Functor Future where
  fmap f (Future x) = Future (f x)
```

```haskell
instance Applicative Future where
  pure x = Future x
  (Future f) <*> (Future x) = Future (f x)
```

Once you've made `Future` into a monad, try "desugaring" `foo (Future 21)`, where `foo` is defined as:

```haskell
foo :: (Num a) => Future a -> Future a
foo m = do
  x <- m
  return (x*2)
```

7 The Maybe monad

These exercises are heavily inspired by the All About Monads article on the Haskell wiki.

Suppose you have some historic data about the heritage of some people, and you want to write a program to make some queries about them, e.g. "Who is the father of the mother of this person?". For each person you are given their name and their parents, but the data is incomplete, so some parents are missing. Therefore we use `Maybe` values to represent the parents.

We write the following data type.

```haskell
data Person = Person { name :: String,
                       father :: Maybe Person,
                       mother :: Maybe Person
                     } deriving Show
```

Remember, the record syntax means we now have the functions `name :: Person -> String`, `father :: Person -> Maybe Person`, and `mother :: Person -> Maybe Person`.

- Optional: Remove the `deriving Show`, and instead make your own implementation of the instance, e.g. with `show = name`.

In figure 1 there is some data you can work with. Casper & Shakira and Kobe & Husna are the main ancestors, with Kaan and Aubree as children respectively, in other words, an arc goes from a parent to a child.

In the program, this has been modeled from child to parents instead:

```haskell
-- Female
shakira = Person "Shakira" Nothing Nothing
husna   = Person "Husna"   Nothing Nothing
aubree  = Person "Aubree"  (Just kobe) (Just husna)
juna    = Person "Juna"    (Just ewan) (Just aubree)
sarah   = Person "Sarah"   (Just quentin) (Just juna)
```

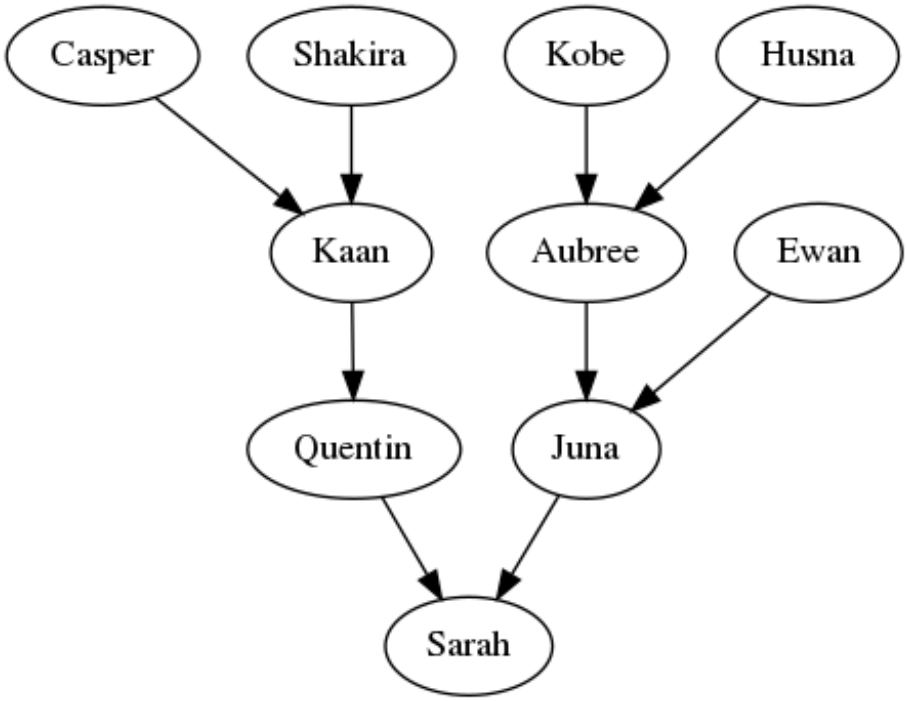

Figure 1: Family tree

```haskell
-- Male
casper  = Person "Casper"  Nothing Nothing
kobe    = Person "Kobe"    Nothing Nothing
kaan    = Person "Kaan"    (Just casper) (Just shakira)
ewan    = Person "Ewan"    Nothing Nothing
quentin = Person "Quentin" (Just kaan) Nothing
```

- Write a function that returns the paternal grandmother of a person, i.e. their father's mother. `paternalGrandmother :: Person -> Maybe Person`
- Write a function `mothersMaternalGrandfather :: Person -> Maybe Person`, that returns the mother's mother's father of a person.

What might make writing these functions a little tedious is that we'd like to use our `father` and `mother` functions, but they only work for `Person`s and we keep getting `Maybe Person`s instead. A solution could be to write a `Maybe Person -> Maybe Person` function, for each `Person -> Maybe Person` function. If we did that, we might realize that it works the same, every time: If the given `Maybe Person` is `Nothing` we have nothing to compute and return `Nothing`, otherwise we have a `Just p` and we can apply the function to `p`.

So instead we will make a function that helps us unwrap the `Person` from the `Maybe` and applies the function, whenever it can.

- Write a function `withMaybe :: Maybe Person -> (Person -> Maybe Person) -> Maybe Person` that applies the given function if possible, as described.

Now, to write `maternalGrandfather` we can just do:

```haskell
maternalGrandfather p    = (mother p) `withMaybe` father
```

And because the result of using `withMaybe` is again a `Maybe Person` we can chain them as long as we want:

```haskell
fathersMothersMothersFather p    = (father p) `withMaybe` mother `withMaybe` mother `withMay
```

Note that the function is doubly cool, not only can we chain the functions, but a `Nothing` at any point, correctly stops the further computation.

And if we really like our new function, or just prefer all the functions being on the right of the initial `p`, we can make `p` to a `Maybe Person` using `Just`.

```haskell
fathersPaternalGrandmother p    = (Just p) `withMaybe` father `withMaybe` father `withMaybe`
```

- Write a `femaleRolemodel` function that given a person, returns the closest female ancestor if one exists, i.e. their mother, or paternalGrandmother, or fathersPaternalGrandmother, etc. For Sarah this is Juna, but for Quentin, this is Shakira. This exercise is a little different than the previous, you can use anything you've learnt so far. Hint: the `isJust` function from `Data.Maybe` could be useful.

While working on another project with databases, you notice you have a lot of functions that look like `queryXY :: X -> Maybe Y` that you would like to chain, and it reminds you of the `withMaybe` function you made. You realize `withMaybe` does not rely on the `Person` type, except for in the signature, so you can generalize it to:

```
withMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  `withMaybe` _    = Nothing
(Just x) `withMaybe` f    = f x
```

At this point, you have pretty much rediscovered the monad, without maybe thinking too much about it.

Monads are a generalization of that pattern, where the `withMaybe` function is called bind and looks like `>>=` instead. Additionally, one must provide a way to get a value to a minimal context, for `Maybe` we do it with `Just`.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a

instance Monad Maybe where
  Nothing  >>= _    = Nothing
  (Just x) >>= f    = f x
```

- Write `paternalGrandmother` and `fathersMaternalGrandmother` functions using `return` and `>>=`.
- Desugar the value of `(return sarah) >>= mother`.
- Desugar the value of `bar quentin` and `bar ewan` where definition of `bar` is:

  ```
  bar mp = do
    p <- mp
    f <- father p
    return f
  ```

- Optional (list monad): make a function `parents :: Person -> [Person]` that returns a list of the parents to a given person. Hint: `maybeToList` from `Data.Maybe` could be useful.

8 The list monad

For this exercise assume the following is the definition of `Monad` (as shown in LYH), along with the instance for lists:

```
class Monad m where
    return :: a -> m a
```

```
    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg

instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

You shouldn't add this to a file, but think of it as the implementation.

- Desugar the value of `foo`:

```
import Data.Char

foo = "abc" >>= \c -> [toUpper c, toLower c]
```

- Desugar the value of `bar`. What is its type signature?

```
bar = do
    c <- "abc"
    d <- [1,2,3]
    return (c, d)
```

- Desugar the value of `baz [3,4,5]` where

```
baz :: [Int] -> [Int]
baz xs = do
  x <- xs
  [1,2]           -- Equivalent to _ <- [1,2]
  return x
```

- Write a function `makeTriples :: [a] -> [b] -> [c] -> [(a, b, c)]` using do-notation, such that it returns a list of all triples (a,b,c) where `a` is from the first list, `b` is from the second, and `c` is from the third. For a call `l = makeTriples as bs cs` it should hold that `head l = (head as, head bs, head cs)` and `last l = (last as, last bs, last cs)`.
- Write a function that uses `do`-notation to double the value of all numbers in a list.
- Write a function that uses `do`-notation to make gibberish from a string. The gibberish should come from repeating each letter with an 'o' in between, e.g. the string "b" becomes "bob" and the string "bat" becomes "bobaoatot".

– Feel free to change the letter inserted in between or making sure only consonants are repeated.

9 The State Monad

As with the last exercise we have the following assumption:

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg

newtype State s a = State { runState :: s -> (a,s) }

instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                        (State g) = f a
                                    in  g newState
```

Following the example of a stack from LYH, modified to use the function `state` in `pop` and `push` instead of the constructor `State`, just imagine they do the same.

```
import Control.Monad.Trans.State

type Stack = [Int]

pop :: State Stack Int
pop = state $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = state $ \xs -> ((),a:xs)

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

- What is the type of:
  - `runState pop`
  - `runState (push 42)`
  - `runState stackManip`
- Desugar `runState pop [42,41]`
- Desugar `runState (push 42) []`
- Desugar `runState stackManip [1,2]`
- Write a function that swaps the two top elements of a stack.
- Write a function that returns the third item from the top, but leaves the rest of the stack unchanged.

10 Monad laws

- What are the monad laws?
- Check that the laws hold for the `Maybe` and list monads.