Fortgeschrittene funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

Datentypen sind eine Annotation, wie die Bits im Speicher interpretiert werden sollen.

Datentypen sind eine Annotation, wie die Bits im Speicher interpretiert werden sollen.

Einige primitive Datentypen sollten euch aus anderen Programmiersprachen schon bekannt sein:

- Zahlen (z.B. Int, Integer, Float, Double, ...)
- Zeichenketten (z.B. String, UTF-8-Strings, ...)
- Bool

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:
(`a`, `k` und `v` steht hier jeweis für einen (primitiven) Datentypen)

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:
(a, `k` und `v` steht hier jeweis für einen (primitiven) Datentypen)

- Liste von a

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:
(a, `k` und `v` steht hier jeweis für einen (primitiven) Datentypen)

- Liste von a
- Hashmap von `k` und `v`

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:
(`a`, `k` und `v` steht hier jeweis für einen (primitiven) Datentypen)

- Liste von `a`
- Hashmap von `k` und `v`
- Vektor von `a`

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:
(`a`, `k` und `v` steht hier jeweis für einen (primitiven) Datentypen)

- Liste von `a`
- Hashmap von `k` und `v`
- Vektor von `a`
- Tree von `a`

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:
(a, `k` und `v` steht hier jeweis für einen (primitiven) Datentypen)

- Liste von a
- Hashmap von `k` und `v`
- Vektor von a
- Tree von a
- Zusammengesetzte Typen (z.B. Structs in C/C++)

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:
(a, `k` und `v` steht hier jeweis für einen (primitiven) Datentypen)

- Liste von `a`
- Hashmap von `k` und `v`
- Vektor von `a`
- Tree von `a`
- Zusammengesetzte Typen (z.B. Structs in C/C++)

Im Folgenden gehen wir auf zwei wesentliche zusammengesetzte Typen in Haskell ein: `Maybe` und `Either`.

Einen neuen Datentypen definieren wir in Haskell mit dem Keyword `data`:

```
data Maybe a = Nothing
             | Just a
```

Einen neuen Datentypen definieren wir in Haskell mit dem Keyword `data`:

```
data Maybe a = Nothing
             | Just a
```

Was hat das für einen Sinn?

Einen neuen Datentypen definieren wir in Haskell mit dem Keyword `data`:

```
data Maybe a = Nothing
             | Just a
```

Was hat das für einen Sinn?

`Maybe a` gibt das Ergebnis einer Berechnung vom Typ a an, die fehlschlagen kann.

Einen neuen Datentypen definieren wir in Haskell mit dem Keyword `data`:

```
data Maybe a = Nothing
             | Just a
```

Was hat das für einen Sinn?

`Maybe a` gibt das Ergebnis einer Berechnung vom Typ a an, die fehlschlagen kann.

In klassischen Sprachen wird hier meist ein „abgesprochener" Fehlerzustand zurückgegeben (0, -1, null, ...). In Haskell wird dies über den Rückgabetyp deutlich gemacht.

Nachteile:

- Ein neuer Datentyp, den man kennen muss

Nachteile:

- Ein neuer Datentyp, den man kennen muss

Vorteile:

- keine Absprachen, die man vergessen kann

Nachteile:

- Ein neuer Datentyp, den man kennen muss

Vorteile:

- keine Absprachen, die man vergessen kann
- einheitliche Behandlung aller Fälle

Nachteile:

- Ein neuer Datentyp, den man kennen muss

Vorteile:

- keine Absprachen, die man vergessen kann
- einheitliche Behandlung aller Fälle
- mehrere möglicherweise fehlschlagende Operationen gruppieren und nur solange evaluieren, bis die erste fehlschlägt oder alle erfolgreich sind

Beispiel: Finden eines Elementes in einer Liste

Beispiel: Finden eines Elementes in einer Liste

```
ghci > import Data.List
ghci > :t findIndex
findIndex :: (a -> Bool) -> [a] -> Maybe Int
```

Beispiel: Finden eines Elementes in einer Liste

```
ghci > import Data.List
ghci > :t findIndex
findIndex :: (a -> Bool) -> [a] -> Maybe Int

ghci > findIndex (== 5) [1..10]
Just 4                 -- [1..10] !! 4 => 5
```

Beispiel: Finden eines Elementes in einer Liste

```
ghci > import Data.List
ghci > :t findIndex
findIndex :: (a -> Bool) -> [a] -> Maybe Int

ghci > findIndex (== 5) [1..10]
Just 4                 -- [1..10] !! 4 => 5

ghci > findIndex (== 1000) [1..10]
Nothing
```

Beispiel: Finden eines Elementes in einer Liste

```
ghci > import Data.List
ghci > :t findIndex
findIndex :: (a -> Bool) -> [a] -> Maybe Int

ghci > findIndex (== 5) [1..10]
Just 4                   -- [1..10] !! 4 => 5

ghci > findIndex (== 1000) [1..10]
Nothing
```

Da wir 1000 in der Liste der `Ints` von 1-10 nicht finden können, haben wir keinen gültigen Index, daher wird `Nothing` zurück gegeben.

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

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

Was hat das für einen Sinn?

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

Was hat das für einen Sinn?

Either benutzt man, wenn man ein erwartetes Ergebnis `Right b` vom Typen `b` hat **oder** einen Fehlerzustand `Left a` vom Typen `a`. Meistens ist das erste Argument `String` um eine lesbare Fehlermeldung zu bekommen.

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

Was hat das für einen Sinn?

Either benutzt man, wenn man ein erwartetes Ergebnis `Right b` vom Typen `b` hat **oder** einen Fehlerzustand `Left a` vom Typen `a`. Meistens ist das erste Argument `String` um eine lesbare Fehlermeldung zu bekommen.

Einfach zu merken: `Right` ist der „richtige“ Konstruktor.

Beispiel für eine Benutzung von `Either`:

```
parse5 :: String -> Either String Int
parse5 "5" = Right 5
parse5 _   = Left "Could not parse 5"
```

Beispiel für eine Benutzung von `Either`:

```haskell
parse5 :: String -> Either String Int
parse5 "5" = Right 5
parse5 _   = Left "Could not parse 5"

parse5 "5"   -- Right 5
parse5 "abc" -- Left "Could not parse 5"
```

Man kann Typen generell in 4 Kategorien einteilen:

Man kann Typen generell in 4 Kategorien einteilen:

- **Summentypen**: Maybe, Either, Bool, Int, ...

Man kann Typen generell in 4 Kategorien einteilen:

- **Summentypen**: Maybe, Either, Bool, Int, ...
- **Produkttypen**: Vektoren, Tupel, ...

Man kann Typen generell in 4 Kategorien einteilen:

- **Summentypen**: Maybe, Either, Bool, Int, ...
- **Produkttypen**: Vektoren, Tupel, ...
- **rekursive Typen**: Liste, Stream, ...

Man kann Typen generell in 4 Kategorien einteilen:

- **Summentypen**: Maybe, Either, Bool, Int, ...
- **Produkttypen**: Vektoren, Tupel, ...
- **rekursive Typen**: Liste, Stream, ...
- **Exponentialtypen**: Funktionen

Man kann Typen generell in 4 Kategorien einteilen:

- **Summentypen**: Maybe, Either, Bool, Int, ...
- **Produkttypen**: Vektoren, Tupel, ...
- **rekursive Typen**: Liste, Stream, ...
- **Exponentialtypen**: Funktionen

Warum werden diese Typen gerade so genannt?

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Sum = S1 Bool | S2 (Maybe Bool) deriving Show
```

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Sum = S1 Bool | S2 (Maybe Bool) deriving Show
```

```
S1 True, S1 False,
S2 (Just True), S2 (Just False), S2 (Nothing)
```

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Sum = S1 Bool | S2 (Maybe Bool) deriving Show
```

```
S1 True, S1 False,
S2 (Just True), S2 (Just False), S2 (Nothing)
```

Insgesamt 5. Daher auch die Bezeichnung Summentyp, da $2 + (2 + 1) = 5$ (2 Belegungen für `Bool`, 2+1 für `Maybe Bool`).

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Prod = P Bool Sum
```

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Prod = P Bool Sum
```

```
P True (S1 True), P False (S1 True), P True (S1 False), ...
... , P True (S2 Nothing), P False (S2 Nothing)
```

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Prod = P Bool Sum
```

```
P True (S1 True), P False (S1 True), P True (S1 False), ...
... , P True (S2 Nothing), P False (S2 Nothing)
```

Insgesamt 10. Daher auch die Bezeichnung Produkttyp, da $2 * 5 = 10$ (2 Belegungen für `Bool`, 5 für `Sum`).

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Stream a = SI a (Stream a)
```

Wie viele mögliche Werte kann folgender Typ annehmen?

```
data Stream a = SI a (Stream a)
```

Unendlich viele (siehe auch: unendliche Listen).

Wie viele mögliche Funktionen folgender Form gibt es?

```
f :: Maybe Bool -> Bool
```

Wie viele mögliche Funktionen folgender Form gibt es?

```haskell
f :: Maybe Bool -> Bool
```

Insgesamt 8, weil $8 = 2^3$:

Just True	Just False	Nothing
False	False	False
False	False	True
False	True	False
False	True	True
True	False	False
True	False	True
True	True	False
True	True	True

Wozu interessiert uns das?

Wozu interessiert uns das?

- Wir können Abschätzen, wie viele *mögliche* Bewohner eines Typen es gibt.

Wozu interessiert uns das?

- Wir können Abschätzen, wie viele *mögliche* Bewohner eines Typen es gibt.
- Es sollte uns lehren immer die kleinstmöglichen Typen zu nehmen.

Wozu interessiert uns das?

- Wir können Abschätzen, wie viele *mögliche* Bewohner eines Typen es gibt.
- Es sollte uns lehren immer die kleinstmöglichen Typen zu nehmen.
- Es bietet viele theoretische Spielereien (Beweise, Kategorientheorie, ...).

Wozu interessiert uns das?

- Wir können Abschätzen, wie viele *mögliche* Bewohner eines Typen es gibt.
- Es sollte uns lehren immer die kleinstmöglichen Typen zu nehmen.
- Es bietet viele theoretische Spielereien (Beweise, Kategorientheorie, ...).
- Wir können euch jetzt mit „Summentyp“ und „Produkttyp“ nerven und können hierauf verweisen :)

Viele Typen haben ähnliche oder gleiche Eigenschaften. Diese Eigenschaften fasst man zu **Typklassen** zusammen.

Viele Typen haben ähnliche oder gleiche Eigenschaften. Diese Eigenschaften fasst man zu **Typklassen** zusammen.

- Zahlen kann man alle verrechnen, auch wenn z.B. `Int` und `Double` verschiedene Typen sind.
- Listen, Vektoren, Arrays haben alle Elemente, über die man z.B. iterieren kann.
- Maybe, Either, Listen, etc. haben (vielleicht) einen Inhalt, die man verändern kann.

Viele Typen haben ähnliche oder gleiche Eigenschaften. Diese Eigenschaften fasst man zu **Typklassen** zusammen.

- Zahlen kann man alle verrechnen, auch wenn z.B. `Int` und `Double` verschiedene Typen sind.
- Listen, Vektoren, Arrays haben alle Elemente, über die man z.B. iterieren kann.
- Maybe, Either, Listen, etc. haben (vielleicht) einen Inhalt, die man verändern kann.

Warnung: Typklassen haben nichts mit den Klassen der Objektorientierung zu tun, sondern eher mit Templates und abstrakten Klassen.

```haskell
class  Eq a  where
     (==) :: a -> a -> Bool
--or (/=) :: a -> a -> Bool

class Eq a => Ord a  where
     (<=) :: a -> a -> Bool
-- definiert automatisch: compare, >=, <, >, max, min
```

```
class  Eq a  where
     (==) :: a -> a -> Bool
--or (/=) :: a -> a -> Bool

class Eq a => Ord a  where
     (<=) :: a -> a -> Bool
-- definiert automatisch: compare, >=, <, >, max, min
```

Im folgenden stellen wir ein paar Typklassen vor:
`Functor, Applicative, Monoid, Monad, MonadPlus`

Ein Funktor F lässt sich auf jedem Datentypen definieren, der sowas wie einen „Inhalt“ hat.

Ein Funktor F lässt sich auf jedem Datentypen definieren, der sowas wie einen „Inhalt“ hat.

Genauer: Dass ein Datentyp ein Funktor ist wird definiert über die Funktion `fmap`, die es erlaubt, eine Funktion auf den Inhalt anzuwenden.

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

`f` heisst hier **der Kontext**, in dem `a` steht.

Ein Funktor F lässt sich auf jedem Datentypen definieren, der sowas wie einen „Inhalt“ hat.

Genauer: Dass ein Datentyp ein Funktor ist wird definiert über die Funktion `fmap`, die es erlaubt, eine Funktion auf den Inhalt anzuwenden.

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

`f` heisst hier **der Kontext**, in dem a steht.

Spoiler: Liste ist ein Funktor. Maybe auch. Wieso?

Hilfreiche Analogie:
Für den Anfang kann man sich den **Kontext** als eine „Kiste“ vorstellen, in der etwas liegt.

Hilfreiche Analogie:
Für den Anfang kann man sich den **Kontext** als eine „Kiste“ vorstellen, in der etwas liegt.

`fmap` wertet also nur Funktionen, die auf dem **Inhalt** der Kiste funktionieren würden, zu Funktionen auf, die auf **Kisten mit Dingen** funktionieren.

Hilfreiche Analogie:
Für den Anfang kann man sich den **Kontext** als eine „Kiste" vorstellen, in der etwas liegt.

`fmap` wertet also nur Funktionen, die auf dem **Inhalt** der Kiste funktionieren würden, zu Funktionen auf, die auf **Kisten mit Dingen** funktionieren.

Man kann `fmap` daher auch etwas anders Klammern:

```
fmap :: (a -> b) -> (f a -> f b)
```

`fmap` nimmt somit eine Funktion und gibt eine neue Funktion zurück, die auf dem Kontext `f` funktioniert.

Functor-Instanz von Maybe:

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

Functor-Instanz von Maybe:

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

Functor-Instanz von Listen:

```
instance Functor [] where
    fmap f (x:xs) = f x : (fmap f xs)
    fmap _ []     = []
```

Functor-Instanz von Maybe:

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

Functor-Instanz von Listen:

```
instance Functor [] where
    fmap f (x:xs) = f x : (fmap f xs)
    fmap _ []     = []
```

fmap auf Listen ist einfach das bekannte map.

Functor-Instanz von Maybe:

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

Functor-Instanz von Listen:

```
instance Functor [] where
    fmap f (x:xs) = f x : (fmap f xs)
    fmap _ []     = []
```

fmap auf Listen ist einfach das bekannte map.

fmap hat auch eine infix-Schreibweise: <$>

Beispiel:

```
ghci > fmap (+1) (Just 3)
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
       Just 4
ghci > fmap (+1) Nothing
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
       Just 4
ghci > fmap (+1) Nothing
       Nothing
ghci > fmap (+1) [1..10]
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
       Just 4
ghci > fmap (+1) Nothing
       Nothing
ghci > fmap (+1) [1..10]
       [2,3,4,5,6,7,8,9,10,11]
ghci > fmap (+1) (Right 2)
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
       Just 4
ghci > fmap (+1) Nothing
       Nothing
ghci > fmap (+1) [1..10]
       [2,3,4,5,6,7,8,9,10,11]
ghci > fmap (+1) (Right 2)
       Right 3
ghci > fmap (+1) (Left 2)
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
       Just 4
ghci > fmap (+1) Nothing
       Nothing
ghci > fmap (+1) [1..10]
       [2,3,4,5,6,7,8,9,10,11]
ghci > fmap (+1) (Right 2)
       Right 3
ghci > fmap (+1) (Left 2)
       Left 2
```

Funktoren sind mathematische Objekte mit festen Eigenschaften. Damit der Compiler auch die richtigen Optimierungen machen kann, muss jeder Funktor folgende Regeln erfüllen:

Funktoren sind mathematische Objekte mit festen Eigenschaften. Damit der Compiler auch die richtigen Optimierungen machen kann, muss jeder Funktor folgende Regeln erfüllen:

```
-- Strukturerhaltung
fmap id = id
```

Die Datenstruktur darf sich nicht ändern.

Funktoren sind mathematische Objekte mit festen Eigenschaften. Damit der Compiler auch die richtigen Optimierungen machen kann, muss jeder Funktor folgende Regeln erfüllen:

```haskell
-- Strukturerhaltung
fmap id = id
```

Die Datenstruktur darf sich nicht ändern.

```haskell
-- Composability
fmap (f . g) = fmap f . fmap g
```

Mehrere `fmaps` hintereinander dürfen zusammengefasst werden, ohne, dass sich das Ergebnis ändert.

Negativbeispiel

```
fmap' f []     = []
fmap' f (a:as) = (f a):a:(fmap' f as)
```

Negativbeispiel

```
fmap' f []     = []
fmap' f (a:as) = (f a):a:(fmap' f as)
```

```
fmap' id [1,2,3] = [1,1,2,2,3,3] /= [1,2,3]
```

Negativbeispiel

```
fmap' f Nothing  = Nothing
fmap' f (Just a) = Just (f (f a))
```

Negativbeispiel

```
fmap' f Nothing  = Nothing
fmap' f (Just a) = Just (f (f a))

(fmap' (+1) . fmap' (*2)) (Just 1)
= fmap' (+1) (Just ((*2) ((*2) 1))
= fmap' (+1) (Just 4)
= Just 6

(fmap' ((+1).(*2)) (Just 1)
= Just (((+1).(*2).(+1).(*2)) 1)
= Just 7
```

`Applicative` funktioniert ähnlich zu `Functor`. Hierbei kann man auch mit Funktionen in einem Kontext arbeiten.

`Applicative` funktioniert ähnlich zu `Functor`. Hierbei kann man auch mit Funktionen in einem Kontext arbeiten.

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

`Applicative` funktioniert ähnlich zu `Functor`. Hierbei kann man auch mit Funktionen in einem Kontext arbeiten.

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

`pure` bringt etwas in den Standardkontext (z.B. Liste mit 1 Element, `Just x`, `Right x`, ..).
`<*>` ist fast ein `fmap`, nur dass die Funktion auch in demselben Kontext liegt.

Applicative-Instanz von Maybe:

```
instance Applicative Maybe where
    pure                     = Just
    Nothing  <*> _           = Nothing
    (Just f) <*> something = fmap f something
```

Applicative-Instanz von Maybe:

```
instance Applicative Maybe where
    pure                       = Just
    Nothing  <*> _             = Nothing
    (Just f) <*> something     = fmap f something
```

Applicative-Instanz von Listen:

```
instance Applicative [] where
    pure a        = [a]
    (f:fs) <*> x  = fmap f x ++ (fs <*> x)
    []     <*> _  = []
```

Auch für `Applicative` gibt es Gesetze:

```
pure f    <*> x      = fmap f x
pure id   <*> v      = v
pure f    <*> pure x = pure (f x)
     u    <*> pure y = pure ($ y) <*> u
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

Diese Gesetze regeln (abgesehen von der Verkettung) nur, dass sich Applicative konsistent zu Functor verhält.

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
       Just 4
ghci > Nothing <*> Just 3
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
       Just 4
ghci > Nothing <*> Just 3
       Nothing
ghci > pure (+1) <*> Right 2
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
       Just 4
ghci > Nothing <*> Just 3
       Nothing
ghci > pure (+1) <*> Right 2
       Right 3
ghci > pure (+1) <*> Just 2
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
       Just 4
ghci > Nothing <*> Just 3
       Nothing
ghci > pure (+1) <*> Right 2
       Right 3
ghci > pure (+1) <*> Just 2
       Just 3
ghci > [(+1),(*2)] <*> [1..5]
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
       Just 4
ghci > Nothing <*> Just 3
       Nothing
ghci > pure (+1) <*> Right 2
       Right 3
ghci > pure (+1) <*> Just 2
       Just 3
ghci > [(+1),(*2)] <*> [1..5]
       [2,3,4,5,6,2,4,6,8,10]
ghci > pure (*) <*> [1..3] <*> [1..3]
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
       Just 4
ghci > Nothing <*> Just 3
       Nothing
ghci > pure (+1) <*> Right 2
       Right 3
ghci > pure (+1) <*> Just 2
       Just 3
ghci > [(+1),(*2)] <*> [1..5]
       [2,3,4,5,6,2,4,6,8,10]
ghci > pure (*) <*> [1..3] <*> [1..3]
       [1,2,3,2,4,6,3,6,9]
```

Ein Monoid ist ein Konzept, was uns häufiger begegnen wird. Mathematisch gesehen ist ein Monoid eine Menge mit *assoziativer Operation* und *neutralem Element*.

Ein Monoid ist ein Konzept, was uns häufiger begegnen wird. Mathematisch gesehen ist ein Monoid eine Menge mit *assoziativer Operation* und *neutralem Element*.

Folglich lautet die Definition in Haskell:

```
class Monoid a where
 mempty  :: a
 mappend :: a -> a -> a

 mconcat :: [a] -> a
 mconcat = foldr mappend mempty
```

Ein Monoid ist ein Konzept, was uns häufiger begegnen wird. Mathematisch gesehen ist ein Monoid eine Menge mit *assoziativer Operation* und *neutralem Element*.

Folglich lautet die Definition in Haskell:

```
class Monoid a where
 mempty  :: a
 mappend :: a -> a -> a

 mconcat :: [a] -> a
 mconcat = foldr mappend mempty
```

Monoide, die ihr schon kennt, sind z.B. Listen (mit [] und ++) und Bäume.

Ein Monoid ist ein Konzept, was uns häufiger begegnen wird. Mathematisch gesehen ist ein Monoid eine Menge mit *assoziativer Operation* und *neutralem Element*.

Folglich lautet die Definition in Haskell:

```
class Monoid a where
 mempty  :: a
 mappend :: a -> a -> a

 mconcat :: [a] -> a
 mconcat = foldr mappend mempty
```

Monoide, die ihr schon kennt, sind z.B. Listen (mit `[]` und `++`) und Bäume.

Für `mappend` schreibt man auch `<>`.

Wozu Monaden?

Wozu Monaden?

- Man kann auch ohne sie funktional programmieren.

Wozu Monaden?

- Man kann auch ohne sie funktional programmieren.
- Monaden verhalten sich wie ein Semikolon in anderen Sprachen.

Wozu Monaden?

- Man kann auch ohne sie funktional programmieren.
- Monaden verhalten sich wie ein Semikolon in anderen Sprachen.
- Monaden „arbeiten“ im Hintergrund.

Beispiel:

```
f            :: Maybe Header
getInbox     :: Maybe Folder
getFirstMail :: Folder -> Maybe Mail
getHeader    :: Mail -> Maybe Header
```

Beispiel:

```
f            :: Maybe Header
getInbox     :: Maybe Folder
getFirstMail :: Folder -> Maybe Mail
getHeader    :: Mail -> Maybe Header
```

Ohne Monaden:

```
f = case getInbox of
      (Just folder) ->
          case getFirstMail folder of
            (Just mail) ->
                case getHeader mail of
                  (Just head) -> return head
                  Nothing     -> Nothing
            Nothing     -> Nothing
      Nothing       -> Nothing
```

Beispiel:

```
f            :: Maybe Header
getInbox     :: Maybe Folder
getFirstMail :: Folder -> Maybe Mail
getHeader    :: Mail -> Maybe Header
```

Mit Monaden:

```
f = do
      folder <- getInbox
      mail   <- getFirstMail folder
      header <- getHeader mail
      return header
```

Wie funktioniert diese Magie?

Wie funktioniert diese Magie?

Monaden benutzen die Funktion „bind " (»=) um Berechnungen zu verketten und Arbeit für uns im Hintergrund zu übernehmen.

Wie funktioniert diese Magie?

Monaden benutzen die Funktion „bind “ (»=) um Berechnungen zu verketten und Arbeit für uns im Hintergrund zu übernehmen.

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

Wie funktioniert diese Magie?

Monaden benutzen die Funktion „bind " (»=) um Berechnungen zu verketten und Arbeit für uns im Hintergrund zu übernehmen.

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

`return` funktioniert analog zu `pure` und bringt ein Element in den Standardkontext.
(»=) enthält die ganze Magie. Prinzipiell „packt" es ein `m a` aus und wendet die mitgegegbene Funktion an.

Monad-Instanz von Maybe:

```
instance Monad Maybe where
    return                  = pure   -- = Just
    (Just a) >>= f          = f a
    Nothing >>= _           = Nothing
```

Monad-Instanz von Maybe:

```
instance Monad Maybe where
    return                  = pure    -- = Just
    (Just a) >>= f          = f a
    Nothing >>= _           = Nothing
```

Monad-Instanz von Listen:

```
instance Monad [] where
    return       = pure
    (x:xs) >>= f = f x ++ (xs >>= f)
    []     >>= _ = []
```

Zurück zu unserem Beispiel. Wie helfen nun Monaden?

Zurück zu unserem Beispiel. Wie helfen nun Monaden?

```
f = case getInbox of
      (Just folder) ->
          case getFirstMail folder of
            (Just mail) ->
                case getHeader mail of
                  (Just head) -> return head
                  Nothing     -> Nothing
            Nothing     -> Nothing
      Nothing       -> Nothing
```

Schreiben wir mittels »= um zu:

Schreiben wir mittels »= um zu:

```
f = getInbox >>= (\folder ->
        getFirstMail folder >>= (\mail ->
            getHeader mail >>= (\header ->
                return header
            )
        )
    )
```

Schreiben wir mittels »= um zu:

```
f = getInbox >>= (\folder ->
        getFirstMail folder >>= (\mail ->
            getHeader mail >>= (\header ->
                return header
            )
        )
    )
```

»= fäng hier den Nothing-Fall ab und wir geben eine Funktion mit, die nur noch den Just-Fall behandeln muss.

Da dieses ganze `x »= (\v -> ... »= (\w -> ...))` hässlich ist, gibt es die do-notation.
Wir können das Beispiel von oben also umschreiben als:

Da dieses ganze `x >>= (\v -> ... >>= (\w -> ...))` hässlich ist, gibt es die do-notation.
Wir können das Beispiel von oben also umschreiben als:

```
f = do
      folder <- getInbox
      mail   <- getFirstMail folder
      header <- getHeader mail
      return header
```

Da dieses ganze `x »= (\v -> ... »= (\w -> ...))` hässlich ist, gibt es die do-notation.
Wir können das Beispiel von oben also umschreiben als:

```
f = do
      folder <- getInbox
      mail   <- getFirstMail folder
      header <- getHeader mail
      return header
```

<- extrahiert hier den Wert „aus der Monade“:

```
getInbox :: Maybe Folder

folder   :: Folder
```

Da dieses ganze `x >>= (\v -> ... >>= (\w -> ...))` hässlich ist, gibt es die do-notation.
Wir können das Beispiel von oben also umschreiben als:

```
f = do
      folder <- getInbox
      mail   <- getFirstMail folder
      header <- getHeader mail
      return header
```

<- extrahiert hier den Wert „aus der Monade“:

```
getInbox :: Maybe Folder
```

```
folder   :: Folder
```

Dieses automatische Zusammenfassen funktioniert nur, wenn **alle** Funktionen als letzten Wert etwas in derselben Monade zurückgeben.

In manchen Fällen können wir sogar auf die do-notation verzichten. Wenn jeder zurückgegebene Wert direkt von der nächsten Funktion gebraucht wird, dann sparen wir uns die λ-Ausdrücke und die do-Notation und verketten direkt:

In manchen Fällen können wir sogar auf die do-notation verzichten. Wenn jeder zurückgegebene Wert direkt von der nächsten Funktion gebraucht wird, dann sparen wir uns die λ-Ausdrücke und die do-Notation und verketten direkt:

```
f = getInbox >>= getFirstMail >>= getHeader
```

In manchen Fällen können wir sogar auf die do-notation verzichten. Wenn jeder zurückgegebene Wert direkt von der nächsten Funktion gebraucht wird, dann sparen wir uns die λ-Ausdrücke und die do-Notation und verketten direkt:

```
f = getInbox >>= getFirstMail >>= getHeader
```

Eine weitere Funktion, die einem in diesem Kontext begegnet ist », welche das Ergebnis verwirft und die nächste Funktion ohne Parameter aufruft:

```
  m >> k = m >>= \_ -> k

main = putStrLn "Geben sie etwas ein."
                    >>  getLine
                    >>= putStrLn
```

In manchen Fällen können wir sogar auf die do-notation verzichten. Wenn jeder zurückgegebene Wert direkt von der nächsten Funktion gebraucht wird, dann sparen wir uns die λ-Ausdrücke und die do-Notation und verketten direkt:

```
f = getInbox >>= getFirstMail >>= getHeader
```

Eine weitere Funktion, die einem in diesem Kontext begegnet ist $\gg$, welche das Ergebnis verwirft und die nächste Funktion ohne Parameter aufruft:

```
  m >> k = m >>= \_ -> k

main = putStrLn "Geben sie etwas ein."
                    >>  getLine
                    >>= putStrLn
```

Dieses Programm gibt einen String aus (mit dem Ergebnis `IO ()`) und wir schmeissen das Ergebnis weg und rufen einfach die nächsten Funktionen auf.

Monaden haben ähnlich zu `Functor` und `Applicative` auch mathematische Regeln, die eine Implementation erfüllen muss:

- Linksidentität

```
return x >>= f == f x
```

- Rechtsidentität

```
m >>= return == m
```

- Assoziativität

```
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Monaden haben ähnlich zu `Functor` und `Applicative` auch mathematische Regeln, die eine Implementation erfüllen muss:

- Linksidentität

```
return x >>= f == f x
```

- Rechtsidentität

```
m >>= return == m
```

- Assoziativität

```
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Die Assoziativität ist etwas schwer zu erkennen. Deutlicher wird es, wenn wir eine umgeformte Funktion definieren:

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

Somit ergibt sich für die Regeln:

```
return <=< f     == f
f <=< return     == f
(f <=< g) <=< h == f <=< (g <=< h)
```

Somit ergibt sich für die Regeln:

```
return <=< f      == f
f <=< return      == f
(f <=< g) <=< h == f <=< (g <=< h)
```

Diese Regeln sind relativ „natürlich“, da sie im Prinzip nur Funktionskomposition `(.)` auf Monaden wohldefinieren.

Wir können jede Monade auch als Monoid auffassen (mit `return` als neutralem Element und »= als Verknüpfung).

Wir können jede Monade auch als Monoid auffassen (mit `return` als neutralem Element und »= als Verknüpfung).

Wenn wir `return` als 1 sehen und »= als Multiplikation, dann haben manche Monaden auch eine 0 und eine Addition mit einem Distributivgesetz.

Hierfür gibt es die Typklasse `MonadPlus`:

```
class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Beispiel:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Beispiel:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Genaugenommen gibt es zwei Möglichkeiten für ein Distributivgesetz: Analog zu einem Körper (wie bei `[]`) oder mittels „Left Catch" (z.B. bei `Maybe`, `IO`, `STM`, ...).

Beispiel:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Genaugenommen gibt es zwei Möglichkeiten für ein Distributivgesetz: Analog zu einem Körper (wie bei `[]`) oder mittels „Left Catch“ (z.B. bei `Maybe`, `IO`, `STM`, ...).

Unterschied:

```
mplus mzero a = a
mplus a mzero = a
```

vs.

```
mplus mzero b = b
mplus a b     = a
```

Die do-notation hängt - wie wir eben gesehen haben - mit der Monade zusammen und ist über »= definiert.

Die do-notation hängt - wie wir eben gesehen haben - mit der Monade zusammen und ist über »= definiert.
Damit verhält sich diese Notation **für jede Monade anders**.

Die do-notation hängt - wie wir eben gesehen haben - mit der Monade zusammen und ist über »= definiert.
Damit verhält sich diese Notation **für jede Monade anders**.

Vorher hatten wir nur `do` im Kontext von `IO`. Daher gehen wir nun auf andere Monaden ein und wie hier die do-notation genutzt wird.

Die bereits bekannte List-Comprehension

```
let l = [x*y | x <- [1..5], y <- [1..5], x + y == 5]
```

ist nur syntaktischer Zucker für die monadische do-notation:

```
let l = do
        x <- [1..5]
        y <- [1..5]
        guard (x + y == 5)
        return x*y
```

mit

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

Häufig hat man das Problem, dass man einen Zustand in einem Programm herumreichen möchte.

Häufig hat man das Problem, dass man einen Zustand in einem Programm herumreichen möchte.

Hierzu gibt es 2 Möglichkeiten:

1. Den Zustand immer explizit an die Funktion übergeben
2. Den Zustand in einer Monade verstecken

Häufig hat man das Problem, dass man einen Zustand in einem Programm herumreichen möchte.

Hierzu gibt es 2 Möglichkeiten:

1. Den Zustand immer explizit an die Funktion übergeben
2. Den Zustand in einer Monade verstecken

Letzteres hat den Vorteil, dass man auch Funktionen aufwerten kann, die den Zustand ignorieren.

Die `State`-Monade macht keine „Magie“ im Hintergrund, sondern hält einfach nur einen Zustand fest und hat den Typen

```
State s a
```

wobei `s` der interne Zustand und `a` der Rückgabewert ist.

Die State-Monade macht keine „Magie“ im Hintergrund, sondern hält einfach nur einen Zustand fest und hat den Typen

```
State s a
```

wobei s der interne Zustand und a der Rückgabewert ist.

Um damit zu arbeiten gibt es u.A. folgende Hilfsfunktionen:

```
get    :: State s s
put    :: s -> State s ()
modify :: (s -> s) -> State s ()
```

jeweils um den internen Zustand zu holen, setzen oder zu modifizieren.

Beispiel:

```
countme :: a -> State Int a
countme a = do
              modify (+1)
              return a

example :: State Int Int
example = do
              x <- countme (2+2)
              y <- return (x*x)
              z <- countme (y-2)
              return z

examplemain = runState example 0
-- -> (14,2), 14 = wert von z, 2 = interner counter
```

Beispiel 2:

```
module Main where
import Control.Monad.State
type CountState = (Bool, Int)

startState :: CountState
startState = (False, 0)

play :: String -> State CountState Int
--play ...
```

```haskell
play :: String -> State CountState Int
play []      = do
               (_, score) <- get
               return score
play (x:xs) = do
 (on, score) <- get
 case x of
   'C' -> if on then put (on, score + 1) else put (on, score)
   'A' -> if on then put (on, score - 1) else put (on, score)
   'T' -> put (False, score)
   'G' -> put (True, score)
   _   -> put (on, score)
 play xs

main = print $ runState (play "GACAACTCGAAT") startState
-- -> (-3,(False,-3))
```