Übungsblatt 3

Throat-Clearing

a.k.a. Imports, damit der Code funktioniert.

```
import Control.Applicative
import Control.Monad
import Data.Monoid
```

Vorwort

Die Typklassen, die auf diesem Zettel implementiert werden sollen sind teilweise nicht eindeutig. Ein gutes *Indiz* für eine falsche implementation kann sein, dass Informationen "weggeschmissen" werden - allerdings muss man bei anderen Implementationen genau dies machen.

Applicative

Nachdem wir uns letzte Woche ausführlich mit der Typklasse `Functor` beschäftigt haben, bauen wir nun darauf auf und definieren die Applicative-Instanz. Zur Erinnerung:

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

Nehmen sie an, sie hätten folgende Datentypen mit ihren `Functor`-Instanzen gegeben. Schreiben sie jeweils die Applicative-Instanz:

```
data Identity a = Identity { unIdentity :: a }
                           deriving (Show, Eq)

instance Functor Identity where
  fmap f (Identity a) = Identity (f a)

data Vielleicht a = Etwas a
                  | Nichts
                deriving (Show, Eq)

instance Functor Vielleicht where
  fmap f (Etwas a) = Etwas (f a)
  fmap _ Nichts    = Nichts

data EntwederOder b a = Entweder a
```

```
                    | Oder b
                deriving (Show, Eq)

instance Functor (EntwederOder b) where
  fmap f (Entweder a) = Entweder (f a)
  fmap _ (Oder b)     = Oder b

data List a = Cons a (List a)
            | Nil
                deriving (Show, Eq)

instance Functor List where
  fmap f (Cons a r) = Cons (f a) (fmap f r)
  fmap _ Nil        = Nil

instance Monoid List where
  mempty                = Nil
  mappend Nil bs        = bs
  mappend (Cons a as) bs = Cons a (mappend as bs)

data V3 a = V3 a a a

instance Functor V3 where
  fmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

Monad

Zu welchen der oben aufgeführten Typen gibt es eine Monaden-Instanz? Wie sieht diese aus? Schreiben sie diese (falls möglich).

Bonus

Seien folgende Funktionen gegeben:

```
login    :: Maybe Account
login    = undefined

getInbox :: Account -> Maybe Inbox
getInbox = undefined

getMails :: Inbox -> [Mail]
getMails = undefined
```

```
safeHead :: [a] -> Maybe a
safeHead = undefined
```

Schreiben sie eine Funktion:

```
getFirstMail :: Maybe Mail
```

welche die oben genannten 4 Funktionen nutzt um die erste Mail aus dem gegebenen Account zurückzuliefern, sofern alles erfolgreich war.

Können sie beide Varianten (einmal mittels `do`-notation und einmal mit `>>=`) schreiben?