Type Constructor Classes

```
-- Imports necessary to compile this file in ghc
import Prelude hiding (Functor(..), map)
import System.Environment (getArgs)
```

So far, we have used classes to overload functions for different types. This idea can be transfered to type constructors. For example, we've already seen two `map` functions: one for lists

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

and one for trees.

```
data Tree a = Empty | Node (Tree a) a (Tree a)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Empty = Empty
mapTree f (Node l x r) =
  Node (mapTree f l) (f x) (mapTree f r)
```

What both definitions have in common is that the `map` function can be defined for type constructors with arity one and we can generalise the type of `map`, which can be modeled by means of a type constructor class as follows.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Here the variable `f` is a variable for type constructor. It does not abstract from a type, but from a type constructor of arity one.

Then we can define the following `Functor` instances.

```
instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree
```

Also for `Maybe` it is possible to define an instance as follows.

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

We apply the given function to the (possibly present) value in the container.

Using the class `Functor`, it is now possible to define functions like the following.

```haskell
inc :: Functor f => f Int -> f Int
inc = fmap (+1)
```

This function can then be applied to lists, trees or maybe values.

Also `IO` is a unary type constructor and there is also `Functor` instance.

```haskell
instance Functor IO where
  fmap f a = do x <- a
                return (f x)
```

With this instance at hand, we can write the following program.

```haskell
main = do x <- fmap length getLine
          print x
```

It reads a string from the user and prints its length.

```
ghci> main
abc
3
```

Another example, which prints the first parameter from the console, can be defined as follows.

```haskell
main2 = do x <- fmap head getArgs
           print x
```

Saving this program as a file `print-first-arg.hs`, we can execute with the following command.

```
bash# runhaskell print-first-arg.hs 42 43 44
42
```

The class `Functor` and all presented instances (except the one for trees) are predefined in Haskell. You can directly use them and easily define new instances for your own data types.

Instances of class `Functor` have to fulfil the following laws (called functor laws).

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

These laws basically capture that `fmap` is a homomorphism.

As an example we check these laws for the `Maybe` instance.

```haskell
    fmap id Nothing  = Nothing     = id Nothing
    fmap id (Just x) = Just (id x) = id (Just x)

    fmap (f . g) Nothing
      = Nothing
```

```
  = fmap f (fmap g Nothing)
  = (fmap f . fmap g) Nothing

fmap (f . g) (Just x)
  = Just ((f . g) x)
  = Just (f (g x))
  = fmap f (fmap g (Just x))
  = (fmap f . fmap g) (Just x)
```

For recursive data structures like lists or trees we have to use structural induction to prove the functor laws. To prove the functor laws for `IO` we need other laws for the `do` notation, which we do not know yet. However we will discuss them later.