Monadische Parser

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

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Nov 23^{th}, 2022

Parsing mit coolen Typen[1]

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])]
```

[1] und schlechten Wortwitzen

Parsing mit coolen Typen[2]

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.

[2] 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)]
```

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])
```

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 = (<|>)
```

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])
```

```
noP :: Parser t a
noP = Parser $ \cs -> []

satP :: (t -> Bool) -> Parser t t

satP c = Parser go
  where go [] = []
        go (x:xs) | c x = [(x,xs)]
        go _ = []

satP c = itemP >>= \x -> if c x then pure x else mzero

satP c = Parser goL >>= \x ->
  if c x then Parser (\cs -> [(x,cs)])
         else Parser (\cs -> [])
  where goL [] = []
        goL (x:xs) = [(x,xs)]
```

fuz rho *do*h

```
 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  testDO = 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  chainl :: Parser t a -> Parser t (a -> a -> a) -> a
 2    -> Parser t a
 3  chainl p op a = (p `chainl1` op) <|> return a
 4
 5  chainl1 :: Parser t a -> Parser t (a -> a -> a)
 6    -> Parser t a
 7  chainl1 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 vesteht 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!