

# Monadische Parser

8 out of 10 cats do ... *parsing*

Christian Höner zu Siederdisen

`christian.hoener.zu.siederdisen@uni-jena.de`

Theoretische Bioinformatik, Bioinformatik Uni Jena

9. Nov 2023

## Parsing mit coolen Typen<sup>1</sup>

---

Parser bisher:

```
1 pSumPNP :: [Token] -> Maybe (Expr,[Token])
```

generalisieren:

```
1 type Parser = [Token] -> Maybe (Expr,[Token])
```

0...*n* Parses:

```
1 type Parser = [Token] -> [(Expr,[Token])]
```

Expr generalisieren:

```
1 type Parser a = [Token] -> [(a,[Token])]
```

Token generalisieren:

```
1 type Parser t a = [t] -> [(a,[t])]
```

---

<sup>1</sup>und schlechten Wortwitzen

## Parsing mit coolen Typen<sup>2</sup>

---

Namen erfinden:

```
1 newtype Parser t a = Parser {parse :: [t] -> [(a,[t])]}
```

und vergleichen:

```
1 type Parser t a = [t] -> [(a,[t])]
2
3 pSumPNP :: Parser Token Expr
4 pSumPNP :: Parser Char Expr
5     == [Char] -> [(Expr,[Char])]
6     == String -> [(Expr,String)]
```

Ein Parser ist eine Funktion die eine Eingabeliste  $[t]$  von Token nimmt und eine Liste  $[(a, [t])]$  von Parses  $a$  zusammen mit der restlichen Eingabe  $[t]$  liefert.

---

<sup>2</sup>und schlechten Wortwitzen

## Parsen eines Tokens

---

```
1 newtype Parser t a = Parser { parse :: [t] -> [(a,[t])] }
2
3 itemP :: Parser t t
4 itemP = Parser go
5   where go [] = []
6         go (x:xs) = [(x,xs)]
7
8
9
10 atomP :: Eq t => t -> Parser t t
11 atomP c = Parser go
12   where go [] = []
13         go (x:xs) | x/=c = []
14         go (x:xs) = [(x,xs)]
```

## Definition (Typklasse)

Mechanismus um Mengen von Operationen über verschiedenen Typen zu implementieren. Dies erlaubt generische Interfaces und Polymorphismus.

- Erlaubt ad-hoc Polymorphismus: Code kann gegen die Typklasse geschrieben werden und funktioniert generisch auf allen Typen die die Typklasse unterstützen
- definiert eine Menge von Funktionen, genannt Methoden
- diese werde dann für verschiedene Typen implementiert
- definiert Verhalten und Fähigkeiten
- `class` Keyword und List von Signaturen
- Modularität, Code wieder nutzen, Abstraktion

Beispiel für Typen, die auf Gleichheit getestet werden können.

```
1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
```

Beispiel für Typen, die auf Gleichheit getestet werden können.

```
1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
4   x /= y = not (x==y)
5   x == y = not (x/=y)
```

Instanzen einer solchen Typklasse erfordern Implementation der Methoden

```
1 data Person = Person String Int
2
3 instance Eq Person where
```

Beispiel für Typen, die auf Gleichheit getestet werden können.

```
1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
4   x /= y = not (x==y)
5   x == y = not (x/=y)
```

Instanzen einer solchen Typklasse erfordern Implementation der Methoden

```
1 data Person = Person String Int
2
3 instance Eq Person where
4   (Person name1 age1) == (Person name2 age2)
5     = (name1 == name2) && (age1 == age2)
```



Achtung!

---

Ab hier kommt eine Lüge!

Ab hier kommt eine Lüge!  
Eine Kleine!

Ab hier kommt eine Lüge!  
Eine Kleine!

Das Haskell-Typklassensystem ist kompliziert und ich vereinfache es hier  
auf das Jahr 2000 (in etwa)

Ab hier kommt eine Lüge!  
Eine Kleine!

Das Haskell-Typklassensystem ist kompliziert und ich vereinfache es hier  
auf das Jahr 2000 (in etwa)

### Definition (Funktork)

Ein Funktor ist eine Typklasse die Typen repräsentiert über die Funktionen gemapped werden können.

Damit kann eine Funktion auf die Werte innerhalb eines *Containers* oder *Kontext* angewandt werden. Die Struktur des Containers bleibt erhalten.

```
1 class Functor f where
```

### Definition (Funktork)

Ein Funktor ist eine Typklasse die Typen repräsentiert über die Funktionen gemapped werden können.

Damit kann eine Funktion auf die Werte innerhalb eines *Containers* oder *Kontext* angewandt werden. Die Struktur des Containers bleibt erhalten.

```
1 class Functor f where
2   fmap :: (a -> b) -> f a -> f b
3
4 data Maybe a = Nothing | Just a
```

### Definition (Funktork)

Ein Funktor ist eine Typklasse die Typen repräsentiert über die Funktionen gemapped werden können.

Damit kann eine Funktion auf die Werte innerhalb eines *Containers* oder *Kontext* angewandt werden. Die Struktur des Containers bleibt erhalten.

```
1 class Functor f where
2   fmap :: (a -> b) -> f a -> f b
3
4 data Maybe a = Nothing | Just a
5
6 instance Functor Maybe where
```

### Definition (Funktork)

Ein Funktor ist eine Typklasse die Typen repräsentiert über die Funktionen gemapped werden können.

Damit kann eine Funktion auf die Werte innerhalb eines *Containers* oder *Kontext* angewandt werden. Die Struktur des Containers bleibt erhalten.

```
1 class Functor f where
2   fmap :: (a -> b) -> f a -> f b
3
4 data Maybe a = Nothing | Just a
5
6 instance Functor Maybe where
7   fmap _ Nothing = Nothing
```



### Definition (Funktork)

Ein Funktor ist eine Typklasse die Typen repräsentiert über die Funktionen gemapped werden können.

Damit kann eine Funktion auf die Werte innerhalb eines *Containers* oder *Kontext* angewandt werden. Die Struktur des Containers bleibt erhalten.

```
1 class Functor f where
2   fmap :: (a -> b) -> f a -> f b
3
4 data Maybe a = Nothing | Just a
5
6 instance Functor Maybe where
7   fmap _ Nothing = Nothing
8   fmap f (Just x) = Just (f x)
```

```
1 newtype Parser t a = Parser { parse :: [t] -> [(a,[t])] }
```

```
1 newtype Parser t a = Parser { parse :: [t] -> [(a,[t])] }  
2  
3 instance Functor (Parser t) where
```

```
1 newtype Parser t a = Parser { parse :: [t] -> [(a,[t])] }
2
3 instance Functor (Parser t) where
4     fmap :: (a -> b) -> Parser t a -> Parser t b
```

```
1 newtype Parser t a = Parser { parse :: [t] -> [(a,[t])] }
2
3 instance Functor (Parser t) where
4     fmap :: (a -> b) -> Parser t a -> Parser t b
5     fmap f (Parser p) =
```

```
1 newtype Parser t a = Parser { parse :: [t] -> [(a,[t])] }
2
3 instance Functor (Parser t) where
4     fmap :: (a -> b) -> Parser t a -> Parser t b
5     fmap f (Parser p) =
6         Parser (\cs -> [(f a,ds) | (a,ds) <- p cs])
```

## Definition (Monade)

Eine Monad is eine Typklasse die eine *Berechnung* mit einem spezifischem *sequentiell* Verhalten definiert.

- Erlaubt Seiteneffekte, zB I/O oder State zu manipulieren
- Zwei Operationen zum Kombinieren und sequentiellm Aufruf
- `return :: a -> m a` nimmt Werte und verpackt sie innerhalb der Monade, also in den monadischen Kontext
- `(>>=) :: m a -> (a -> m b) -> m b` "bind": nimmt einen monadischen Wert und eine Funktion die auf dem unterliegenden Wert operiert, wobei das Resultat innerhalb der Monade bleibt
- "bind" erlaubt das Verketteten von Operationen
- "referential transparency" bleibt stets erhalten

## Maybe Monade

---

```
1 data Maybe a = Nothing | Just a
2
3 instance Monad Maybe where
4     return :: a -> m a
5     return x = Just x
6
7     (>>=) :: m a -> (a -> m b) -> m b
8     Nothing >>= f = Nothing
9     Just x >>= f = f x
```

- `return` nimmt Werte und wrapped die in `Just`
- `>>=` "bind", nimmt `Maybe a`'s und wendet `f` nur auf die `a` in `Just a` an. Wobei das Ergebnis `Just (f a)` ist



```
1 instance Monad (Parser t) where
2   return :: a -> Parser t a
3   return = pure
4   (>>=) :: Parser t a -> (a -> Parser t b) -> Parser t b
5   Parser p >>= pq = Parser (\cs ->
6     [(b,es) | (a,ds) <- p cs
7       , let Parser q = pq a
8         , (b,es) <- q ds])
```

```
1 noP :: Parser t a
2 noP = Parser $ \cs -> []
3
4 satP :: (t -> Bool) -> Parser t t
5
6 satP c = Parser go
7   where go [] = []
8         go (x:xs) | c x = [(x,xs)]
9         go _ = []
10
11 satP c = itemP >>= \x -> if c x then pure x else mzero
12
13 satP c = Parser goL >>= \x ->
14   if c x then Parser (\cs -> [(x,cs)])
15   else Parser (\cs -> [])
16   where goL [] = []
17         goL (x:xs) = [(x,xs)]
```

```
1  satP c = do
2    x <- itemP
3    if c x then pure x else mzero
4
5  testPP =
6    itemP >>= \x1 ->
7    itemP >>= \x2 ->
8    itemP >>
9    itemP >>= \x4 ->
10   return (x1,x2,x4)
11
12 testD0 = do
13   x1 <- itemP
14   x2 <- itemP
15   itemP
16   x4 <- itemP
17   return (x1,x2,x4)
```

```
1 theseP :: Eq t => [t] -> Parser t [t]
2 theseP [] = pure []
3 theseP (t:ts) = satP (t==) >> theseP ts
4
5 manyP p = someP p <|> return []
6
7 someP p = do {x <- p; xs <- manyP p; return (x:xs)}
8
9 -- btw. "many" und "some" gibt es fuer *alle* Alternative
   's!
```

In Haskell liegt die Kunst nicht darin moeglichst viele verschiedene Kombinatoren zu haben, sondern wenige, *generische* Kombinatoren die breite Anwendung finden.

Deshalb machen auch "Monaden" Sinn: sie beschreiben generische strukturelle Features

## Listen, und Klammern

---

```
1 sepBy :: Parser t a -> Parser t b -> Parser t [a]
2 p 'sepBy' s = (p 'sepBy1' s) <|> return []
3
4 -- HEY! Das sind ja programmierbare Semikolons!
5
6 sepBy1 :: Parser t a -> Parser t b -> Parser t [a]
7 p 'sepBy1' s = do {a <- p; as <- many (s >> p)
8                   ;return (a:as)}
9
10 bracketedP :: Parser t l -> Parser t x -> Parser t r
11             -> Parser t x
12 bracketedP lP xP rP = do
13   _l <- lP
14   x   <- xP
15   _r <- rP
16   return x
```

## Operatoren und Operanden

---

```
1 chain1 :: Parser t a -> Parser t (a -> a -> a) -> a
2   -> Parser t a
3 chain1 p op a = (p 'chain1' op) <|> return a
4
5 chain11 :: Parser t a -> Parser t (a -> a -> a)
6   -> Parser t a
7 chain11 p op = p >>= go
8   where go a = do
9         f <- op
10        b <- p
11        go (f a b)
12        <|> return a
```

## Noch schnell ein lexikalischer Parser

---

```
1 spaceP :: Parser Char String
2 spaceP = many (satP isSpace)
3
4 tokenP :: Parser Char a -> Parser Char a
5 tokenP p = p <* spaceP
6
7 stringP :: String -> Parser Char String
8 stringP = tokenP . theseP
```

## Und ein neuer Expr Parser

---

```
1 digitP :: Parser Char Int
2 digitP = satP isDigit >=> \x -> pure (ord x - ord '0')
3
4 numberP :: Parser Char Expr
5 numberP = do
6   ds <- some digitP
7   spaceP
8   return . Num $ foldl (\acc x -> 10*acc + x) 0 ds
9
10 bracketP :: Parser Char Expr
11 bracketP = bracketedP l exprP r
12   where l = tokenP $ atomP '('
13         r = tokenP $ atomP ')'
```

Dieser Parser braucht jetzt auch kein Tokenizing mehr! Und versteht Leerzeichen!



## Das ist ja einfach ...

---

```
1 addopP, mulopP
2   :: Parser Char (Expr -> Expr -> Expr)
3
4 addopP = (stringP "+" >> pure (App Add))
5         <|> (stringP "-" >> pure (App Sub))
6
7 mulopP = (stringP "*" >> pure (App Mul))
8         <|> (stringP "/" >> pure (App Div))
```

## Der komplette Expr Parser

---

```
1  -- Expr's sind Terme mit addop's verbunden
2
3  exprP :: Parser Char Expr
4  exprP = termP 'chainl1' addopP
5
6  -- Terme sind factors mit Multiplikationen verbunden
7
8  termP = factorP 'chainl1' mulopP
9
10 -- factors sind Zahlen oder wohlgeformte Klammern
11
12 factorP = numberP <|> bracketP
```

- Wir haben Functor, Alternative, Applicative, Monad als Abstraktionsmittel kennengelernt
- Jede dieser Abstraktionen erlaubt es eine Zahl vorgefertigter Kombinatoren zu nutzen
- Unser neuer Parser ist ein Beispiel fuer Monaden in Aktion
- Und auch fuer do-Notation, die aber nur syntaktischer Zucker ist
- Unser neuer Parser kann prinzipiell alle legalen Parses, nicht nur einen, erzeugen

Es folgt dann die Frage ob sich der "Monad" Aufwand lohnt? (Ja) Und die Konstruktion eines effizienteren Countdown!

## Applicatives ???!

---

```
1 instance Applicative (Parser t) where
2   pure :: a -> Parser t a
3   pure x = Parser (\cs -> [(x,cs)])
4   (<*>) :: Parser t (a -> b) -> Parser t a -> Parser t b
5   Parser p <*> Parser q = Parser (\cs ->
6     [ (f a,es) | (f,ds) <- p cs
7       , (a,es) <- q ds])
```

## Alternatives ?!

---

```
1 instance Applicative (Parser t) => Alternative (Parser t)
2 where
3     empty = noP
4     Parser p <|> Parser q = Parser $ \cs -> p cs ++ q cs
5
6
7 instance (Monad (Parser t), Alternative (Parser t))
8 => MonadPlus (Parser t) where
9     mzero = empty
10    mplus = (<|>)
```