Fortgeschrittene funktionale Programmierung in Haskell

Übungszettel 2

Aufgabe 2.1:

In der Vorlesung wurde neben dem Datentypen `Maybe` auch `Either` vorgestellt. Zur Erinnerung: `Either` ist definiert als

```
data Either a b = Left a
                | Right b
```

Erstellen Sie hierzu die Instanzen für

- Functor

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

- Applicative

  ```
  class Functor f => Applicative f where
      pure  :: a -> f a
      (<*>) :: f (a -> b) -> f a -> f b
  ```

- Monad

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

Aufgabe 2.2:

Ein weiterer einfacher Datentyp ist `Identity`, welcher der Datentyp ist, der keinen Effekt hat. Definiert ist `Identity` als

```
newtype Identity a = Identity {runIdentity :: a}
```

Dieses definiert zwei Funktionen: Eine, um eine `Identity` zu erstellen, und eine, um wieder an ihren Inhalt zu kommen:

```
Identity    :: a -> Identity a
runIdentity :: Identity a -> a
```

Erstellen Sie hier ebenfalls Instanzen für Functor, Applicative und Monad **ohne** Pattern-matching auf den Inhalt der `Identity` (Nutzen Sie die Funktion `runIdentity`).

Aufgabe 2.3:

Hinweis: Dies ist eine Knobelaufgabe, in der beide obigen Aufgaben auf einmal gemacht werden müssen. Typed Holes (_) helfen euch sehr weiter.
In der Vorlesung wurde ebenfalls die State-Monade besprochen. Die Definition ist wie folgt:

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

Dieses definiert ebenfalls zwei Funktionen:

```
State    :: (s -> (a,s)) -> State s a
runState :: State s a -> s -> (a,s)
```

Erstellen Sie hier ebenfalls die Instanzen:

```
Functor (State s)
Applicative (State s)
Monad (State s)
```

Besonders Motivierte können auch die in der Vorlesung angesprochenen Funktionen get, put und modify implementieren und überprüfen, ob der Code aus der Vorlesung auch mit dem selbstgeschriebenen State funktioniert.