

CS314. Функциональное программирование

Лекция 9. Функциональные парсеры

В. Н. Брагилевский

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

13 октября 2017 г.

Задача разбора текста

- Парсер — это функция, анализирующая текст с целью выявления его логической структуры.
- Результат работы парсера — это представление выявленной структуры в виде некоторого конкретного типа данных.

Примеры

- Арифметическое выражение.
- URL.
- Поэма (главы, строфы, стихи).
- CSV-файл.
- ...

Содержание

- 1 Тип для парсера
- 2 Простейшие парсеры
- 3 Выбор и повторения
- 4 Разбор выражения, заданного грамматикой

Содержание

- 1 Тип для парсера
- 2 Простейшие парсеры
- 3 Выбор и повторения
- 4 Разбор выражения, заданного грамматикой

Тип для парсера

```
type Parser a = String -> a
```

+ непоглощённый вход

```
type Parser a = String -> (a, String)
```

+ возможность неудачи или нескольких результатов

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

+ возможность объявления экземпляров классов типов

```
newtype Parser a = Parser (String -> [(a, String)])
```

Тип для парсера: окончательная версия

```
newtype Parser a = Parser {apply :: String -> [(a, String)]}
```

```
ghci> :t apply  
apply :: Parser a -> String -> [(a, String)]
```

Функция parse

```
parse :: Parser a -> String -> a  
parse p = fst . head . apply p
```

Парсер как функтор

```

{-# LANGUAGE InstanceSigs #-}

newtype Parser a = Parser {apply :: String -> [(a, String)]}

instance Functor Parser where
  fmap :: (a -> b) -> Parser a -> Parser b
  fmap f p = ???

```

- Результатом `fmap` должен быть новый парсер (то есть функция в обёртке);
- функция `f` должна применяться к результатам применения парсера `p`.

```

instance Functor Parser where
  fmap f p = Parser $
    \s -> [ (f a, s') | (a, s') <- apply p s ]

```

Парсер как аппликативный функтор

```
newtype Parser a = Parser {apply :: String -> [(a, String)]}
```

```
instance Applicative Parser where
```

```
  pure :: a -> Parser a
```

```
  pure a = ???
```

```
  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

```
  f <*> p = ???
```

- pure: парсер, всегда возвращающий заданное значение;
- (<*>): нужно получить результаты первого парсера, затем второго, а после этого применить первые ко вторым.

```
instance Applicative Parser where
```

```
  pure a = Parser $ \s -> [(a, s)]
```

```
  f <*> p = Parser $
```

```
    \s -> [ (f' a, s'') | (f', s') <- apply f s,
                  (a, s'') <- apply p s' ]
```

Парсер как монада

```
newtype Parser a = Parser {apply :: String -> [(a, String)]}
```

```
instance Monad Parser where
```

```
  return = pure
```

```
  (>>=) :: Parser a -> (a -> Parser b) -> Parser b
```

```
  p >>= q = ???
```

- (>>=): парсер, который для каждого из возможных результатов x парсера p продолжает разбор парсером $(q\ x)$.

```
instance Monad Parser where
```

```
  return = pure
```

```
  p >>= q = Parser $
```

```
    \s -> [ (y, s'') | (x, s') <- apply p s,
```

```
              (y, s'') <- apply (q x) s']
```

Множество парсеров как моноид

- Нейтральный элемент: парсер, возвращающий пустой список (нет результата).
- Операция в моноиде: второй парсер используется, только если первый завершился неудачей (аналог моноида First для Maybe).

```
instance Alternative Parser where
  empty = Parser $ \s -> []
  p <|> q = Parser $
    \s -> let ps = apply p s in
           if null ps then apply q s else ps

instance MonadPlus Parser where
  mzero = empty
  mplus = (<|>)
```

Содержание

- 1 Тип для парсера
- 2 Простейшие парсеры**
- 3 Выбор и повторения
- 4 Разбор выражения, заданного грамматикой

Чтение символа

```

getc :: Parser Char
getc = Parser f
  where
    f [] = []
    f (c:cs) = [(c, cs)]

```

```

ghci> apply getc "12345"
[('1',"2345")]
ghci> apply (getc >> getc) "12345"
[('2',"345")]
ghci> apply (getc >> getc >> getc) "12345"
[('3',"45")]
ghci> apply ((,,) <$> getc <*> getc <*> getc) "12345"
[(('1','2','3'),"45")]

```

Чтение символа, удовлетворяющего предикату

```
sat :: (Char -> Bool) -> Parser Char
```

```
sat pr = do
```

```
  c <- getc
```

```
  guard $ pr c
```

```
  return c
```

```
ghci> apply (sat isLower) "abc"
```

```
[( 'a', "bc" )]
```

```
ghci> apply (sat isDigit) "123"
```

```
[( '1', "23" )]
```

```
ghci> apply (sat isDigit) "abc"
```

```
[]
```

```
lower :: Parser Char
```

```
lower = sat isLower
```

```
digit :: Parser Int
```

```
digit = digitToInt <$>
        sat isDigit
```

Конкретный символ и строка

```
char :: Char -> Parser ()  
char x = sat (==x) >> return ()
```

```
string :: String -> Parser ()  
string = mapM_ char
```

```
ghci> apply (char 'x') "abc"  
[]  
ghci> apply (char 'x') "xabc"  
[((), "abc")]  
ghci> apply (string "ab") "abc"  
[((), "c")]  
ghci> apply (string "ab") "xabc"  
[]
```

Пример: сумма цифр

Блок do

```
addition :: Parser Int
addition = do
  n <- digit
  char '+'
  m <- digit
  return $ n + m
```

Аппликативный стиль

```
addition = (+) <$> digit <*> (char '+' >> digit)
```

Содержание

- 1 Тип для парсера
- 2 Простейшие парсеры
- 3 Выбор и повторения**
- 4 Разбор выражения, заданного грамматикой

Выбор

```
ghci> apply (string "ab" <|> string "12") "abc"  
[((),"c")]  
ghci> apply (string "ab" <|> string "12") "123"  
[((),"3")]  
ghci> apply (string "ab" <|> string "12") "xxx"  
[]
```

Пример: разбор схемы URL

```
scheme = string "http" >> (char 's' <|> return ())
```

```
ghci> apply scheme "http"  
[((), "")]  
ghci> apply scheme "https"  
[((), "")]  
ghci> apply scheme "ftp"  
[]  
ghci> apply scheme "http:"  
[((), ":")]  
ghci> apply scheme "https:"  
[((), ":")]
```

Пример: цифра или сумма цифр

Версия 1 (неправильная)

```
ghci> let expr = digit <|> addition
ghci> apply expr "1"
[(1,"")]
ghci> apply expr "1+2"
[(1,"+2")]
```

Версия 2 (неэффективная)

```
ghci> let expr = addition <|> digit
ghci> apply expr "1"
[(1,"")]
ghci> apply expr "1+2"
[(3,"")]
```

Пример: цифра или сумма цифр

Версия 3 (наилучшая, двухэтапная)

```

addition = digit >>= rest
  where
    rest m = (+m) <$> (char '+' >> digit) <|> return m

```

```

ghci> apply addition "1+2"
[(3,"")]
ghci> apply addition "1"
[(1,"")]
ghci> apply addition "1+2+3"
[(3,"+3")]
ghci> apply addition "abc"
[]

```

Повторение: строчные буквы

```
lowers :: Parser String
lowers = (:) <$> lower <*> lowers <|> return ""
```

```
ghci> apply lowers "isUpper"
[("is","Upper")]
ghci> apply lowers "Upper"
[("", "Upper")]
```

Повторение: общий случай

```
many :: Parser a -> Parser [a]
many p = many1 p <|> return []
```

```
many1 :: Parser a -> Parser [a]
many1 p = (:) <$> p <*> many p
```

```
ghci> apply (many digit) "123"
[[1,2,3], ""]
ghci> apply (many digit) ""
[[[]], ""]
ghci> apply (many1 digit) "123"
[[1,2,3], ""]
ghci> apply (many1 digit) ""
[]
```

Примеры

```
space :: Parser ()
space = many (sat isSpace) >> return ()

natural :: Parser Int
natural = foldl1 (\m n -> m * 10 + n) <$> many1 digit

integer :: Parser Int
integer = (*) <$> minus <*> natural
  where
    minus = (char '-' >> return (-1)) <|> return 1
```

Повторения с разделителем

```
optional :: a -> Parser a -> Parser a  
optional v p = p <|> return v
```

```
sepBy1 :: Parser a -> Parser a1 -> Parser [a]  
sepBy1 p sep = (:) <$> p <*> many (sep >> p)
```

```
sepBy :: Parser a -> Parser a1 -> Parser [a]  
sepBy p sep = optional [] (sepBy1 p sep)
```

Пример: разбор списка целых чисел

```
symbol s = space >> string s
```

```
token p = space >> p
```

```
bracket op cl p = do
```

```
  symbol op
```

```
  x <- p
```

```
  symbol cl
```

```
  return x
```

```
intList = bracket "[" "]" $  
           sepBy (token integer) (symbol ",")
```

```
ghci> apply intList "[ 2 , 4, 54 , 0]"  
[[2,4,54,0],""]
```

Пример: разбор URL (упрощённо)

```
data Scheme = FTP | HTTP | HTTPS | Unk String
           deriving Show
```

```
type Server = String
```

```
type Path = String
```

```
data URL = URL Scheme Server Path
         deriving Show
```

```
scheme = (string "https" >> return HTTPS) <|>
         (string "http" >> return HTTP) <|>
         (string "ftp" >> return FTP) <|>
         Unk <$> lowers
```

```
url = URL <$> scheme <*>
      (string "://" >> many1 (sat (/='/'))) <*>
      many getc
```

Пример: разбор URL (упрощённо)

```
ghci> apply url "http://mmcs.sfedu.ru/photo"  
[(URL HTTP "mmcs.sfedu.ru" "/photo", "")]  
ghci> apply url "xxx://mmcs.sfedu.ru/"  
[(URL (Unk "xxx") "mmcs.sfedu.ru" "/", "")]  
ghci> apply url "12345"  
[]
```

- На чём ломается третий пример?

Содержание

- 1 Тип для парсера
- 2 Простейшие парсеры
- 3 Выбор и повторения
- 4 Разбор выражения, заданного грамматикой**

План

- 1 Тип данных для представления выражения.
- 2 Грамматика (в нотации Бэкуса—Наура).
- 3 Парсер.

Выражения с полным набором скобок

```
data Expr = Con Int | Bin Op Expr Expr
  deriving Show
```

```
data Op = Plus | Minus
  deriving Show
```

```
expr ::= nat | '(' expr op expr ')'
op    ::= '+' | '-'
nat   ::= {digit}+
digit ::= '0' | '1' | '2' | ... | '9'
```

```

expr ::= nat | '(' expr op expr ')'
op   ::= '+' | '-'
nat  ::= {digit}+
digit ::= '0' | '1' | '2' | ... | '9'

```

```

expr :: Parser Expr
expr = token (<|> bracket "(" ")" binary)
  where
    constant = Con <$> natural
    binary = do
      e1 <- expr
      p <- op
      e2 <- expr
      return $ Bin p e1 e2
    op = (symbol "+" >> return Plus) <|>
         (symbol "-" >> return Minus)

```

Выражения с полным набором скобок: примеры

```

ghci> parse expr "(2+3)"
Bin Plus (Con 2) (Con 3)
ghci> parse expr "( 2 + 3)"
Bin Plus (Con 2) (Con 3)
ghci> parse expr "((2+3)-((2-2)+1))"
Bin Minus
  (Bin Plus (Con 2) (Con 3))
  (Bin Plus
    (Bin Minus (Con 2) (Con 2))
    (Con 1))

```

Выражения с опущенными скобками (1)

```

expr ::= expr op term | term
term ::= nat | '(' expr ')'

```

```

expr = token (binary <|> term)

```

```

where

```

```

  binary = do

```

```

    e1 <- expr

```

```

    p <- op

```

```

    e2 <- term

```

```

    return $ Bin p e1 e2

```

```

term = token (constant <|> bracket "(" ")" expr)

```

Зацикливание!

Поможет?

```

expr = token (term <|> binary)

```

Нет!

Выражения с опущенными скобками (2)

```
expr ::= term {op term}*
```

```
expr = token (term >>= rest)
```

```
  where
```

```
    rest e1 = optional e1 $ do
```

```
      p <- op
```

```
      e2 <- term
```

```
      rest $ Bin p e1 e2
```

Выражения с четырьмя операциями

```
data Expr = Con Int | Bin Op Expr Expr
  deriving Show
data Op = Plus | Minus | Mul | Div
  deriving Show
```

```
expr ::= term {addop term}*
term  ::= factor {mulop factor}*
factor ::= nat | '(' expr ')'
addop  ::= '+' | '-'
mulop  ::= '*' | '/'
```

```
expr = token (term >>= rest)
```

```
  where
```

```
    rest e1 = optional e1 $ do
```

```
      p <- addop
```

```
      e2 <- term
```

```
      rest $ Bin p e1 e2
```

```
  term = token (factor >>= more)
```

```
  more e1 = optional e1 $ do
```

```
    p <- mulop
```

```
    e2 <- factor
```

```
    more $ Bin p e1 e2
```

```
  factor = token (constant <|> bracket "(" ")" expr)
```

```
  addop = (symbol "+" >> return Plus) <|>
```

```
        (symbol "-" >> return Minus)
```

```
  mulop = (symbol "*" >> return Mul) <|>
```

```
        (symbol "/" >> return Div)
```

```
  constant = Con <$> natural
```

```
expr ::= term {addop term}*
term ::= factor {mulop factor}*
factor ::= nat | '(' expr ')'
```

```
addop ::= '+' | '-'
```

```
mulop ::= '*' | '/'
```

Похожие функции

- rest и more
- addop и mulop

Выражения с четырьмя операциями (последняя версия)

```

expr = token (term >>= rest addop term)
  where
    rest op unit e1 = optional e1 $ do
      p <- op
      e2 <- unit
      rest op unit $ Bin p e1 e2
    term = token (factor >>= rest mulop factor)
    factor = token (constant <|> bracket "(" ")" expr)
    addop = binop ("+", Plus) ("-", Minus)
    mulop = binop ("*", Mul) ("/", Div)
    binop (s1, cons1) (s2, cons2) =
      (symbol s1 >> return cons1) <|>
      (symbol s2 >> return cons2)
    constant = Con <$> natural

```

Выражения с четырьмя операциями: пример

```
ghci> parse expr "2*(2-3)+2/3"
Bin Plus
  (Bin Mul
    (Con 2)
    (Bin Minus (Con 2) (Con 3)))
  (Bin Div (Con 2) (Con 3))
```

- ① R. Bird. Thinking Functionally with Haskell (chap. 11, based on 'Monadic Parsing in Haskell' by Graham Hutton and Eric Meijer).