
Использование и архитектура Parsec

Дмитрий Тимофеев

Санкт-Петербургская группа пользователей
Haskell

15 декабря 2007 г.

Синтаксический анализ

- Задачи:
 - Проверка правильности исходных данных - соответствие текста грамматике
 - Содержательная обработка исходных данных
-

ПОДХОДЫ

- Полностью ручная реализация
 - Генераторы парсеров (YACC, Narry)
 - Комбинаторные библиотеки
 - “List of successes”
 - Монадические
 - Parsec
-

Монадические парсеры

type Parser a

return :: a -> Parser a

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

satisfy :: (Char -> Bool) -> Parser Char

(<|>) :: Parser a -> Parser a -> Parser a

Parsec

```
import Text.ParserCombinators.Parsec
```

```
simple :: Parser Char
```

```
simple = letter
```

```
> parseTest simple "a"
```

```
'a'
```

Примитивные парсеры: *letter, char, string, ...*

Последовательность парсеров

bracketedLetter :: Parser Char

```
bracketedLetter = do { char '('  
                      ; c <- letter  
                      ; char ')'   
                      ; return c  
                      }
```

Последовательность парсеров

> parseTest bracketedLetter "(a)"

'a'

> parseTest bracketedLetter "(ab)"

parse error at (line 1, column 3):

unexpected "b"

expecting ")"

Альтернативы

```
parens :: Parser ()  
parens = do { char '('  
            ; parens  
            ; char ')'   
            ; parens  
            }  
<|> return ()
```

Предпросмотр

```
test1 = string "OCaml" <|> string "Haskell"
```

```
test2 = string "(lisp)" <|> string "(scheme)"
```

```
> parseTest test1 "Haskell"
```

```
"Haskell"
```

```
> parseTest test2 "(lisp)"
```

```
"(lisp)"
```

Предпросмотр

```
test1 = string "OCaml" <|> string "Haskell"
```

```
test2 = string "(lisp)" <|> string "(scheme)"
```

```
> parseTest test2 "(scheme)"
```

```
parse error at (line 1, column 1):
```

```
unexpected "s"
```

```
expecting "(lisp)"
```

Предпросмотр

- Причина: Parser строит LL(1)-парсер
 - Ограничение: решение принимается на основании анализа только первого символа строки
 - Осознанный выбор: эффективность
 - Есть возможность преодолеть ограничения LL(1)
-

Предпросмотр

Вариант 1:

```
test2 = do { char '('  
            ; s <- string "lisp" <|> string "scheme"  
            ; char ')'   
            ; return ( "(" ++ s ++ ")" )  
            }
```

Предпросмотр

Вариант 2:

```
test2 = try (string "(lisp)") <|> string "(scheme)"
```

Комбинатор *try* так изменяет наш парсер, что он в случае неудачи оставляет уже прочитанные СИМВОЛЫ В ПОТОКЕ

Архитектурные принципы

- Эффективность: отказ от затрат на поддержку неограниченного предпросмотра, LL(1) по умолчанию, можно увеличить количество просматриваемых символов, используя *try*
 - Информативность сообщений об ошибках
-

Ограничение предпросмотра

- В конструкции $p \langle | \rangle q$ парсер q в принципе не вызывается, если p обработал больше одного символа
- Каждый парсер отслеживает, сколько символов он обработал

Ограничение предпросмотра

Упрощенная реализация:

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

```
data Consumed a = Consumed (Reply a)  
                | Empty (Reply a)
```

```
data Reply a = Ok a String | Error
```

Базовые комбинаторы

`return x = \input -> Empty (Ok x input)`

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

`satisfy test = \input -> case (input) of`

`[] -> Empty Error`

`(c:cs) | test c -> Consumed (Ok c cs)`

`| otherwise -> Empty Error`

Базовые комбинаторы

char c = satisfy (==c)

letter = satisfy isAlpha

digit = satisfy isDigit



Базовые комбинаторы

$(\gg=) :: \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b$

$p \gg= f = \backslash \text{input} \rightarrow \text{case } (p \text{ input}) \text{ of}$

Empty reply1

$\rightarrow \text{case } (\text{reply1}) \text{ of}$

Ok x rest $\rightarrow ((f \ x) \ \text{rest})$

Error $\rightarrow \text{Empty Error}$

...

Базовые комбинаторы

Consumed reply1

-> *Consumed*

(case (reply1) of

Ok x rest -> case ((f x) rest) of

Consumed reply2 -> reply2

Empty reply2 -> reply2

error -> error)

Базовые комбинаторы

$(\langle | \rangle) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a$

$p \langle | \rangle q = \backslash \text{input} \rightarrow \text{case } (p \text{ input}) \text{ of}$

$\text{Empty Error} \rightarrow (q \text{ input})$

$\text{Empty ok} \rightarrow \text{case } (q \text{ input}) \text{ of}$

$\text{Empty } _ \rightarrow \text{Empty ok}$

$\text{consumed} \rightarrow \text{consumed}$

$\text{consumed} \rightarrow \text{consumed}$

try

try :: Parser a -> Parser a

try p = \input -> case (p input) of

Consumed Error -> Empty Error

other -> other

Обработка ошибок

- К типу `Parser` добавляется состояние:
 - `type Parser a = State -> Consumed a`
 - `data State = State String Pos`
- К результатам разбора добавляется сообщение об ошибках, причем к удачному разбору – тоже
 - `data Message = Message Pos String [String]`
 - `data Reply a = Ok a State Message | Error Message`

Обработка ошибок

- Меняется реализация `return`, `satisfy`, (`<|>`) – [Leijen, Meijer, 2001]
 - В результате получаем возможность приписывать парсерам информативные обозначения – комбинатор `<?>`
-

Семантика

nesting :: Parser Int

```
nesting = do { char '('  
              ; n <- nesting  
              ; char ')'   
              ; m <- nesting  
              ; return (max (n + 1) m)  
            } <|> return 0
```

Дополнительные возможности

- Лексический анализ:
 - Лексический анализ объединяется с синтаксическим анализом
 - Лексический анализ выполняется отдельными специализированными парсерами
 - Реализованными в библиотеке
 - Разработанными самостоятельно с помощью комбинаторов
 - Использование отдельного сканера, параметризация по типу потока (не только Char)
- Возможность генерации парсера выражений по набору операций с учетом ассоциативности и приоритета

Источники информации

- Сайт Parsec:
<http://legacy.cs.uu.nl/daan/parsec.html>
 - Daan Leijen. Parsec, a fast combinator parser, 2001
 - Daan Leijen, Erik Meijer. Parsec: Direct Style Monadic Parser Combinators For The Real World, 2001
-