

# 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

Dec 1<sup>st</sup>, 2022

# 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)]

1  -- | This is Maybe
2  data Option a = Nul | Has a
3      deriving (Show)

```

## Functoren ?!

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

```
1 -- Funktion f auf Elemente in Has anwenden
2 instance Functor Option where
3   fmap :: (a->b) -> Option a -> Option b
4   fmap f Nul      = Nul
5   fmap f (Has a) = Has (f a)
```

# 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])
```

```
1 -- Sowohl Funktion als auch Argument sind "eingepackt"
2 instance Applicative Option where
3   pure :: a -> Option a
4   pure = Has
5   (<*>) :: Option (a->b) -> Option a -> Option b
6   Nul <*> _ = Nul
7   _ <*> Nul = Nul
8   Has f <*> Has a = Has (f a)
```

## 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 = (<|>)

1 -- | Entweder lhs (bevorzugt) oder rhs nutzen
2 instance Applicative Option => Alternative Option where
3     empty :: Option a
4     empty = Nul
5     (<|>) :: Option a -> Option a -> Option a
6     Nul <|> option = option
7     Has a <|> _     = Has a
```

# Monads: Are you joking?

```
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 -- | a innerhalb eines "Option" Kontext bearbeiten
2 instance Monad Option where
3   (>>=) :: Option a -> (a->Option b) -> Option b
4   Nul >>= f = Nul
5   Has a >>= f = f a
```



```

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)]
  
```

## fuz rho doh

```
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)
```

# Combinator-Time

```

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
```

## Zusammenfassung

- 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!

# Memoisation

*Wikipedia:* Memoisation oder Memoisierung ist eine Technik, um Computerprogramme zu beschleunigen, indem Rückgabewerte von Funktionen zwischengespeichert anstatt neu berechnet werden.

*Warum ist das von Interesse? Was macht mkExpr? Wie haeufig sehen wir Teilsequenzen in mkExpr?*

1,2,3,4,5,6

# fix und Open Recursion

Betrachten wir noch einmal *Fibonacci*

```

1 fib :: Int -> Int
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
5         ^^^         ^^^

```

- fib nutzt (offensichtlich Rekursion)
- wir wollen jetzt die Rekursion "heraus ziehen" in eine eigene Funktion

```

1 fob :: (Int -> Int) -> Int -> Int
2 fob f 0 = 1
3 fob f 1 = 1
4 fob f n = f (n-1) + f (n-2)
5           ^^^         ^^^
6
7 runfob :: Int -> Int
8 runfob = fib runfob

```

# fix und Open Recursion

```

1  fob :: (Int -> Int) -> Int -> Int
2  fob f 0 = 1
3  fob f 1 = 1
4  fob f n = f (n-1) + f (n-2)
5             ^^^         ^^^
6
7  runfob :: Int -> Int
8  runfob = fib runfob
9
10 -- fib (fib (fib (fib (...))))

```

## Generalisieren von runfob

```

1  -- 'f' ist der Typ von Funktionen, zB
2  -- fix :: ((Int -> Int) -> (Int -> Int)) -> (Int -> Int)
3  fix :: (f -> f) -> f
4  fix f = let x = f x in x

```

# fix und Open Recursion

```

1 fob :: (Int -> Int) -> Int -> Int
2 fob f 0 = 1
3 fob f 1 = 1
4 fob f n = f (n-1) + f (n-2)
5           ^^^           ^^^
6
7 fix :: (f -> f) -> f
8 fix f = let x = f x in x
9
10 runfob :: Int -> Int
11 runfob = fix fob
  
```

fix erlaubt es uns in jedem Rekursionsschritt interessante Dinge mit fob zu machen, wobei die "interessanten Dinge" auf jeder Ebene passieren!

# fix und Open Recursion

```

1  fob :: (Int -> Int) -> Int -> Int
2  fob f 0 = 1
3  fob f 1 = 1
4  fob f n = f (n-1) + f (n-2)
5             ^^^         ^^^
6
7  fix :: (f -> f) -> f
8  fix f = let x = f x in x
9
10 -- fix fib 10
11 -- fix (\f -> fib ((1+) . f)) 10
  
```

# Memotables

```
1 fob :: (Int -> Int) -> Int -> Int
2 fob f 0 = 1
3 fob f 1 = 1
4 fob f n = f (n-1) + f (n-2)
5
6 fix :: (f -> f) -> f
7 fix f = let x = f x in x
8
9 -- gib eine Liste von Indices fuer die Funktion f
10 -- und ihre Werte gespeichert werden
11 memoList :: [Int] -> (Int -> a) -> (Int -> a)
12 memoList ks f = (map f ks !!)
13
14 memofib :: Int -> Int
15 memofib = fix (memoList [0..1000] . fib)
16 -- === fix (fib . memoList [0..1000])
```



# memo Expr

```

1  fix :: (f -> f) -> f
2  fix f = let x = f x in x
3
4  -- mkExprs, aber die Rekursion ist jetzt via "mk"
5  orExprs :: ([Int] -> [(Expr, Value)]) -> [Int] -> [(Expr, Value)]
6  orExprs mk [x] = [(Num x, Value x)]
7  orExprs mk xs =
8    [ ev | (ys,zs) <- unmerges xs
9      , l <- mk ys, r <- mk zs
10     , ev <- combine l r ]
11
12 -- wie bei fib, wir machen die Rekursion explizit
13 recExprs :: [Int] -> [(Expr, Value)]
14 recExprs = orExprs recExprs
15
16 fixExprs :: [Int] -> [(Expr, Value)]
17 fixExprs = fix orExprs
  
```

# memo Expr

```

1  fix :: (f -> f) -> f
2  fix f = let x = f x in x
3
4  -- mkExprs, aber die Rekursion ist jetzt via "mk"
5  orExprs :: ([Int] -> [(Expr, Value)]) -> [Int] -> [(Expr, Value)]
6  orExprs mk [x] = [(Num x, Value x)]
7  orExprs mk xs =
8    [ ev | (ys,zs) <- unmerges xs
9      , l <- mk ys, r <- mk zs
10     , ev <- combine l r ]
11
12 -- Speichern fuer alle sequences, die man angibt
13 memoSeqs :: forall a . [[Int]] -> ([Int] -> a) -> ([Int] -> a)
14 memoSeqs sqs f = (tbl Map.!)
15   where
16     tbl :: Map.Map [Int] a
17     tbl = Map.fromList [ (s,f s) | s <- sqs ]

```

# Laufzeit?

```

1 length . concatMap mkExprs $ subseqs [1..6]
2 -- 5341067
3 -- (6.55 secs, 6,155,652,240 bytes)
4
5 length . concatMap recExprs $ subseqs [1..6]
6 -- 5341067
7 -- (6.68 secs, 6,182,532,000 bytes)
8
9 length . concatMap (fix orExprs) $ subseqs [1..6]
10 -- 5341067
11 -- (6.60 secs, 6,182,532,112 bytes)
12
13 length . concatMap (fix (orExprs . memoSeqs (subseqs [1..6])))
14 -- 5341067
15 -- (4.45 secs, 4,110,812,488 bytes)
  
```

# Zusammenfassung

- Rekursion kann durch `fix` den "least fixpoint operator" expliziter gemacht werden
- Dadurch koennen wir in jeder Ebene einer Rekursion Funktionalitaet "injizieren"
- Hier "ziehen" wir eine Datenstruktur durch, mit der wir fuer jede Eingabe die Ausgabe speichern (*memoisieren*)
- `fib` wird dadurch polynomiell
- `orExprs` und `memoSeqs` bringen weniger, da die Datenstrukturen viel groesser sind