

Intermediate Functional Programming in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

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
Applicative-Typklasse mit ihrer monoidal-Erweiterung „Alternative“.

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

Was heisst das genau?

Kernstück für das stückweise Parsen bildet die Applicative-Typklasse mit ihrer monoidal-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 falsche 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)
```

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 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 Alternativen:

```
{-# 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 Rechte ausführt, wenn das Linke einen Fehler wirft.

Für das Gerät brauchen wir die Mächtigkeit von Alternativen:

```
{-# 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 Rechte ausführt, wenn das Linke einen Fehler wirft.

Wir matchen hier jeweils auf den String, schmeissen das gematchte Ergebnis weg (den String selbst) 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
```

<* (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]
```