Token Parsing

Zu Fuß

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

Theoretische Bioinformatik, Bioinformatik Uni Jena

9. Nov 2023

Unser Ziel

- Transformieren von String-Eingaben in `Expr`-Trees
- String / Liste von Char enthält aufeinanderfolgende gleiche Elemente (Ziffern)
- Identifikation "atomarer" Elemente in der Eingabe
- Umschreiben in Liste die einzelne Arten von Elementen unterscheidbar macht

Tokenizing

Tokens zerlegen eine Eingabe in “atomare” Bausteine auf denen wir leichter arbeiten können. Was müssen wir parsen?

```
1  data Token
```

Tokenizing

Tokens zerlegen eine Eingabe in “atomare” Bausteine auf denen wir leichter arbeiten können. Was müssen wir parsen?

```
1  data Token
2     = TNum Int
3     | TOp Op
4     | TLeft
5     | TRight
6     deriving (Show,Eq)
```

Tokenizer

```
1  tokenize :: String -> [Token]
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
3  tokenize (x:xs)
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
3  tokenize (x:xs)
4  -- eine Zahl
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
3  tokenize (x:xs)
4  -- eine Zahl
5    | isDigit x = let (ls,rs) = span isDigit xs
6                  in  TNum (read (x:ls)) : tokenize rs
7  -- Klammer links
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
3  tokenize (x:xs)
4  -- eine Zahl
5    | isDigit x = let (ls,rs) = span isDigit xs
6                  in  TNum (read (x:ls)) : tokenize rs
7  -- Klammer links
8    | x == '(' = TLeft : tokenize xs
9  -- Klammer rechts
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
3  tokenize (x:xs)
4  -- eine Zahl
5    | isDigit x = let (ls,rs) = span isDigit xs
6                  in  TNum (read (x:ls)) : tokenize rs
7  -- Klammer links
8    | x == '(' = TLeft : tokenize xs
9  -- Klammer rechts
10   | x == ')' = TRight : tokenize xs
11 -- Operator
```

Tokenizer

```
1  tokenize :: String -> [Token]
2  tokenize [] = []
3  tokenize (x:xs)
4  -- eine Zahl
5    | isDigit x = let (ls,rs) = span isDigit xs
6                  in  TNum (read (x:ls)) : tokenize rs
7  -- Klammer links
8    | x == '(' = TLeft : tokenize xs
9  -- Klammer rechts
10   | x == ')' = TRight : tokenize xs
11 -- Operator
12   | x `elem` "+-*/" = TOp (parseOp x) : tokenize xs
13
14 parseOp :: Char -> Op
15 parseOp '+' = Add
16 ...
```

Von Token zu Expr

Parser wie `pSumPNP` erwarten eine Liste von Token und geben eine Expr zurueck, sowie eine Restliste von *nicht bearbeiteten* Token

```
1  token2Expr :: [Token] -> Expr
```

Von Token zu Expr

Parser wie pSumPNP erwarten eine Liste von Token und geben eine Expr zurueck, sowie eine Restliste von *nicht bearbeiteten* Token

```
1  token2Expr :: [Token] -> Expr
2  token2Expr xs = case pSumPNP xs
3    of Just (expr,[]) -> expr
4       Nothing -> error (show xs)
```

Wir bauen den Parser jetzt aber mal bottom-up auf

hoechste “Precedence”: Zahlen und Klammern

Zur Erinnerung: `data Maybe a = Nothing | Just a`

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

hoechste “Precedence”: Zahlen und Klammern

Zur Erinnerung: `data Maybe a = Nothing | Just a`

```
1  pNumParen :: [Token] -> Maybe (Expr,[Token])
2
3  -- eine Zahl zu parsen ist einfach
```

hoechste “Precedence”: Zahlen und Klammern

Zur Erinnerung: `data Maybe a = Nothing | Just a`

```
1  pNumParen :: [Token] -> Maybe (Expr,[Token])
2
3  -- eine Zahl zu parsen ist einfach
4  pNumParen (TNum n:xs) = Just (Zahl n, xs)
5
6  -- bei linker Klammer: auf Rest den kompletten Parser
7  -- rekursiv laufen lassen
```

hoechste “Precedence”: Zahlen und Klammern

Zur Erinnerung: `data Maybe a = Nothing | Just a`

```
1  pNumParen :: [Token] -> Maybe (Expr,[Token])
2
3  -- eine Zahl zu parsen ist einfach
4  pNumParen (TNum n:xs) = Just (Zahl n, xs)
5
6  -- bei linker Klammer: auf Rest den kompletten Parser
7  -- rekursiv laufen lassen
8  pNumParen (TLeft:xs) = case pSumPNP xs of
9  -- alles bis zur schliessenden Klammer + Rest
10   Just (expr, TRight:ys) -> Just (expr, ys)
11 -- misses clothing bracket
12   Just _ -> Nothing
13   Nothing -> Nothing
14 pNumParen _ = Nothing
```

Produkt + Division

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

Produkt + Division

```
1  pProdNP :: [Token] -> Maybe (Expr,[Token])
2
3  -- von links Zahl oder Klammer lesen
```

Produkt + Division

```
1  pProdNP :: [Token] -> Maybe (Expr,[Token])
2
3  -- von links Zahl oder Klammer lesen
4  pProdNP xs = case pNumParen xs of
5
6  -- es geht mit Mul weiter
```

Produkt + Division

```
1  pProdNP :: [Token] -> Maybe (Expr,[Token])
2
3  -- von links Zahl oder Klammer lesen
4  pProdNP xs = case pNumParen xs of
5
6  -- es geht mit Mul weiter
7    Just (el, TOp Mul:ys) -> case pProdNP ys of
8
9  -- zweites Argument ebenso erfolgreich geparsed
```

Produkt + Division

```
1  pProdNP :: [Token] -> Maybe (Expr,[Token])
2
3  -- von links Zahl oder Klammer lesen
4  pProdNP xs = case pNumParen xs of
5
6  -- es geht mit Mul weiter
7    Just (el, TOp Mul:ys) -> case pProdNP ys of
8
9  -- zweites Argument ebenso erfolgreich geparsed
10     Just (er, zs) -> Just (App Mul el er, zs)
11     Nothing -> Nothing
12
13 -- analog fuer Division
```

Produkt + Division

```
1  pProdNP :: [Token] -> Maybe (Expr,[Token])
2
3  -- von links Zahl oder Klammer lesen
4  pProdNP xs = case pNumParen xs of
5
6  -- es geht mit Mul weiter
7    Just (el, TOp Mul:ys) -> case pProdNP ys of
8
9  -- zweites Argument ebenso erfolgreich geparsed
10     Just (er, zs) -> Just (App Mul el er, zs)
11     Nothing -> Nothing
12
13 -- analog fuer Division Just (el, TOp Div:ys) -> case pProdNP ys
       ofJust (er, zs) -> Just (App Div el er, zs)Nothing -> Nothingres
       -> res
```

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

-- erstmal testen ob Mul,(), oder Zahl
pSumPNP xs = case pProdNP xs of

-- Danach folgt Add Token
  Just (el, TOp Add:ys) -> case pSumPNP ys of

-- zweites Argument ebenso erfolgreich geparsed
    Just (er, zs) -> Just (App Add el er, zs)
    Nothing -> Nothing

-- analog fuer Subtraktion
  Just (el, TOp Sub:ys) -> case pSumPNP ys of
    Just (er, zs) -> Just (App Sub el er, zs)
    Nothing -> Nothing
  res -> res
```

Run!

```
1  main :: IO ()
2  main = do
3    print "Give me numbers"
4    ns :: [Int] <- sort . map read . words <$> getLine
5    print "Give me a target"
6    tgt :: Int <- fmap read getLine
7    printf "Given numbers %s get closest to the target
         number %d with an arithmetic expression:\n" (
         unwords $ fmap show ns) tgt
8    suggest :: String <- getLine
9    let (expr,nst) = nearest tgt . concatMap mkExprs $
         subseqs ns
10        myexp = token2Expr $ tokenize suggest
11   print myexp
12   printf "You have: %d\n" $ auswerten myexp
13   print expr
14   printf "Optimal is: %d\n" nst
```

Und nun?

- Verbessern von `mkExprs` (Memoization)
- Bessere Parser (Monaden!)
- Fehlerbehebung (Monaden!)
- Ein- und Ausgabe (Monaden!)
- “gegeneinander spielen” (. . . Monaden!)