Комбинаторы парсеров

Михаил Беляев

20 декабря 2019 г.

История

Комбинаторы парсеров (также называемые «монадными комбинаторами парсеров») впервые появились в 2008 году в статье Frost, Hafiz & Callaghan

Референсная реализация кода к статье была написана на Haskell

Зачем вообще это?

Комбинаторы парсеров позволяют:

- Разбивать разбор грамматик на основе рекурсивного спуска на модули
- Решить при этом (частично) проблему разбора контекстно-зависимых грамматик за полиномиальное время

Кроме того, отдельные части становятся предельно простыми в сравнении с традиционными ad-hoc парсерами и средствами генерации парсеров

Библиотеки

На данный момент самой известной является библиотека `Parsec`, самой быстрой — библиотека `Attoparsec`

Существует большое количество подражателей в других языках:

- `JParsec` и `ParsecJ` для Java
- `Bennu` для Javascript
- `Parsy`, `Picoparse`, `FuncparserLib` для Python
- `Boost::Spirit` и другие для C++
- Стандартные библиотеки Scala, F# и некоторых других языков
- и т.д.

Основная идея

- Есть базовый набор простых парсеров, которые умеют разбирать примитивный ввод
  - «Любой символ в диапазоне от „0“ до „9“»
  - «Любой из символов заданной строки»
  - «Пробел любого вида»
- Есть набор простых комбинаторов, позволяющих на основе одних парсеров строить другие
  - «Парсер A 1 или больше раз подряд»
  - «Парсер A, перемежаемый парсером B»
  - «Попробуй парсер A, если не получилось — попробуй B»

Если это всё вам напомнило регулярные выражения — то не зря напомнило

Основная идея

Важно: у любого парсера есть **результат**

Результат может быть любого типа $\Rightarrow$ парсер — это тип с параметром

Например

```
data Result a = Success a | Failure Error
type Parser a = String -> (a, Maybe String)
```

На практике парсеры устроены сложнее:

- Строки читать не очень хорошо
- Нужно следить за позицией во входном потоке
- Ошибки должны быть говорящими

Простые парсеры

```haskell
-- Принимаем 1 символ, возвращаем его же
char :: Char -> Parser Char
-- Принимаем символы из диапазона
range :: Char -> Char -> Parser Char
-- Принимаем символы из списка
oneOf :: [Char] -> Parser Char
```

Простые парсеры

```haskell
-- Принимаем 1 символ, возвращаем его же
char :: Char -> Parser Char
-- Принимаем символы из диапазона
range :: Char -> Char -> Parser Char
-- Принимаем символы из списка
oneOf :: [Char] -> Parser Char
```

Или, более общий вариант

```haskell
conforms :: (Char -> Bool) -> Parser Char
char c = conforms (== c)
range a b = conforms (\c -> c >= a && c <= b)
oneOf cs = conforms (\c -> c `elem` cs)
```

Простые парсеры: квазипарсеры

```haskell
-- Не пытаться ничего читать,
-- сразу вернуть ошибку
parserError :: Error -> Parser a
-- Не пытаться ничего читать,
-- сразу вернуть значение
parserSuccess :: a -> Parser a
```

Парсер-комбинаторы: простые

```haskell
-- Всё, кроме заданного
parserNot :: (Parser Char) -> Parser Char
-- Попробовать первый, если не сработал, то второй
parserOr :: Parser a ->
            Parser b ->
            Parser (Either a b)
```

Парсер-комбинаторы: коллекции

```haskell
-- Несколько элементов подряд
parserSeq :: Parser a ->
             Parser b ->
             Parser (a,b)
-- Много (0 или больше) элементов подряд
parserMany :: Parser a -> Parser [a]
```

Парсер-комбинаторы: монадные операции

```haskell
-- Парсер является функтором!
parserMap :: (a -> b) -> Parser a -> Parser b
-- Парсер является монадой!
parserReturn = parserSuccess
parserFail = parserError
parserFlatten :: (Parser (Parser a)) -> Parser a
```

Парсер-комбинаторы: try

```
parserTry :: Parser a -> Parser a
```

Особенность реализации: иногда нужно попробовать разобрать текст и в случае неудачи вернуться туда, откуда был начат разбор

Синтаксический сахар для употребимых операций

```
space :: Parser ()
eof :: Parser ()
digit :: Parser Char
alpha :: Parser Char
upper :: Parser Char
...
```

Пример: разбор номера группы

```
data GroupNumber = GroupNumber Integer Integer
parseNumber = read <$> parserMany1 digit
parseGroupNumber = do{ dept <- parseNumber;
                       char '/';
                       gp <- parseNumber;
                       return $ GroupNumber dept gp  }
parseManyGNs = do{ gn <- parseGroupNumber;
                   gns <- parserMany parseGroupNumber';
                   return (gn: gns) }
                 where
                  parseGroupNumber' = do { char ',';
                                           parseGroupNumber }
```

Практическая часть: библиотека `Parsec`

- Парсер — это не просто монада, а монада-трансформер
- Позволяет в процессе разбора использовать своё состояние
- Поддерживает чтение из множества разных типов
- `parserOr` — это `<|>` из `Alternative`
- В остальном все парсер-комбинаторы похожи

Практическая часть: библиотека `Parsec`

```
data ParsecT s u m a
type Parsec s u a = ParsecT s u Identity a
-- s - входные данные
-- u - пользовательское состояние
-- m - внутренняя монада
-- a - собственно параметр
```

Packrat parsing

- Оптимизация, которая позволяет парсить что угодно за гарантированные $O(N)$
- Работает за счёт ленивости

http://kspt.icc.spbstu.ru/course/lang

belyaev@kspt.icc.spbstu.ru

КСПТ | КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ СПбПУ

ПОЛИТЕХ
Санкт-Петербургский политехнический университет Петра Великого