

Funktionale Programmierung - fortgeschrittene Konzepte und Anwendungen

Vorlesung 12 vom 14.01.2020: Parsing

Till Mossakowski

Otto-von-Guericke Universität Magdeburg

Wintersemester 2019/20

(Diese und folgende Folien basieren auf

Folien © by J. Betzendahl, S. Dresselhaus CC-BY-NC-SA)

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**
 - ▶ Aktionen und Zustände
 - ▶ Monaden als Berechnungsmuster
 - ▶ **Parser**
 - ▶ Webentwicklung
 - ▶ Linsen

Functor

Inzwischen kennen wir die Typklasse `Functor` ziemlich gut. Ein Funktortyp erlaubt uns, einen anderen Typen in einen **Kontext** (Container) zu heben:

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

Functor

Inzwischen kennen wir die Typklasse `Functor` ziemlich gut. Ein Funktortyp erlaubt uns, einen anderen Typen in einen **Kontext** (Container) zu heben:

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

Wir wissen außerdem, dass die Gesetze für `Functor` garantieren, dass nicht die Struktur des Kontexts, sondern lediglich der Wert im Kontext verändert werden kann:

$$\begin{aligned} \text{fmap id} &= \text{id} \\ \text{fmap (f . g)} &= (\text{fmap f}) . (\text{fmap g}) \end{aligned}$$

Applicative Functors

Ein Applicative ist ein Functor ausgestattet mit zwei zusätzlichen Funktionen, (`<*>`) (`apply`) und `pure`.

`pure` hebt einen Wert in einen (effektlosen) **Standardkontext**. (`<*>`) erlaubt es, auch Funktionen anzuwenden, die selbst im Kontext liegen.

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

Implementierung in einer Monade:

```
pure :: Monad m => a -> m a
pure = return
(<*>) :: Monad m => m (a -> b) -> m a -> m b
mf <*> ma = do f <- mf
               a <- ma
               return $ f a
```

Monads

Eine Monade ist ein `Applicative`, ausgestattet mit entweder

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

oder, äquivalenterweise,

```
join :: m (m a) -> m a.
```

```
class Applicative m => Monad m where
```

```
  return :: a -> m a
```

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

Außerdem hat jede Monade eine `return`-Funktion. Diese ist aber (etwa) gleichbedeutend mit `pure` aus `Applicative` und hat **nichts** mit Rückgabewerten zu tun, wie in anderen Sprachen.

im Vergleich

Nebeneinandergelegt wird die Ähnlichkeit der charakteristischen Funktionen der drei Typklassen klarer:

```
(<$>) :: (a -> b) -> f a -> f b -- Anwendung von fmap
(<*>) :: f (a -> b) -> f a -> f b
(=<<) :: (a -> m b) -> m a -> m b -- == flip (>>=)
```

Instanzen für Maybe

Der option type Maybe ist sowohl ein Funktor, als auch ein Applicative als auch eine Monade:

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


Instanzen für Maybe

Der option type Maybe ist sowohl ein Funktor, als auch ein Applicative als auch eine Monade:

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

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

Instanzen für Maybe

Der option type Maybe ist sowohl ein Funktor, als auch ein Applicative als auch eine Monade:

```
instance Functor Maybe where
```

```
  fmap _ Nothing = Nothing
```

```
  fmap f (Just a) = Just (f a)
```

```
instance Applicative Maybe where
```

```
  pure          = Just
```

```
  Nothing <*> _ = Nothing
```

```
  (Just f) <*> x = f <$> x
```

```
instance Monad Maybe where
```

```
  return          = Just
```

```
  Nothing >>= _ = Nothing
```

```
  (Just x) >>= g = g x
```

Parsing

Parser

Definition: Ein **Parser** ist ein Programm, das Eingabedaten (oft Strings) analysiert und in ein Format umwandelt, das besser zur Weiterverarbeitung geeignet ist.

Parser

Definition: Ein **Parser** ist ein Programm, das Eingabedaten (oft Strings) analysiert und in ein Format umwandelt, das besser zur Weiterverarbeitung geeignet ist.

Die Aufgabe ist also, **unstrukturierte** Daten in **strukturierte** umzuwandeln.

Parser

Definition: Ein **Parser** ist ein Programm, das Eingabedaten (oft Strings) analysiert und in ein Format umwandelt, das besser zur Weiterverarbeitung geeignet ist.

Die Aufgabe ist also, **unstrukturierte** Daten in **strukturierte** umzuwandeln.

Häufige Einsatzgebiete für Parser:

- ▶ Text (z.B. config-files, logs, Sensordaten)
- ▶ Programme (in einer Programmiersprache)
- ▶ JSON (z.B. für Webentwicklung)
- ▶ XML (z.B. (X)HTML)
- ▶ Binärcode (z.B. 3D-Modelle, Netzwerkcode)

Der naive Ansatz

Intuitiv wäre es denkbar, das Problem durch die Verwendung von `if-then-else`-Bäumen, regulären Ausdrücken oder `pattern matching` anzugehen.

Der naive Ansatz

Intuitiv wäre es denkbar, das Problem durch die Verwendung von `if-then-else`-Bäumen, regulären Ausdrücken oder `pattern matching` anzugehen.

Diese sind aber oft in der Praxis schnell zu komplex oder nicht effizient oder mächtig genug.

Der naive Ansatz

Intuitiv wäre es denkbar, das Problem durch die Verwendung von `if-then-else`-Bäumen, regulären Ausdrücken oder `pattern matching` anzugehen.

Diese sind aber oft in der Praxis schnell zu komplex oder nicht effizient oder mächtig genug.

⇒ **This is not the Haskell way to do that!**

In Haskell wollen wir lieber kleinere Teilprobleme offensichtlich korrekt Lösen und diese dann miteinander kombinieren können (**divide and conquer** / **puzzle programming**).

Parsing in Haskell

In Haskell werden zum Parsen oft `Applicatives` und `Monaden` benutzt. Wir werden einem simplen Beispiel davon folgen und uns danach eine moderne Bibliothek (`attoparsec`) in Aktion ansehen.

¹<http://www.willamette.edu/~fruehr/haskell/seuss.html>

Parsing in Haskell

In Haskell werden zum Parsen oft `Applicatives` und `Monaden` benutzt. Wir werden einem simplen Beispiel davon folgen und uns danach eine moderne Bibliothek (`attoparsec`) in Aktion ansehen.

Der zentrale Datentyp eines Parsers ist wie folgt definiert:

```
newtype Parser a = Parser (String -> [(a,String)])
```

¹<http://www.willamette.edu/~fruehr/haskell/seuss.html>

Parsing in Haskell

In Haskell werden zum Parsen oft Applicatives und Monaden benutzt. Wir werden einem simplen Beispiel davon folgen und uns danach eine moderne Bibliothek (`attoparsec`) in Aktion ansehen.

Der zentrale Datentyp eines Parsers ist wie folgt definiert:

```
newtype Parser a = Parser (String -> [(a,String)])
```

Merkreim (Fritz Ruehr)¹:

**„A parser of things is a function from strings
to a list of pairs of things and strings!“**

¹<http://www.willamette.edu/~fruehr/haskell/seuss.html>

Herleitung des Parsertypen

Wie kommt es zu diesem spezifischen Typen? Der erste Gedanke wäre vielleicht viel eher in der Nähe von Folgendem:

```
-- newtype Parser a = Parser (String -> a)?
```

Herleitung des Parsertypen

Wie kommt es zu diesem spezifischen Typen? Der erste Gedanke wäre vielleicht viel eher in der Nähe von Folgendem:

```
-- newtype Parser a = Parser (String -> a)?
```

Hier ist aber das Problem, dass es eventuell **mehrere** Ergebnisse geben könnte, die geparkt werden könnten (dazu auch später mehr). Also vielleicht eine Liste von Ergebnissen?

```
-- newtype Parser a = Parser (String -> [a])?
```

Herleitung des Parsertypen

Wie kommt es zu diesem spezifischen Typen? Der erste Gedanke wäre vielleicht viel eher in der Nähe von Folgendem:

```
-- newtype Parser a = Parser (String -> a)?
```

Hier ist aber das Problem, dass es eventuell **mehrere** Ergebnisse geben könnte, die geparkt werden könnten (dazu auch später mehr). Also vielleicht eine Liste von Ergebnissen?

```
-- newtype Parser a = Parser (String -> [a])?
```

Das ist besser, allerdings bleibt die Möglichkeit, dass Teile des Eingabestrings „übrig bleiben“, die wir gerne weiter verwenden oder zumindest analysieren wollen. Also geben wir die auch mit zurück.

```
newtype Parser a = Parser (String -> [(a,String)])
```

Verstehen des Parsertypen

```
newtype Parser a = Parser (String -> [(a,String)])
```

Parser sind also Funktionen, die einen `String` einlesen und eine Liste von Paaren von was auch immer geparkt werden soll und `Strings` zurück geben.

Verstehen des Parsertypen

```
newtype Parser a = Parser (String -> [(a,String)])
```

Parser sind also Funktionen, die einen `String` einlesen und eine Liste von Paaren von was auch immer geparkt werden soll und `Strings` zurück geben.

Die erste Konvention ist, dass eine leere Liste als Rückgabewert bedeutet, dass kein Wert geparkt werden konnte. Alle Elemente einer nichtleeren Liste sind also Erfolge (**list-of-successes**).

Verstehen des Parsertypen

```
newtype Parser a = Parser (String -> [(a,String)])
```

Parser sind also Funktionen, die einen `String` einlesen und eine Liste von Paaren von was auch immer geparkt werden soll und `Strings` zurück geben.

Die erste Konvention ist, dass eine leere Liste als Rückgabewert bedeutet, dass kein Wert geparkt werden konnte. Alle Elemente einer nichtleeren Liste sind also Erfolge (**list-of-successes**).

Die zweite Konvention ist, dass der `String`, der im Paar mit zurück gegeben wird, die nicht verbrauchte Suffix des Eingabestrings ist.

Ein sehr simpler Parser

Hier ist ein trivialer Parser, der den ersten Char aus einem String einliest und bei leerer Eingabe fehlschlägt:

```
-- newtype Parser a = Parser (String -> [(a,String)])
item :: Parser Char
item = Parser (\xs -> case xs of
                    ""      -> []
                    (c:cs) -> [(c,cs)])
```

Ein sehr simpler Parser

Hier ist ein trivialer Parser, der den ersten Char aus einem String einliest und bei leerer Eingabe fehlschlägt:

```
-- newtype Parser a = Parser (String -> [(a,String)])
item :: Parser Char
item = Parser (\xs -> case xs of
                    ""      -> []
                    (c:cs) -> [(c,cs)])
```

Wir sehen, dass die λ -Notation in Fällen wie diesem auch mit case-of-Syntax benutzt werden kann, damit wir die Funktion nicht extra benannt auslagern müssen.

Noch ein sehr simpler Parser

Hier haben wir einen anderen Parser. Dieser erwartet an der aktuellen Stelle einen **bestimmten** Char (welcher als Parameter übergeben wird) und schlägt fehl, wenn er ihn nicht vorfindet.

```
-- different notation is possible, too
char :: Char -> Parser Char
char c = Parser foo
  where
    foo :: String -> [(a,String)]
    foo "" = []
    foo (x:xs) | (x /= c) = []
                | otherwise = [(c,xs)]
```

Parser sind Monaden (I)

Parser nach dieser Definition (und auch nach moderneren) sind Monaden. Hier ist die Instanz (sie erfüllt auch die monad laws):

```
instance Monad Parser where
  return a = Parser (\cs -> [(a,cs)])
  p >>= f  = Parser (\cs -> concat [parse (f a) cs'
                                     | (a,cs') <- parse p cs])
```

Wie wir es von einer Monade erwarten würden, können wir auch auf vorherige Ergebnisse im aktuellen Kontext zugreifen.

Parser sind Monaden (II)

Werfen wir zuerst einen Blick auf die Bedeutung von `pure` bzw. `return` im Parserkontext:

```
return a = Parser (\cs -> [(a,cs)])
```

Hier wird ein konstanter Wert zurück gegeben und es wird nichts vom Inputstring verbraucht. Das wird insbesondere häufig in Verbindung mit der Verwendung von vorhergehenden Ergebnissen benötigt. Wir werden dazu auch bald ein Beispiel haben.

Parser sind Monaden (III)

Die Definition für ($\gg=$) ist etwas komplizierter, aber immer noch überschaubar. Die Funktion `parse` ist hier ein „Dekonstruktor“.

```
parse :: Parser a -> String -> [(a, String)]
```

```
parse (Parser p) = p
```

```
p >>= f = Parser (\cs -> concat [parse (f a) cs'
                                | (a,cs') <- parse p cs])
```

`p >>= f` wendet zuerst den Parser `p` auf den Eingabestring `cs` an. Auf die Strings der Ergebnisliste wird dann der zweite Parser angewendet. Das Ergebnis ist eine Liste von Listen, die mit `concat` geplättet wird.

do-Notation

Wir wissen jetzt, dass Parser nach unserer Definition Monaden sind. Das bedeutet, dass Haskeells do-Notation auch für Parser zu verwenden.

Zum Beispiel so:

```
-- only succeeds when input starts with "foo"  
fooParser :: Parser ()  
fooParser = do char 'f'  
               char 'o'  
               char 'o'  
               return ()
```

Dieser Stil, um Parser zu schreiben, wird uns im Verlauf der Vorlesung auch nochmal begegnen.

Alternative

Eine weitere Typklasse, die für Parser oftmals wichtig wird ist `Alternative`. Hier wird die Idee festgehalten, dass wie viele verschiedene Parser hintereinander hängen können, von denen wir aber nur ein Resultat haben wollen. Hier ist der Code:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Alternative

Eine weitere Typklasse, die für Parser oftmals wichtig wird ist `Alternative`. Hier wird die Idee festgehalten, dass wie viele verschiedene Parser hintereinander hängen können, von denen wir aber nur ein Resultat haben wollen. Hier ist der Code:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Die Instanz ist nicht kompliziert:

```
instance Alternative Parser where
  empty = Parser (\cs -> [])
  p <|> q = Parser (\cs -> parse p cs ++ parse q cs)
```

Alternativen

Mit (<|>) (sprich: **pick** oder **alternative**) können wir Alternativen ohne Monadeninstanz (!) ausdrücken:

```
parseBool :: Parser Bool
parseBool = (char 'T' >> return True)
           <|> (char 'F' >> return False)
```

Alternative vs. Monoid

Einigen dürfte die Definition von `Alternative` bekannt vorkommen. Sie erinnert an die Definition eines `Monoids` bzw. die der Typklasse `Monoid`. Das ist nicht nur Zufall.

Alternative vs. Monoid

Einigen dürfte die Definition von `Alternative` bekannt vorkommen. Sie erinnert an die Definition eines `Monoids` bzw. die der Typklasse `Monoid`. Das ist nicht nur Zufall.

Beide Klassen stellen eine `Monoid`struktur dar und für bestimmte Typen (wie z.B. `[]`) fallen `mempty` und `empty` bzw. `mappend` und `(<|>)` sogar zusammen.

Der Unterschied (hier ohne genaue Begründung), dass `Alternative` nicht die Möglichkeit bietet, interne Daten zu kombinieren, `Monoid` allerdings schon.

Alternative vs. Monoid

Einigen dürfte die Definition von `Alternative` bekannt vorkommen. Sie erinnert an die Definition eines `Monoids` bzw. die der Typklasse `Monoid`. Das ist nicht nur Zufall.

Beide Klassen stellen eine `Monoid`struktur dar und für bestimmte Typen (wie z.B. `[]`) fallen `mempty` und `empty` bzw. `mappend` und `(<|>)` sogar zusammen.

Der Unterschied (hier ohne genaue Begründung), dass `Alternative` nicht die Möglichkeit bietet, interne Daten zu kombinieren, `Monoid` allerdings schon.

Für Details siehe einen Post auf StackOverflow mit sehr guten Antworten: <http://stackoverflow.com/questions/13080606/>

Deterministische Auswahl

Die Instanz für `Alternative` mit `<|>` gibt uns einen nichtdeterministischen choice-Operator. Wenn ein String auf mehrere Arten und Weisen geparkt werden kann, kriegen wir hier **alle** Ergebnisse.

Deterministische Auswahl

Die Instanz für `Alternative` mit `<|>` gibt uns einen nichtdeterministischen `choice`-Operator. Wenn ein String auf mehrere Arten und Weisen geparkt werden kann, kriegen wir hier **alle** Ergebnisse.

In der Praxis ist es oft so, dass wir nur am ersten Resultat interessiert sind. Auch dafür können wir jetzt eine Funktion schreiben, um so einen **deterministischen** `choice`-Operator zu haben (und Laziness für uns Arbeit ersparen lassen).

```
(+++)  
p +++ q = Parser (\cs -> case parse (p <|> q) cs of  
    []      -> []  
    (x:xs) -> [x])
```

more parser combinators

Dank dieser Instanzen können wir jetzt auch sehr simple und elegante „higher-order“-Parser schreiben. Präzision im Detail erlaubt Eleganz auf höherer Abstraktionsebene:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do {c <- item; if p c then return c else empty}
```

more parser combinators

Dank dieser Instanzen können wir jetzt auch sehr simple und elegante „higher-order“-Parser schreiben. Präzision im Detail erlaubt Eleganz auf höherer Abstraktionsebene:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do {c <- item; if p c then return c else empty}
```

Und diese können wir natürlich auch wieder weiter verwenden. Der Parser `char`, der oben noch fünf Zeilen gebraucht hat, wird z.B. zum eleganten Einzeiler:

```
char :: Char -> Parser Char
char c = sat (c ==)
```

even more parser combinators

Wir können jetzt noch viele andere Parserkombinatoren definieren, die uns nützlich sein werden:

even more parser combinators

Wir können jetzt noch viele andere Parserkombinatoren definieren, die uns nützlich sein werden:

- ▶ `string` parst einen spezifischen String:

```
string :: String -> Parser String
string ""      = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}
```

even more parser combinators

Wir können jetzt noch viele andere Parserkombinatoren definieren, die uns nützlich sein werden:

- ▶ `string` parst einen spezifischen String:

```
string :: String -> Parser String
string ""      = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}
```

- ▶ `many` und `many1` parsen wiederholte Anwendungen eines Parsers (Null oder mehr bzw. eine oder mehr):

```
many :: Alternative f => f a -> f [a]
many v = some v <|> pure [] where
  some v = (:) <$> v <*> many v
liftA2 :: Applicative f => (a->b->c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
many1 :: Alternative f => f a -> f [a]
many1 p = liftA2 (:) p (many p)
```

even more parser combinators

Wir können jetzt noch viele andere Parserkombinatoren definieren, die uns nützlich sein werden:

- ▶ `string` parst einen spezifischen String:

```
string :: String -> Parser String
string ""      = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}
```

- ▶ `many` und `many1` parsen wiederholte Anwendungen eines Parsers (Null oder mehr bzw. eine oder mehr):

```
many :: Alternative f => f a -> f [a]
many v = some v <|> pure [] where
  some v = (:) <$> v <*> many v
liftA2 :: Applicative f => (a->b->c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
many1 :: Alternative f => f a -> f [a]
many1 p = liftA2 (:) p (many p)
```

- ▶ ...

Kombinatoren aus Bibliotheken

Moderne Parser-Bibliotheken wie `attoparsec` bieten noch einige andere nützliche Funktionen dieser Art an:

²<http://www.serpentine.com/blog/2014/05/31/attoparsec/>

Kombinatoren aus Bibliotheken

Moderne Parser-Bibliotheken wie `attoparsec` bieten noch einige andere nützliche Funktionen dieser Art an:

- ▶ `digit`, `decimal` usw. können verwendet werden, um einfach und simpel Zahlen zu parsen.

²<http://www.serpentine.com/blog/2014/05/31/attoparsec/>

Kombinatoren aus Bibliotheken

Moderne Parser-Bibliotheken wie `attoparsec` bieten noch einige andere nützliche Funktionen dieser Art an:

- ▶ `digit`, `decimal` usw. können verwendet werden, um einfach und simpel Zahlen zu parsen.
- ▶ `count` kann die genaue Anzahl kontrollieren („Ich erwarte hier genau 15 Zahlen...“).

²<http://www.serpentine.com/blog/2014/05/31/attoparsec/>

Kombinatoren aus Bibliotheken

Moderne Parser-Bibliotheken wie `attoparsec` bieten noch einige andere nützliche Funktionen dieser Art an:

- ▶ `digit`, `decimal` usw. können verwendet werden, um einfach und simpel Zahlen zu parsen.
- ▶ `count` kann die genaue Anzahl kontrollieren („Ich erwarte hier genau 15 Zahlen...“).
- ▶ `endOfLine` und `isHorizontalSpace` können für `Whitespace` etc. eingesetzt werden

²<http://www.serpentine.com/blog/2014/05/31/attoparsec/>

Kombinatoren aus Bibliotheken

Moderne Parser-Bibliotheken wie `attoparsec` bieten noch einige andere nützliche Funktionen dieser Art an:

- ▶ `digit`, `decimal` usw. können verwendet werden, um einfach und simpel Zahlen zu parsen.
- ▶ `count` kann die genaue Anzahl kontrollieren („Ich erwarte hier genau 15 Zahlen...“).
- ▶ `endOfLine` und `isHorizontalSpace` können für `Whitespace` etc. eingesetzt werden
- ▶ ...

²<http://www.serpentine.com/blog/2014/05/31/attoparsec/>

Kombinatoren aus Bibliotheken

Moderne Parser-Bibliotheken wie `attoparsec` bieten noch einige andere nützliche Funktionen dieser Art an:

- ▶ `digit`, `decimal` usw. können verwendet werden, um einfach und simpel Zahlen zu parsen.
- ▶ `count` kann die genaue Anzahl kontrollieren („Ich erwarte hier genau 15 Zahlen...“).
- ▶ `endOfLine` und `isHorizontalSpace` können für Whitespace etc. eingesetzt werden
- ▶ ...

Durch diese vielen Kombinatoren ist es in der Regel recht simpel (und effizient²), sich selbst komplexe Parser selbst zusammen zu setzen.

²<http://www.serpentine.com/blog/2014/05/31/attoparsec/>

Further Reading

Große Teile der vorangegangenen Sektion basieren auf einem Paper, das online verfügbar ist und auch Codebeispiele verlinkt.

Wenn es in der Vorlesung zu schnell oder zu unübersichtlich war, dann sei dieses Paper wärmstens empfohlen:

FUNCTIONAL PEARLS: Monadic Parsing in Haskell
Graham Hutton, Erik Meijer
<http://www.cs.nott.ac.uk/~pszgmh/pearl.pdf>

Beispiel: Log-Datei

Gegeben ist die Datei `log.txt` (siehe unten), die wir nun in Haskell-Datentypen überführen möchten:

```
2016-02-29T11:16:23Z 124.67.34.60 keyboard
2016-02-29T11:32:12Z 212.141.23.67 mouse
2016-02-29T11:33:08Z 212.141.23.67 monitor
2016-02-29T12:12:34Z 125.80.32.31 speakers
2016-02-29T12:51:50Z 101.40.50.62 keyboard
2016-02-29T13:10:45Z 103.29.60.13 mouse
```

Beispiel: Log-Datei

Gegeben ist die Datei `log.txt` (siehe unten), die wir nun in Haskell-Datentypen überführen möchten:

```
2016-02-29T11:16:23Z 124.67.34.60 keyboard
2016-02-29T11:32:12Z 212.141.23.67 mouse
2016-02-29T11:33:08Z 212.141.23.67 monitor
2016-02-29T12:12:34Z 125.80.32.31 speakers
2016-02-29T12:51:50Z 101.40.50.62 keyboard
2016-02-29T13:10:45Z 103.29.60.13 mouse
```

Wir interessieren uns an dieser Stelle nicht dafür, was diese Einträge bedeuten, sondern nur dafür, dass wir sie in nützlichere Datentypen überführen können.

einzelne Logeinträge

Schauen wir uns also eine Zeile genauer an:

```
2016-02-29T11:16:23Z 124.67.34.60 keyboard
```

Jede Zeile hat also folgende Elemente:

- ▶ ein Datum (Nach ISO 8601:2004³),
- ▶ eine IP-Adresse (0.0.0.0 - 255.255.255.255) und
- ▶ ein Gerät (String als Identifier).

³siehe: https://en.wikipedia.org/wiki/ISO_8601

Zieldatentypen (I)

Wir benötigen außerdem Datenstrukturen, in die wir unsere Eingabe überführen möchten.

Eine einzelne Zeile können wir so repräsentieren:

```
data LogZeile = LogZeile Datum IP Device
```

Und für das gesamte Log:

```
type Log = [LogZeile]
```

Zieldatentypen (II)

Mit dem Modul `Data.Time` können wir uns einen Daten-Typen bauen (hier in **record syntax**):

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

Zieldatentypen (II)

Mit dem Modul `Data.Time` können wir uns einen Daten-Typen bauen (hier in **record syntax**):

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

Auch wenn die Notation leicht anders ist, können wir wie gewohnt Elemente unseres Typs über den Konstruktor erstellen:

```
ghci> Datum (fromGregorian 2016 05 06)
        (TimeOfDay 13 37 0)
Datum {tag = 2016-05-06, zeit = 13:37:00}
```

Zieldatentypen (III)

Eine IP hat in unserem Fall (IPv4) eine Range von 0.0.0.0 bis zu 255.255.255.255. Das können wir gut mit vier Werten vom Typ Word8 (8 bit unsigned integer) aus Data.Word abbilden:

```
import Data.Word
```

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

Zieldatentypen (III)

Eine IP hat in unserem Fall (IPv4) eine Range von 0.0.0.0 bis zu 255.255.255.255. Das können wir gut mit vier Werten vom Typ `Word8` (8 bit unsigned integer) aus `Data.Word` abbilden:

```
import Data.Word

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

Weil wir `Word8` statt `Int` verwenden, liegen keine ungültigen Werte im Wertebereich (**correct-by-construction**).

```
ghci> IP 123 231 13 37
IP 123 231 13 37
```

Zieldatentypen (IV)

Für den Typen der Geräte (welche nur einen String als Identifier hatten) legen wir einen einfachen Summentypen an:

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

Zieldatentypen (IV)

Für den Typen der Geräte (welche nur einen String als Identifier hatten) legen wir einen einfachen Summentypen an:

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

Wir können dieses Typen bei Bedarf beliebig erweitern (zum Beispiel um den Konstruktor Joystick).

Der Compiler kann uns dann über alle Stellen warnen, wo wir den neuen Bewohner noch nicht abfangen.

IP-Parser

Ein Parser für den IP-Typen könnte so aussehen:

```
-- imports and pragmas here  
ipParser :: Parser IP  
ipParser = do  
  d1 <- decimal  
  char '.'  
  d2 <- decimal  
  char '.'  
  d3 <- decimal  
  char '.'  
  d4 <- decimal  
  return $ IP d1 d2 d3 d4
```

Zeitparser

Ein Parser für unseren Typen Datum sähe also ungefähr so aus:

```
-- imports and pragmas here
zeitParser :: Parser Datum
zeitParser = do
  y  <- count 4 digit; char '-'
  mm <- count 2 digit; char '-'
  d  <- count 2 digit; char 'T'
  h  <- count 2 digit; char ':'
  m  <- count 2 digit; char ':'
  s  <- count 2 digit; char 'Z'
  return $
    Datum (fromGregorian (read y) (read mm) (read d))
          (TimeOfDay (read h) (read m) (read s))
```

Geräteparser

Der Parser für Geräte (genau eines aus n verschiedenen Möglichkeiten) ist mit Hilfe von (`<|>`) schnell geschrieben:

```
deviceParser :: Parser Device
deviceParser =
    (string "mouse"    >> return Mouse)
<|> (string "keyboard" >> return Keyboard)
<|> (string "monitor"  >> return Monitor)
<|> (string "speakers" >> return Speakers)
```

Geräteparser

Der Parser für Geräte (genau eines aus n verschiedenen Möglichkeiten) ist mit Hilfe von (`<|>`) schnell geschrieben:

```
deviceParser :: Parser Device
deviceParser =
    (string "mouse"    >> return Mouse)
<|> (string "keyboard" >> return Keyboard)
<|> (string "monitor" >> return Monitor)
<|> (string "speakers" >> return Speakers)
```

Wir matchen jeweils auf einen `String` und falls wir erfolgreich sind, schmeißen wir ihn weg und heben den entsprechenden Wert mit `return` in die `Monade`. Bei einem Fehlschlag versuchen wir es mit dem nächsten Gerät.

Logzeilenparser

Für eine Zeile unserer Logdatei ist ein Parser dank **puzzle programming** ebenfalls kein Kunststück mehr.

```
zeilenParser :: Parser LogZeile
zeilenParser = do
  datum  <- zeitParser    ; space
  ip     <- ipParser      ; space
  device <- deviceParser
  return $ LogZeile datum ip device
```

Logzeilenparser

Für eine Zeile unserer Logdatei ist ein Parser dank **puzzle programming** ebenfalls kein Kunststück mehr.

```
zeilenParser :: Parser LogZeile
zeilenParser = do
  datum  <- zeitParser    ; space
  ip     <- ipParser      ; space
  device <- deviceParser
  return $ LogZeile datum ip device
```

Ein komplettes Log zu parsen geht dann wie folgt:

```
logParser :: Parser Log
logParser = many (do l <- zeilenParser
                    endOfLine
                    return l)
```

Testlauf

Was bleibt, ist ein kleines Testprogramm zu schreiben, und es auszuführen:

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

Testlauf

Was bleibt, ist ein kleines Testprogramm zu schreiben, und es auszuführen:

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

Das Ergebnis:

```
Right [LogZeile (Datum {tag = 2016-02-29, zeit = 11:16:23}) (IP 124 67 34 60) Keyboard,
       LogZeile (Datum {tag = 2016-02-29, zeit = 11:32:12}) (IP 212 141 23 67) Mouse,
       LogZeile (Datum {tag = 2016-02-29, zeit = 11:33:08}) (IP 212 141 23 67) Monitor,
       LogZeile (Datum {tag = 2016-02-29, zeit = 12:12:34}) (IP 125 80 32 31) Speakers,
       LogZeile (Datum {tag = 2016-02-29, zeit = 12:51:50}) (IP 101 40 50 62) Keyboard,
       LogZeile (Datum {tag = 2016-02-29, zeit = 13:10:45}) (IP 103 29 60 13) Mouse]
```


Zusammenfassung

- ▶ Parser transformieren **Text** in **strukturierte Datentypen**
- ▶ Gängige Parser sind **monadisch**
- ▶ Gängige Parser verwenden **Kombinatoren** wie `many`, `+++`, `<|>`
- ▶ Zwei typische monadische Kombinatorparser:

`attoparsec` schnell; geeignet für sehr großen Input: Logfiles, Paketstrukturen etc.

`parsec` mehr Inputtypen (kann mit Scanner interagieren);
Monadentransformer (Zustand möglich); geeignet für
Terme, Programmiersprachen; bessere Fehlermeldungen



xkcd by Randall Munroe, CC-BY-NC
<https://xkcd.com/1270/>