

Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

- 1 Wiederholung und Beispiele
- 2 Parsing
- 3 Arbeit am Beispiel
- 4 Parser-Funktionsweise

Die zentrale Struktur mit der wir uns im Folgenden beschäftigen ist

Die zentrale Struktur mit der wir uns im Folgenden beschäftigen ist

```
data V3 a = V3 a a a
```

ein dreidimensionaler Vektor.

Die zentrale Struktur mit der wir uns im Folgenden beschäftigen ist

```
data V3 a = V3 a a a
```

ein dreidimensionaler Vektor.

Wieso?

Die zentrale Struktur mit der wir uns im Folgenden beschäftigen ist

```
data V3 a = V3 a a a
```

ein dreidimensionaler Vektor.

Wieso?

- Sehr viele Algorithmen in der Informatik basieren auf vektoriellen Daten.

Die zentrale Struktur mit der wir uns im Folgenden beschäftigen ist

```
data V3 a = V3 a a a
```

ein dreidimensionaler Vektor.

Wieso?

- Sehr viele Algorithmen in der Informatik basieren auf vektoriellen Daten.
- Die Operationen für den 3D-Fall unterscheiden sich nur unwesentlich vom n D-Fall.

Die zentrale Struktur mit der wir uns im Folgenden beschäftigen ist

```
data V3 a = V3 a a a
```

ein dreidimensionaler Vektor.

Wieso?

- Sehr viele Algorithmen in der Informatik basieren auf vektoriellen Daten.
- Die Operationen für den 3D-Fall unterscheiden sich nur unwesentlich vom n D-Fall.
- Die Konzepte sind einigen von euch wahrscheinlich bereits bekannt, hier geht es erstmal so oder so nur um die Umsetzung.

Was für Operationen soll so ein Vektor unterstützen?

Was für Operationen soll so ein Vektor unterstützen?

- Skalare Multiplikation

Was für Operationen soll so ein Vektor unterstützen?

- Skalare Multiplikation
- Vektoraddition

Was für Operationen soll so ein Vektor unterstützen?

- Skalare Multiplikation
- Vektoraddition
- Skalarprodukt

Was für Operationen soll so ein Vektor unterstützen?

- Skalare Multiplikation
- Vektoraddition
- Skalarprodukt

```
mul  :: Num a => a -> V3 a -> V3 a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung von $(*x)$ auf jede Komponente
- Vektoraddition
- Skalarprodukt

```
mul  :: Num a => a -> V3 a -> V3 a
(*2) :: Num a => a -> a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Vektoraddition
- Skalarprodukt

`vmap :: (a -> a) -> V3 a -> V3 a`

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Vektoraddition
- Skalarprodukt

`vmap :: (a -> b) -> V3 a -> V3 b`

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Vektoraddition
- Skalarprodukt

```
vmap :: (a -> b) -> V3 a -> V3 b  
vadd :: Num a => V3 a -> V3 a -> V3 a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Vektoraddition
- Skalarprodukt

```
vmap :: (a -> b) -> V3 a -> V3 b  
vadd :: Num a => V3 a -> V3 a -> V3 a  
(+)  :: Num a => a -> a -> a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Anwendung einer Funktion mit zwei Argumenten auf zwei Vektoren
- Skalarprodukt

```
vmap    :: (a -> b) -> V3 a -> V3 b
vapply  :: (a -> a -> a) -> V3 a -> V3 a -> V3 a
(+)     :: Num a => a -> a -> a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Anwendung einer Funktion mit zwei Argumenten auf zwei Vektoren
- Skalarprodukt

```
vmap    :: (a -> b) -> V3 a -> V3 b  
vapply  :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
(+)     :: Num a => a -> a -> a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Anwendung einer Funktion mit zwei Argumenten auf zwei Vektoren
- Skalarprodukt

```
vmap    :: (a -> b) -> V3 a -> V3 b  
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
vdot    :: Num a => V3 a -> V3 a -> a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Anwendung einer Funktion mit zwei Argumenten auf zwei Vektoren
- Erst `vapply (*)`, dann Zusammenfassen mit `+`

```
vmap    :: (a -> b) -> V3 a -> V3 b
vapply  :: (a -> b -> c) -> V3 a -> V3 b -> V3 c
vdot    :: Num a => V3 a -> V3 a -> a
vcompress :: V3 a -> a
```

Was für Operationen soll so ein Vektor unterstützen?

- Anwendung derselben Funktion auf jede Komponente
- Anwendung einer Funktion mit zwei Argumenten auf zwei Vektoren
- Erst `vapply (*)`, dann Zusammenfassen mit `+`

```
vmap    :: (a -> b) -> V3 a -> V3 b
vapply  :: (a -> b -> c) -> V3 a -> V3 b -> V3 c
vdot    :: Num a => V3 a -> V3 a -> a
vfold   :: (a -> a -> a) -> V3 a -> a
```

`vmap :: (a -> b) -> V3 a -> V3 b`


```
vmap :: (a -> b) -> V3 a -> V3 b  
vmap f (V3 x y z) = ?
```

```
vmap :: (a -> b) -> V3 a -> V3 b  
vmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

```
vmap :: (a -> b) -> V3 a -> V3 b  
vmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

Dieses Muster haben wir schon häufiger gesehen.

```
vmap :: (a -> b) -> V3 a -> V3 b  
vmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

Dieses Muster haben wir schon häufiger gesehen.
Das ist ein Functor.

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

```
vmap :: (a -> b) -> V3 a -> V3 b  
vmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

Dieses Muster haben wir schon häufiger gesehen.
Das ist ein Functor.

```
class Functor f where  
    fmap :: (a -> b) -> f a -> f b  
  
instance Functor V3 where  
    fmap :: (a -> b) -> V3 a -> V3 b  
    fmap = vmap
```

```
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c
```

```
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
vapply f (V3 x y z) (V3 a b c) = ?
```

```
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
vapply f (V3 x y z) (V3 a b c) = V3 (f x a) (f y b) (f z c)
```



```
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
vapply f (V3 x y z) (V3 a b c) = V3 (f x a) (f y b) (f z c)
```

Wenn wir schon generisch sind: Was machen wir bei einer Funktion mit drei Argumenten?

```
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
vapply f (V3 x y z) (V3 a b c) = V3 (f x a) (f y b) (f z c)
```

Wenn wir schon generisch sind: Was machen wir bei einer Funktion mit drei Argumenten? Was bei vier Argumenten?

```
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
vapply f (V3 x y z) (V3 a b c) = V3 (f x a) (f y b) (f z c)
```

Wenn wir schon generisch sind: Was machen wir bei einer Funktion mit drei Argumenten? Was bei vier Argumenten? Was bei n Argumenten?

```
vapply :: (a -> b -> c) -> V3 a -> V3 b -> V3 c  
vapply f (V3 x y z) (V3 a b c) = V3 (f x a) (f y b) (f z c)
```

Wenn wir schon generisch sind: Was machen wir bei einer Funktion mit drei Argumenten? Was bei vier Argumenten? Was bei n Argumenten?

Funktionen sind *gecurried*. Wir brauchen immer nur „ein Argument mehr“ verarbeiten.

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (v3 1 2 3) (v3 4 5 6)
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)      (V3 4 5 6)  
          (V3 (1*) (2*) (3*)) (V3 4 5 6)
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)  
           (V3 (1*) (2*) (3*))  (V3 4 5 6)  
           (V3 (1*4) (2*5) (3*6))
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3   4   10   18)
```


Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3  4    10    18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3  4    10    18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

```
(*) :: Num a => a -> a -> a
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3  4    10    18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

```
(*) :: Num a => a -> (a -> a)
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3 4      10    18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

```
(*) :: Num a => a -> b
```

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

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3  4    10    18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

```
(*) :: Num a => a -> b
app :: (a -> a -> a) -> V3 a -> V3 (a -> a)

b = (a -> a)
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3  4    10    18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

```
(*) :: Num a => a -> b
app :: (a -> (a -> a)) -> V3 a -> V3 (a -> a)

b = (a -> a)
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3 4      10    18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

```
(*) :: Num a => a -> b
app :: (a -> b) -> V3 a -> V3 b

b = (a -> a)
```

Gehen wir das mal an einem Beispiel durch:

```
vapply (*) (V3 1 2 3)          (V3 4 5 6)
           (V3 (1*) (2*) (3*))  (V3 4 5 6)
           (V3 (1*4) (2*5) (3*6))
           (V3  4    10   18)
```

Wie können wir nun dieses V3 (1*) (2*) (3*) erzeugen?

```
(*) :: Num a => a -> b
fmap :: (a -> b) -> V3 a -> V3 b

b = (a -> a)
```



```
fmap (*) (V3 1 2 3) = V3 (1*) (2*) (3*)
```

V3 (1*) (2*) (3*) :: V3 (a -> a)

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
```

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> a) -> V3 a -> V3 a
```

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> b) -> V3 a -> V3 b
```

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> b) -> V3 a -> V3 b
```

Dies sollte uns auch bekannt vorkommen.

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> b) -> V3 a -> V3 b
```

Dies sollte uns auch bekannt vorkommen. Es ist ein `Applicative`.

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> b) -> V3 a -> V3 b
```

Dies sollte uns auch bekannt vorkommen. Es ist ein `Applicative`.

`Applicative` greift in unserem Beispiel auf `fmap` zurück um die Funktion in den Kontext zu bekommen und sofort anzuwenden.


```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> b) -> V3 a -> V3 b
```

Dies sollte uns auch bekannt vorkommen. Es ist ein `Applicative`.

`Applicative` greift in unserem Beispiel auf `fmap` zurück um die Funktion in den Kontext zu bekommen und sofort anzuwenden. Aber eigentlich könnten wir auch mit

```
V3 ((*) (*) (*)) :: V3 (a -> a -> a)
```

anfangen und 2x `apply` aufrufen.

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> b) -> V3 a -> V3 b
```

Dies sollte uns auch bekannt vorkommen. Es ist ein `Applicative`.

`Applicative` greift in unserem Beispiel auf `fmap` zurück um die Funktion in den Kontext zu bekommen und sofort anzuwenden. Aber eigentlich könnten wir auch mit

```
V3 ((*) (*) (*)) :: V3 (a -> a -> a)
```

anfangen und 2x `apply` aufrufen.

Hierfür brauchen wir noch eine Funktion

```
ins :: (a -> a -> a) -> V3 (a -> a -> a)
```

```
V3 (1*) (2*) (3*) :: V3 (a -> a)
V3 4 5 6          :: V3 a
apply :: V3 (a -> b) -> V3 a -> V3 b
```

Dies sollte uns auch bekannt vorkommen. Es ist ein `Applicative`.

`Applicative` greift in unserem Beispiel auf `fmap` zurück um die Funktion in den Kontext zu bekommen und sofort anzuwenden. Aber eigentlich könnten wir auch mit

```
V3 ((*) (*) (*)) :: V3 (a -> a -> a)
```

anfangen und 2x `apply` aufrufen.

Hierfür brauchen wir noch eine Funktion

```
ins :: b -> V3 b
```

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

Somit haben wir alles für unser Applicative beisammen:

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

Somit haben wir alles für unser Applicative beisammen:

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

Die Instanz für unser V3 ist auch schnell geschrieben:

```
instance Applicative V3 where
  pure f      = undefined
  vf <*> vx = undefined
```

Somit haben wir alles für unser Applicative beisammen:

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

Die Instanz für unser V3 ist auch schnell geschrieben:

```
instance Applicative V3 where
  pure f      = V3 f f f
  vf <*> vx = undefined
```

Somit haben wir alles für unser Applicative beisammen:

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

Die Instanz für unser V3 ist auch schnell geschrieben:

```
instance Applicative V3 where
  pure f                = V3 f f f
  (V3 f g h) (*>) (V3 x y z) = undefined
```

Somit haben wir alles für unser Applicative beisammen:

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

Die Instanz für unser V3 ist auch schnell geschrieben:

```
instance Applicative V3 where
  pure f = V3 f f f
  (V3 f g h) (*>) (V3 x y z) = V3 (f x) (g y) (h z)
```


Somit haben wir alles für unser Applicative beisammen:

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

Die Instanz für unser V3 ist auch schnell geschrieben:

```
instance Applicative V3 where
  pure f = V3 f f f
  (V3 f g h) (*>) (V3 x y z) = V3 (f x) (g y) (h z)
```

Somit fällt unsere Vektoraddition zusammen, und zwar auf:

```
vadd x y = pure (+) (*>) x (*>) y
```

Somit haben wir alles für unser Applicative beisammen:

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

Die Instanz für unser V3 ist auch schnell geschrieben:

```
instance Applicative V3 where
  pure f = V3 f f f
  (V3 f g h) (*>) (V3 x y z) = V3 (f x) (g y) (h z)
```

Somit fällt unsere Vektoraddition zusammen, und zwar auf:

```
vadd x y = (fmap (+) x) (*>) y
```

Somit haben wir alles für unser Applicative beisammen:

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

Die Instanz für unser V3 ist auch schnell geschrieben:

```
instance Applicative V3 where
  pure f = V3 f f f
  (V3 f g h) <*> (V3 x y z) = V3 (f x) (g y) (h z)
```

Somit fällt unsere Vektoraddition zusammen, und zwar auf:

```
vadd x y = (+) <$> x <*> y
```

Wozu braucht man `Applicative` noch?

Wozu braucht man Applicative noch?
Nehmen wir an wir hätten

```
data Person = Person { name :: String
                        , age  :: Int
                        , zip  :: Int
                        , gen  :: Gender
                        }
```

Wozu braucht man Applicative noch?
Nehmen wir an wir hätten

```
data Person = Person { name :: String
                        , age  :: Int
                        , zip  :: Int
                        , gen  :: Gender
                        }
```

und

```
let names = V3 "Alice" "Bob" "Charlie"
let ages  = V3 20 30 40
let zips  = V3 33333 44444 55555
let gens  = V3 Female Male Other
```

dann können wir einfach

```
let all = Person <$> names  
           <*> ages  
           <*> zips  
           <*> gens
```

machen

dann können wir einfach

```
let all = Person <$> names
           <*> ages
           <*> zips
           <*> gens
```

machen und erhalten:

```
all = V3 (Person "Alice"    20 33333 Female)
         (Person "Bob"      30 44444 Male)
         (Person "Charlie"  40 55555 Other)
```


Dieser Fall ist für den V3 etwas konstruiert.

Dieser Fall ist für den V3 etwas konstruiert.
Ein Beispiel aus einer echten Applikation wäre:

```
personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm = renderDivs $ Person
  <$> areq textField "Name" Nothing
  <*> areq (jqueryDayField def
    { jdsChangeYear = True           -- give a year dropdown
    , jdsYearRange = "1900:-5"      -- 1900 till five years ago
    }) "Birthday" Nothing
  <*> aopt textField "Favorite color" Nothing
  <*> areq emailField "Email address" Nothing
  <*> aopt urlField "Website" Nothing
```

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a
vdot x y = _ ((*) <$> x <*> y)
```

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a  
vdot x y = _ ((*) <$> x <*> y)
```

Wir brauchen noch eine Funktion

```
compress :: (a -> a -> a) -> V3 a -> a
```

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a
vdot x y = _ ((*) <$> x <*> y)
```

Wir brauchen noch eine Funktion

```
compress :: (a -> a -> a) -> V3 a -> a
compress f (V3 x y z) = ?
```

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a  
vdot x y = _ ((*) <$> x <*> y)
```

Wir brauchen noch eine Funktion

```
compress :: (a -> a -> a) -> V3 a -> a  
compress f (V3 x y z) = f (f x y) z
```

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a  
vdot x y = compress (+) ((*) <$> x <*> y)
```

Wir brauchen noch eine Funktion

```
compress :: (a -> a -> a) -> V3 a -> a  
compress f (V3 x y z) = f (f x y) z
```

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a  
vdot x y = compress (+) ((*) <$> x <*> y)
```

Wir brauchen noch eine Funktion

```
compress :: (a -> a -> a) -> V3 a -> a  
compress f (V3 x y z) = f (f x y) z
```

Dies ist ein Spezialfall des generischen „Faltens“ von Datestrukturen.

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a  
vdot x y = compress (+) ((*) <$> x <*> y)
```

Wir brauchen noch eine Funktion

```
compress :: (a -> a -> a) -> V3 a -> a  
compress f (V3 x y z) = f (f x y) z
```

Dies ist ein Spezialfall des generischen „Faltens“ von Datestrukturen.

Man kann sich einen Vektor auch als Liste mit drei Elementen vorstellen und auf Listen kennen einige von euch vielleicht bereits `foldl`, `foldr`, ...

Kommen wir aber nun zurück zu unserem V3:

Wir möchten noch ein Skalarprodukt definieren. Die Hälfte haben wir schon:

```
vdot :: V3 a -> V3 a -> a  
vdot x y = compress (+) ((*) <$> x <*> y)
```

Wir brauchen noch eine Funktion

```
compress :: (a -> a -> a) -> V3 a -> a  
compress f (V3 x y z) = f (f x y) z
```

Dies ist ein Spezialfall des generischen „Faltens“ von Datestrukturen.

Man kann sich einen Vektor auch als Liste mit drei Elementen vorstellen und auf Listen kennen einige von euch vielleicht bereits `foldl`, `foldr`, ...

Also wäre es besser, wenn unser V3 in dieser generischen Klasse sitzt, als wenn jeder Nutzer nachsehen muss, was nun `compress` genau tut.

Was heisst dieses `foldr` nochmal?

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) 0 [1,2,3,4]
```

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) (4+0) [1,2,3]
```

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) 4 [1,2,3]
```

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

`foldr (+) (3+4) [1,2]`

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) 7 [1,2]
```


Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) (2+7) [1]
```

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) 9 [1]
```

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) (1+9) []
```

Was heisst dieses `foldr` nochmal?

Sehen wir uns ein Beispiel an:

```
foldr (+) 10 []  
=> 10
```

Die Typklasse hierzu heisst Foldable:

```
class Foldable f where
  foldr :: (a -> b -> b) -> b -> f a -> b
oder
  foldMap :: Monoid m => (a -> m) -> f a -> m
```

Die Typklasse hierzu heisst Foldable:

```
class Foldable f where
  foldr :: (a -> b -> b) -> b -> f a -> b
oder
  foldMap :: Monoid m => (a -> m) -> f a -> m
```

Beide Definitionen sind gleichwertig.

Die Typklasse hierzu heisst Foldable:

```
class Foldable f where
  foldr :: (a -> b -> b) -> b -> f a -> b
oder
  foldMap :: Monoid m => (a -> m) -> f a -> m
```

Beide Definitionen sind gleichwertig.

```
instance Foldable V3 where
  foldr f s (V3 x y z) = f x $ f y $ f z s
```

Die Typklasse hierzu heisst Foldable:

```
class Foldable f where
  foldr :: (a -> b -> b) -> b -> f a -> b
oder
  foldMap :: Monoid m => (a -> m) -> f a -> m
```

Beide Definitionen sind gleichwertig.

```
instance Foldable V3 where
  foldr f s (V3 x y z) = f x $ f y $ f z s
oder
instance Foldable V3 where
  foldMap f (V3 x y z) = f x 'mappend' f y 'mappend' f z
```


Die Typklasse hierzu heisst Foldable:

```
class Foldable f where
  foldr :: (a -> b -> b) -> b -> f a -> b
oder
  foldMap :: Monoid m => (a -> m) -> f a -> m
```

Beide Definitionen sind gleichwertig.

```
instance Foldable V3 where
  foldr f s (V3 x y z) = f x $ f y $ f z s
```

oder

```
instance Foldable V3 where
  foldMap f (V3 x y z) = f x 'mappend' f y 'mappend' f z
```

Diese Typklasse definiert dann zahlreiche weitere Funktionen. Unter anderem auch

```
foldr1 :: (a -> a -> a) -> f a -> a
```

welches einfach nur foldr mit dem ersten Element als Startwert ist.

Wozu brauchen wir diese Abstraktion?

Wozu brauchen wir diese Abstraktion?

Zunächst haben wir für unser Skalarprodukt

```
vdot x y = foldr1 (+) ((*) <$> x <*> y)
```

Wozu brauchen wir diese Abstraktion?

Zunächst haben wir für unser Skalarprodukt

`vdot x y = foldr1 (+) ((*) <$> x <*> y)`

Aber wenn wir jetzt z.B. die Norm berechnen wollen, müssen wir die Wurzel ziehen.

Wozu brauchen wir diese Abstraktion?

Zunächst haben wir für unser Skalarprodukt

```
vdot x y = foldr1 (+) ((*) <$> x <*> y)
```

Aber wenn wir jetzt z.B. die Norm berechnen wollen, müssen wir die Wurzel ziehen. Diese ist z.B. auf `Int` nicht definiert.

Wozu brauchen wir diese Abstraktion?

Zunächst haben wir für unser Skalarprodukt

```
vdot x y = foldr1 (+) ((*) <$> x <*> y)
```

Aber wenn wir jetzt z.B. die Norm berechnen wollen, müssen wir die Wurzel ziehen. Diese ist z.B. auf `Int` nicht definiert. Somit können wir *nicht* schreiben:

```
vnorm x y = sqrt $ foldr1 (+) ((*) <$> x <*> y)
```

Wozu brauchen wir diese Abstraktion?

Zunächst haben wir für unser Skalarprodukt

```
vdot x y = foldr1 (+) ((*) <$> x <*> y)
```

Aber wenn wir jetzt z.B. die Norm berechnen wollen, müssen wir die Wurzel ziehen. Diese ist z.B. auf `Int` nicht definiert. Somit können wir *nicht* schreiben:

```
vnorm x y = sqrt $ foldr1 (+) ((*) <$> x <*> y)
```

Wir müssen also irgendwo die Umwandlung von `Int` \rightarrow `Double` machen.

Wozu brauchen wir diese Abstraktion?

Zunächst haben wir für unser Skalarprodukt

```
vdot x y = foldr1 (+) ((*) <$> x <*> y)
```

Aber wenn wir jetzt z.B. die Norm berechnen wollen, müssen wir die Wurzel ziehen. Diese ist z.B. auf `Int` nicht definiert. Somit können wir *nicht* schreiben:

```
vnorm x y = sqrt $ foldr1 (+) ((*) <$> x <*> y)
```

Wir müssen also irgendwo die Umwandlung von `Int` \rightarrow `Double` machen.

Dazu einige Beispiele:

```
vnorm x y = sqrt $ fromIntegral $ foldr1 (+) ((*) <$> x <*> y)
```

```
vnorm x y = sqrt $ foldr ((*) <$> x <*> y) 0  
                (foldr (+) . fromIntegral)
```

```
vnorm x y = sqrt $ foldr1 (+) ((*) <$> (fromIntegral <$> x)  
                                     <*> (fromIntegral <$> y))
```


Wozu brauchen wir diese Abstraktion?

Zunächst haben wir für unser Skalarprodukt

```
vdot x y = foldr1 (+) ((*) <$> x <*> y)
```

Aber wenn wir jetzt z.B. die Norm berechnen wollen, müssen wir die Wurzel ziehen. Diese ist z.B. auf `Int` nicht definiert. Somit können wir *nicht* schreiben:

```
vnorm x y = sqrt $ foldr1 (+) ((*) <$> x <*> y)
```

Wir müssen also irgendwo die Umwandlung von `Int` \rightarrow `Double` machen.

Dazu einige Beispiele:

```
vnorm x y = sqrt $ fromIntegral $ foldr1 (+) ((*) <$> x <*> y)
```

```
vnorm x y = sqrt $ foldr ((+) . fromIntegral) 0  
                                     ((*) <$> x <*> y)
```

```
vnorm x y = sqrt $ foldr1 (+) ((*) <$> (fromIntegral <$> x)  
                                     <*> (fromIntegral <$> y))
```

Diese Funktionen unterscheiden sich in der Laufzeit - und je nach Datentyp auch im Ergebnis.

```

module V3 where
import Control.Applicative
import Data.Foldable
import Data.Monoid
import Prelude hiding (foldr1)

data V3 a = V3 a a a
instance Functor V3 where
    fmap f (V3 x y z) = V3 (f x) (f y) (f z)
instance Applicative V3 where
    pure f = V3 f f f
    (V3 f g h) <*> (V3 x y z) = V3 (f x) (g y) (h z)
instance Foldable V3 where
    foldMap f (V3 x y z) = f x 'mappend' f y 'mappend' f z

vmul s x = (s*) <$> x
vadd x y = (+) <$> x <*> y
vdot x y = foldr1 (+) $ (*) <$> x <*> y
vnorm x y = sqrt $ vdot x y
vnorm' x y = sqrt . fromIntegral $ vdot x y
    
```

Interessant wird es, wenn wir uns die Typen von unseren Funktionen geben lassen:

```
vmul   :: (Num b, Functor f)
        => b -> f b -> f b
vadd   :: (Applicative f, Num b)
        => f b -> f b -> f b
vdot   :: (Foldable t, Applicative t, Num a)
        => t a -> t a -> a
vnorm  :: (Foldable t, Applicative t, Floating s)
        => t s -> t s -> s
vnorm' :: (Foldable t, Applicative t, Integral s, Floating c)
        => t s -> t s -> c
```

Interessant wird es, wenn wir uns die Typen von unseren Funktionen geben lassen:

```
vmul  :: (Num b, Functor f)
      => b -> f b -> f b
vadd  :: (Applicative f, Num b)
      => f b -> f b -> f b
vdot  :: (Foldable t, Applicative t, Num a)
      => t a -> t a -> a
vnorm :: (Foldable t, Applicative t, Floating s)
      => t s -> t s -> s
vnorm':: (Foldable t, Applicative t, Integral s, Floating c)
      => t s -> t s -> c
```

Hier ist gar nicht mehr die Rede von V3.

Interessant wird es, wenn wir uns die Typen von unseren Funktionen geben lassen:

```
vmul   :: (Num b, Functor f)
        => b -> f b -> f b
vadd   :: (Applicative f, Num b)
        => f b -> f b -> f b
vdot   :: (Foldable t, Applicative t, Num a)
        => t a -> t a -> a
vnorm  :: (Foldable t, Applicative t, Floating s)
        => t s -> t s -> s
vnorm' :: (Foldable t, Applicative t, Integral s, Floating c)
        => t s -> t s -> c
```

Hier ist gar nicht mehr die Rede von V3.

Somit können wir dieselben Funktionen auch für V2, V4 etc. benutzen, wenn wir Functor, Applicative und Foldable definieren.

Interessant wird es, wenn wir uns die Typen von unseren Funktionen geben lassen:

```
vmul   :: (Num b, Functor f)
        => b -> f b -> f b
vadd   :: (Applicative f, Num b)
        => f b -> f b -> f b
vdot   :: (Foldable t, Applicative t, Num a)
        => t a -> t a -> a
vnorm  :: (Foldable t, Applicative t, Floating s)
        => t s -> t s -> s
vnorm' :: (Foldable t, Applicative t, Integral s, Floating c)
        => t s -> t s -> c
```

Hier ist gar nicht mehr die Rede von V3.

Somit können wir dieselben Funktionen auch für V2, V4 etc. benutzen, wenn wir Functor, Applicative und Foldable definieren.

Damit läuft das „Programmieren“ nicht mehr auf das erneute Implementieren, sondern nur auf das Instanzieren hinaus.

Wozu das ganze?

In der „echten Welt“ haben wir häufig Eingabedaten in verschiedensten Formaten:

- Text (z.B. Config-Files, Log-Files)
- JSON (z.B. im Web-Kontext)
- XML (z.B. (X)HTML)
- Binärcode (z.B. 3D-Modelle, Netzwerkcode, ...)

Wozu das ganze?

In der „echten Welt“ haben wir häufig Eingabedaten in verschiedensten Formaten:

- Text (z.B. Config-Files, Log-Files)
- JSON (z.B. im Web-Kontext)
- XML (z.B. (X)HTML)
- Binärcode (z.B. 3D-Modelle, Netzwerkcode, ...)

Diese wollen wir nun in Haskell nutzbar machen, um sie aufzubereiten, zu filtern oder generell weiterzuverarbeiten.

Wie gehen wir das Ganze an?

Naiv: Textvergleiche, Patterns und reguläre Ausdrücke.

Wie gehen wir das Ganze an?

Naiv: Textvergleiche, Patterns und reguläre Ausdrücke.

This is not the Haskell way to do that!

Wie gehen wir das Ganze an?

Naiv: Textvergleiche, Patterns und reguläre Ausdrücke.

This is not the Haskell way to do that!

In Haskell möchten wir gerne **kleine Teilprobleme lösen** (wie z.B. das Lesen eines Zeichens oder einer Zahl) und diese dann zu größeren Lösungen **kombinieren**.

Kernstück für das stückweise Parsen bildet die Typklasse
Applicative mit ihrer Erweiterung „Alternative“.

Kernstück für das stückweise Parsen bildet die Typklasse `Applicative` mit ihrer Erweiterung „Alternative“.

Was heisst das genau?

Kernstück für das stückweise Parsen bildet die Typklasse `Applicative` mit ihrer Erweiterung „Alternative“.

Was heisst das genau?

„Alternative“ ist in der Lage viele Dinge zu probieren und dann das erste zurückzuliefern, was geklappt hat.

So ist man in der Lage verschiedene Möglichkeiten des weiter-parsens auszudrücken.

Kommen wir zunächst zu einem Beispiel. Gegeben ist folgendes Log, welches in Haskell übersetzt werden soll:

```
2013-06-29 11:16:23 124.67.34.60 keyboard
2013-06-29 11:32:12 212.141.23.67 mouse
2013-06-29 11:33:08 212.141.23.67 monitor
2013-06-29 12:12:34 125.80.32.31 speakers
2013-06-29 12:51:50 101.40.50.62 keyboard
2013-06-29 13:10:45 103.29.60.13 mouse
```

Wir haben hier eine Liste von Daten, IP-Adressen und Geräten, die irgendwie interagiert haben.

Zunächst schauen wir uns den Aufbau einer Zeile an

2013-06-29 11:16:23 124.67.34.60 keyboard

Zunächst schauen wir uns den Aufbau einer Zeile an

2013-06-29 11:16:23 124.67.34.60 keyboard

Sie besteht aus

- 1 einem Datum (YYYY-MM-DD hh:mm:ss)
- 2 einer IP (0.0.0.0 - 255.255.255.255)
- 3 einem Gerät (String als Identifier)

Für alles definieren wir nun unsere Wunsch-Datenstrukturen:

Für alles definieren wir nun unsere Wunsch-Datenstrukturen:

Eine Zeile lässt sich darstellen als

```
data LogZeile = LogZeile Datum IP Geraet
```

Und das gesamte Log als viele Zeilen:

```
data Log = Log [LogZeile]
```

Datum (YYYY-MM-DD hh:mm:ss) können wir darstellen als

```
import Data.Time
```

```
data Datum = Datum  
    { tag    :: Day  
    , zeit  :: TimeOfDay  
    } deriving (Show, Eq)
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)  
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Eine IP (0.0.0.0 - 255.255.255.255) ist darstellbar als

```
import Data.Word
```

```
data IP = IP Word8 Word8 Word8 Word8 deriving (Show,Eq)
```

```
> IP 13 37 13 37
```

```
IP 13 37 13 37
```

mit Word8 als unsigned 8-Bit-Integer.

Eine IP (0.0.0.0 - 255.255.255.255) ist darstellbar als

```
import Data.Word
```

```
data IP = IP Word8 Word8 Word8 Word8 deriving (Show,Eq)
```

```
> IP 13 37 13 37
```

```
IP 13 37 13 37
```

mit Word8 als unsigned 8-Bit-Integer.

Wenn „inkorrekte“ Werte nicht darstellbar sind, dann haben wir sie auch nicht in unserem Programm als Problem.

Ein Gerät (String als Identifier) können wir nur als Solches definieren:

```
data Geraet = Mouse
           | Keyboard
           | Monitor
           | Speakers
           deriving (Show,Eq)
```

Ein Gerät (String als Identifier) können wir nur als Solches definieren:

```
data Geraet = Mouse
           | Keyboard
           | Monitor
           | Speakers
           deriving (Show, Eq)
```

Hier können wir nachher bei Bedarf auch schnell welche hinzufügen und der Compiler meckert dann an allen Stellen herum, wo wir nicht mehr alle Fälle abfangen.


```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Mit der Bibliothek attoparsec liest sich der Code fast von allein:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Time
import Data.Attoparsec.Char8

zeitParser :: Parser Datum
zeitParser = do
    undefined
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Mit der Bibliothek attoparsec liest sich der Code fast von allein:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Time
import Data.Attoparsec.Char8

zeitParser :: Parser Datum
zeitParser = do
  y <- count 4 digit
  undefined
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)  
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Mit der Bibliothek attoparsec liest sich der Code fast von allein:

```
{-# LANGUAGE OverloadedStrings #-}  
import Data.Time  
import Data.Attoparsec.Char8  
  
zeitParser :: Parser Datum  
zeitParser = do  
  y <- count 4 digit  
  char '-'  
  undefined
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)  
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Mit der Bibliothek attoparsec liest sich der Code fast von allein:

```
{-# LANGUAGE OverloadedStrings #-}  
import Data.Time  
import Data.Attoparsec.Char8  
  
zeitParser :: Parser Datum  
zeitParser = do  
  y <- count 4 digit  
  char '-'  
  mm <- count 2 digit  
  char '-'  
  undefined
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)  
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Mit der Bibliothek attoparsec liest sich der Code fast von allein:

```
{-# LANGUAGE OverloadedStrings #-}  
import Data.Time  
import Data.Attoparsec.Char8  
  
zeitParser :: Parser Datum  
zeitParser = do  
  y <- count 4 digit; char '-'  
  mm <- count 2 digit; char '-'  
  undefined
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Mit der Bibliothek attoparsec liest sich der Code fast von allein:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Time
import Data.Attoparsec.Char8

zeitParser :: Parser Datum
zeitParser = do
  y  <- count 4 digit; char '-'
  mm <- count 2 digit; char '-'
  d  <- count 2 digit; char ' '
  h  <- count 2 digit; char ':'
  m  <- count 2 digit; char ':'
  s  <- count 2 digit
  undefined
```

```
> Datum (fromGregorian 2014 1 2) (TimeOfDay 13 37 0)
Datum {tag = 2014-01-02, zeit = 13:37:00}
```

Mit der Bibliothek attoparsec liest sich der Code fast von allein:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Time
import Data.Attoparsec.Char8

zeitParser :: Parser Datum
zeitParser = do
  y  <- count 4 digit; char '-'
  mm <- count 2 digit; char '-'
  d  <- count 2 digit; char ' '
  h  <- count 2 digit; char ':'
  m  <- count 2 digit; char ':'
  s  <- count 2 digit
  return $
    Datum { tag  = fromGregorian (read y) (read mm) (read d)
          , zeit = TimeOfDay (read h) (read m) (read s)
          }
```


Für die IP sieht der Code ähnlich aus:

```
{-# LANGUAGE OverloadedStrings #-}  
import Data.Attoparsec.Char8  
import Data.Word  
  
parseIP :: Parser IP  
parseIP = do  
    d1 <- decimal  
    char ','  
    d2 <- decimal  
    char ','  
    d3 <- decimal  
    char ','  
    d4 <- decimal  
    return $ IP d1 d2 d3 d4
```

Für das Gerät brauchen wir die Mächtigkeit von Alternative:

```
{-# LANGUAGE OverloadedStrings #-}  
import Data.Attoparsec.Char8  
import Control.Applicative  
  
geraetParser :: Parser Geraet  
geraetParser =  
    (string "mouse"    >> return Mouse)  
  <|> (string "keyboard" >> return Keyboard)  
  <|> (string "monitor"  >> return Monitor)  
  <|> (string "speakers" >> return Speakers)
```

Für das Gerät brauchen wir die Mächtigkeit von Alternative:

```
{-# LANGUAGE OverloadedStrings #-}  
import Data.Attoparsec.Char8  
import Control.Applicative  
  
geraetParser :: Parser Geraet  
geraetParser =  
    (string "mouse"    >> return Mouse)  
  <|> (string "keyboard" >> return Keyboard)  
  <|> (string "monitor"  >> return Monitor)  
  <|> (string "speakers" >> return Speakers)
```

<|> ist der alternativ-Operator, der das Linke ausführt und bei einem Fehler das Rechte zurückgibt.

Für das Gerät brauchen wir die Mächtigkeit von Alternative:

```
{-# LANGUAGE OverloadedStrings #-}  
import Data.Attoparsec.Char8  
import Control.Applicative  
  
geraetParser :: Parser Geraet  
geraetParser =  
    (string "mouse"    >> return Mouse)  
  <|> (string "keyboard" >> return Keyboard)  
  <|> (string "monitor"  >> return Monitor)  
  <|> (string "speakers" >> return Speakers)
```

<|> ist der alternativ-Operator, der das Linke ausführt und bei einem Fehler das Rechte zurückgibt.

Wir matchen hier jeweils auf den String, schmeissen ihn weg und liefern unseren Datentyp zurück.

Für die gesamte Zeile packen wir einfach unsere Parser zusammen:

```
zeilenParser :: Parser LogZeile
zeilenParser = do
    datum <- zeitParser
    char ' '
    ip <- parseIP
    char ' '
    geraet <- geraetParser
    return $ LogZeile datum ip geraet
```

Für das Log nehmen wir die many-Funktion aus „Alternative“:

```
logParser :: Parser Log  
logParser = many $ zeilenParser <* endOfLine
```

Für das Log nehmen wir die many-Funktion aus „Alternative“:

```
logParser :: Parser Log  
logParser = many $ zeilenParser <* endOfLine
```

<* (aus Applicative) fungiert hier als ein Parser-Kombinator, der erst links matched, dann rechts matched und dann das Ergebnis des Linken zurückliefert.

Für das Log nehmen wir die many-Funktion aus „Alternative“:

```
logParser :: Parser Log  
logParser = many $ zeilenParser <* endOfLine
```

<* (aus Applicative) fungiert hier als ein Parser-Kombinator, der erst links matched, dann rechts matched und dann das Ergebnis des Linken zurückliefert.

Hier verwenden wir diesen um das Zeilenende hinter jeder Zeile loszuwerden.

Nun schreiben wir ein kleines Testprogramm:

```
main :: IO ()  
main = do  
    log <- B.readFile "log.txt"  
    print $ parseOnly logParser log
```

und führen es aus:

```
Right [LogZeile (Datum {tag = 2013-06-29, zeit = 11:16:23}) (IP 124 67 34 60) Keyboard,  
      LogZeile (Datum {tag = 2013-06-29, zeit = 11:32:12}) (IP 212 141 23 67) Mouse,  
      LogZeile (Datum {tag = 2013-06-29, zeit = 11:33:08}) (IP 212 141 23 67) Monitor,  
      LogZeile (Datum {tag = 2013-06-29, zeit = 12:12:34}) (IP 125 80 32 31) Speakers,  
      LogZeile (Datum {tag = 2013-06-29, zeit = 12:51:50}) (IP 101 40 50 62) Keyboard,  
      LogZeile (Datum {tag = 2013-06-29, zeit = 13:10:45}) (IP 103 29 60 13) Mouse]
```

Wie funktioniert so ein Parser nun intern genau?

Wie funktioniert so ein Parser nun intern genau?

Wir werden im folgenden attoparsec nicht im Detail erklären, sondern eher die Idee, die dahinter steht.

Eigentlich hätten wir gerne eine Funktion

```
parse :: String -> a
```

Eigentlich hätten wir gerne eine Funktion

```
parse :: String -> a
```

Aber was machen wir, wenn etwas von dem String überbleibt, damit wir später weitere Dinge parsen können?

Also brauchen wir

```
parse :: String -> (a, String)
```

Also brauchen wir

```
parse :: String -> (a, String)
```

Aber das Parsen kann auch Fehlschlagen.

Also brauchen wir

```
parse :: String -> (Maybe a, String)
```

Und Fehlermeldungen wären auch schön.

Also brauchen wir

```
parse :: String -> (Either String a, String)
```

```
parse :: String -> (Either String a, String)
```

Da wir auch so elegant die do-Notation nutzen möchten, brauchen wir eine Monade. Dazu brauchen wir einen einzelnen Typen:

```
parse :: String -> (Either String a, String)
```

Da wir auch so elegant die do-Notation nutzen möchten, brauchen wir eine Monade. Dazu brauchen wir einen einzelnen Typen:

```
data Parser a =  
  Parser {  
    runParser :: String -> (Either String a, String)  
  }
```

```
parse :: String -> (Either String a, String)
```

Da wir auch so elegant die `do`-Notation nutzen möchten, brauchen wir eine `Monad`. Dazu brauchen wir einen einzelnen `Type`:

```
data Parser a =  
  Parser {  
    runParser :: String -> (Either String a, String)  
  }
```

Dies gibt uns (wie schon neulich bei `State`) zwei Funktionen:

```
Parser    :: (String -> (Either String a, String)) -> Parser a  
runParser :: Parser a -> String -> (Either String a, String)
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)

instance Functor Parser where
fmap f p = Parser $ \input -> undefined
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Functor Parser where
fmap f p = Parser $ \input ->
    let
        (res, rem) = runParser p input
    in
        undefined
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Functor Parser where
fmap f p = Parser $ \input ->
    let
        (res, rem) = runParser p input
    in
        (fmap f res, rem)
```

```
Parser      :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)

instance Applicative Parser where
  pure a      = Parser $ \input -> undefined
  pf <*> pa = undefined
```



```
Parser      :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)

instance Applicative Parser where
  pure a      = Parser $ \input -> (Right a,input)
  pf <*> pa = undefined
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)

instance Applicative Parser where
  pure a      = Parser $ \input -> (Right a,input)
  pf <*> pa = Parser $ \input ->
                        undefined
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Applicative Parser where
  pure a      = Parser $ \input -> (Right a,input)
  pf <*> pa = Parser $ \input ->
    let
      (rf, rem1) = runParser pf input
      (ra, rem2) = runParser pa rem1
    in
      undefined
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Applicative Parser where
  pure a      = Parser $ \input -> (Right a,input)
  pf <*> pa = Parser $ \input ->
    let
      (rf, rem1) = runParser pf input
      (ra, rem2) = runParser pa rem1
    in
      (rf <*> ra, rem2)
```

```
Parser      :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)

instance Monad Parser where
    return a  = pure a
    pa >>= f  = Parser $ \input ->
                    undefined
```

```
Parser      :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Monad Parser where
  return a  = pure a
  pa >>= f  = Parser $ \input ->
    let
      (ra, rem1) = runParser pa input
    in
      undefined
```

```
Parser      :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Monad Parser where
  return a  = pure a
  pa >>= f  = Parser $ \input ->
    let
      (ra, rem1) = runParser pa input
    in
      case ra of
        Left err  -> undefined
        Right a   -> undefined
```

```
Parser      :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Monad Parser where
  return a  = pure a
  pa >>= f  = Parser $ \input ->
    let
      (ra, rem1) = runParser pa input
    in
      case ra of
        Left err  -> (Left err, rem1)
        Right a   -> undefined
```



```
Parser      :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Monad Parser where
  return a  = pure a
  pa >>= f  = Parser $ \input ->
    let
      (ra, rem1) = runParser pa input
    in
      case ra of
        Left err -> (Left err, rem1)
        Right a  -> runParser (f a) rem1
```

Was fehlt uns noch?

Was fehlt uns noch?

- Langweilige Definition simpler Parser (digit, lit, space, ...)

Was fehlt uns noch?

- Langweilige Definition simpler Parser (digit, lit, space, ...)
- Erweiterung mit sog. Continuations (Text „nachschieben“)

Was fehlt uns noch?

- Langweilige Definition simpler Parser (digit, lit, space, ...)
- Erweiterung mit sog. Continuations (Text „nachschieben“)
- Implementation einer Standardfehlerbehandlung

Was fehlt uns noch?

- Langweilige Definition simpler Parser (digit, lit, space, ...)
- Erweiterung mit sog. Continuations (Text „nachschieben“)
- Implementation einer Standardfehlerbehandlung

Wir kümmern uns zunächst um das Letzgenannte, indem wir Alternative implementieren.

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)

instance Alternative Parser where
    empty      = Parser $ \input -> undefined
    pa <|> pb = undefined
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)

instance Alternative Parser where
    empty      = Parser $ \inp -> (Left "",inp)
    pa <|> pb = undefined
```



```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Alternative Parser where
  empty      = Parser $ \inp -> (Left "",inp)
  pa <|> pb = Parser $ \input ->
    let
      (ra, rem1) = runParser pa input
    in
      undefined
```

```
Parser    :: (String -> (Either String a, String)) -> Parser a
runParser :: Parser a -> String -> (Either String a, String)
```

```
instance Alternative Parser where
  empty      = Parser $ \inp -> (Left "",inp)
  pa <|> pb = Parser $ \input ->
    let
      (ra, rem1) = runParser pa input
    in
      case ra of
        Right _ -> (ra, rem1)
        _       -> runParser pb input
```

Schlägt der Parser hier fehl, dann wird der bereits benutzte Input wiederhergestellt und ein weiterer Parser ausprobiert.

Nun können wir ein paar Simple Dinge parsen:

```
parse1 :: Parser Int
parse1 = Parser $ \inp -> case inp of
    ('1':xs) -> (Right 1,xs)
    x         -> (Left "no 1", x)
```

```
parse0 :: Parser Int
parse0 = Parser $ \inp -> case inp of
    ('0':xs) -> (Right 0, xs)
    x         -> (Left "no 0", x)
```

Nun können wir ein paar Simple Dinge parsen:

```
parse1 :: Parser Int
parse1 = Parser $ \inp -> case inp of
    ('1':xs) -> (Right 1,xs)
    x         -> (Left "no 1", x)
```

```
parse0 :: Parser Int
parse0 = Parser $ \inp -> case inp of
    ('0':xs) -> (Right 0, xs)
    x         -> (Left "no 0", x)
```

Dies sind dann intrinsics oder interne Funktionen, die ohne einen Zugriff auf „Parser“ nicht möglich sind.

Nun können wir ein paar Simple Dinge parsen:

```
parse1 :: Parser Int
parse1 = Parser $ \inp -> case inp of
    ('1':xs) -> (Right 1,xs)
    x         -> (Left "no 1", x)
```

```
parse0 :: Parser Int
parse0 = Parser $ \inp -> case inp of
    ('0':xs) -> (Right 0, xs)
    x         -> (Left "no 0", x)
```

Dies sind dann intrinsics oder interne Funktionen, die ohne einen Zugriff auf „Parser“ nicht möglich sind.

Man versteckt die Funktion `Parser`, damit niemand anderes solche „intrinsics“ definieren kann. Damit wird eine Manipulation des Inputs bzw. ein fehlerhaftes Anwenden unmöglich.

Binärzahlen können wir dann wie folgt parsen:

```
parseBin :: Parser Int  
parseBin = parse0 <|> parse1
```

Binärzahlen können wir dann wie folgt parsen:

```
parseBin :: Parser Int  
parseBin = parse0 <|> parse1
```

```
testParser :: Parser [Int]  
testParser = many parseBin
```

Binärzahlen können wir dann wie folgt parsen:

```
parseBin :: Parser Int
parseBin = parse0 <|> parse1

testParser :: Parser [Int]
testParser = many parseBin

main :: IO ()
main = print $ runParser testParser "101001"
-- (Right [1,0,1,0,0,1], "")
```


Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

- Zeichen zu parsen

Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

- Zeichen zu parsen
- solange Zeichen zu parsen, bis ein Stoppzeichen kommt

Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

- Zeichen zu parsen
- solange Zeichen zu parsen, bis ein Stoppzeichen kommt
- Ziffern zu parsen

Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

- Zeichen zu parsen
- solange Zeichen zu parsen, bis ein Stoppzeichen kommt
- Ziffern zu parsen
- Ziffern zu parsen, bis non-Ziffern kommen und die Zahl zurückgeben

Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

- Zeichen zu parsen
- solange Zeichen zu parsen, bis ein Stoppzeichen kommt
- Ziffern zu parsen
- Ziffern zu parsen, bis non-Ziffern kommen und die Zahl zurückgeben
- etc.

Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

- Zeichen zu parsen
- solange Zeichen zu parsen, bis ein Stoppzeichen kommt
- Ziffern zu parsen
- Ziffern zu parsen, bis non-Ziffern kommen und die Zahl zurückgeben
- etc.

Aus diesen Kleinstbausteinen konstruiert man dann Parser für größere Datenstrukturen.

Alle gängigen Parsing-Bibliotheken benutzt man ähnlich. Die Bibliotheken stellen minimalste Intrinsics zur Verfügung um z.B.

- Zeichen zu parsen
- solange Zeichen zu parsen, bis ein Stoppzeichen kommt
- Ziffern zu parsen
- Ziffern zu parsen, bis non-Ziffern kommen und die Zahl zurückgeben
- etc.

Aus diesen Kleinstbausteinen konstruiert man dann Parser für größere Datenstrukturen.

So haben wir in unserem Log-Beispiel zunächst nur Teil-Parser definiert (Datum, IP, Gerät) und diese dann einfach kombiniert.