Парсер-комбинаторы и левая рекурсия

введение

Автор: Екатерина Вербицкая

Лаборатория языковых инструментов JetBrains
Санкт-Петербургский государственный университет
Математико-механический факультет

2 ноября 2015г.

Синтаксический анализ

Сопоставление строки терминалов формальной грамматике языка

- Восходящие
- Нисходящие

- Написанные руками
- Генерируемые по спецификации грамматики
- Написанные руками, но с использованием некого инструментария

Парсер-комбинаторы: общая идея

- Написать примитивные парсеры (например, для обработки отдельных символов)
- Написать комбинаторы для составления из примитивных парсеров более сложных
  - Последовательности
  - Альтернативы
  - Повторение
  - Опциональный парсер
  - ...
- С использованием этого инструментария описать необходимый парсер

Парсер-комбинаторы: тип парсера

$$Parser \quad = \quad String \rightarrow Tree$$

Парсер-комбинаторы: тип парсера

$$
\begin{aligned}
Parser &= String \to Tree \\
Parser &= String \to (Tree, String)
\end{aligned}
$$

Парсер-комбинаторы: тип парсера

$$
\begin{aligned}
Parser &= String \to Tree \\
Parser &= String \to (Tree, String) \\
Parser &= String \to [(Tree, String)]
\end{aligned}
$$

Парсер-комбинаторы: тип парсера

$$
\begin{aligned}
Parser &= String \to Tree \\
Parser &= String \to (Tree, String) \\
Parser &= String \to [(Tree, String)]
\end{aligned}
$$

$$
Parser\ a = String \to [(a, String)]
$$

Примитивные парсеры

```
result :: a -> Parser a
result v = \inp -> [(v,inp)]

zero :: Parser a
zero = \inp -> []

item :: Parser Char
item = \inp -> case inp of
                 [] -> []
                 (x:xs) -> [(x,xs)]
```

Парсер-комбинаторы

```
bind :: Parser a -> (a -> Parser b) -> Parser b
p `bind` f = \inp -> concat [f v inp' | (v,inp') <- p inp]

p `seq` q = p `bind` \x ->
            q `bind` \y ->
            result (x,y)

sat :: (Char -> Bool) -> Parser Char
sat p = item `bind` \x ->
        if p x then result x else zero

plus :: Parser a -> Parser a -> Parser a
p `plus` q = \inp -> (p inp ++ q inp)
```

Маленькие полезные парсеры

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

digit :: Parser Char
digit = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower = sat (\x -> 'a' <= x && x <= 'z')

upper :: Parser Char
upper = sat (\x -> 'A' <= x && x <= 'Z')
```

Менее маленькие полезные парсеры

```
letter :: Parser Char
letter = lower `plus` upper

alphanum :: Parser Char
alphanum = letter `plus` digit

word :: Parser String
word = neWord `plus` result ""
       where
         neWord = letter `bind` \x ->
                  word `bind` \xs ->
                  result (x:xs)

word "Yes!" = [("Yes","!"), ("Ye","s!"),
               ("Y","es!"), ("","Yes!")]
```

Монадические парсер-комбинаторы

```
class Monad m where
  result :: a -> m a
  bind :: m a -> (a -> m b) -> m b

instance Monad Parser where
  -- result :: a -> Parser a
  result v = \inp -> [(v,inp)]

  -- bind :: Parser a -> (a -> Parser b) -> Parser b
  p `bind` f = \inp -> concat [f v out | (v,out) <- p inp]
```

Монадические парсер-комбинаторы

```
class Monad m => MonadOPlus m where
  zero :: m a
  (++) :: m a -> m a -> m a

instance MonadOPlus Parser where
  -- zero :: Parser a
  zero = \inp -> []

  -- (++) :: Parser a -> Parser a -> Parser a
  p ++ q = \inp -> (p inp ++ q inp)
```

Другая форма записи

```
p1 `bind` \x1 ->
p2 `bind` \x2 ->
...
pn `bind` \xn ->
result (f x1 x2 ... xn)
```

```
[ f x1 x2 ... xn | x1 <- p1
                 , x2 <- p2
                 , ...
                 , xn <- pn ]
```

```
string :: String -> Parser String
string "" = [""]
string (x:xs) = [x:xs | _ <- char x, _ <- string xs]
```

Монадические парсер-комбинаторы

```
sat :: (Char -> Bool) -> Parser Char
sat p = [x | x <- item, p x]

many :: Parser a -> Parser [a]
many p = [x:xs | x <- p, xs <- many p] ++ [[]]


ident :: Parser String
ident = [x:xs | x <- lower, xs <- many alphanum]
```

Примеры парсеров

```
many1 :: Parser a -> Parser [a]
many1 p = [x:xs | x <- p, xs <- many p]

nat :: Parser Int
nat = [eval xs | xs <- many1 digit]
      where
        eval xs = foldl1 op [ord x - ord '0' | x <- xs]
        m `op` n = 10*m + n

int :: Parser Int
int = [-n | _ <- char '-', n <- nat] ++ nat

int :: Parser Int
int = [f n | f <- op, n <- nat]
      where
        op = [negate | _ <- char '-'] ++ [id]
```

Пример: список чисел

```
ints :: Parser [Int]
ints = [n:ns | _ <- char '['
             , n <- int
             , ns <- many [x | _ <- char ',', x <- int]
             , _ <- char ']']


sepby1 :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = [x:xs | x <- p
                       , xs <- many [y | _ <- sep, y <- p]]

ints = [ns | _ <- char '['
           , ns <- int `sepby1` char ','
           , _ <- char ']']
```

Список чисел: еще короче

```
bracket :: Parser a -> Parser b -> Parser c -> Parser b
bracket open p close = [x | _ <- open, x <- p, _ <- close]

ints = bracket (char '[')
               (int `sepby1` char ',')
               (char ']')
```

Пример: арифметические выражения

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{expr}\ \mathit{addop}\ \mathit{factor} \mid \mathit{factor} \\
\mathit{addop} & ::= & + \mid - \\
\mathit{factor} & ::= & \mathit{nat} \mid (\ \mathit{expr}\ )
\end{array}
$$

```
expr :: Parser Int
addop :: Parser (Int -> Int -> Int)
factor :: Parser Int

expr = [f x y | x <- expr
              , f <- addop
              , y <- factor] ++ factor

addop = [(+) | _ <- char '+'] ++ [(-) | _ <- char '-']

factor = nat ++ bracket (char '(') expr (char ')')
```

Преимущества монадических парсер-комбинаторов

- Простота
- Гибкость
- Выразительность
- Возможность откатываться (backtracking)
- Лексический анализ не нужно выделять в отдельный шаг
- Можно считать семантику во время синтаксического анализа

Недостатки монадических парсер-комбинаторов

- Если использовать неграмотно, можно получить непредсказуемое время работы и легко исчерпать всю доступную память
  - Наличие общих префиксов у нескольких правил. Решение: факторизация грамматики
  - Вычисление промежуточных результатов, где не было надо. Решение: использование ленивости (например, $p +\!\!+\!\!+\, q = \mathit{first}(p +\!\!+ q)$)

Packrat-парсер: не совсем парсер-комбинаторы

- Нисходящий
- Использует преимущества ленивости и функционального стиля
- Unlimited lookahed + backtracking
- Гарантирует линейное время работы на LL(k) и LR(k) грамматиках
- Использует запоминание

Packrat-парсер: пример

Грамматика языка арифметических выражений

$$
\begin{aligned}
E &:: M + E \mid M \\
M &:: P * M \mid P \\
P &:: (\, E \,) \mid D \\
D &:: 0 \mid 1 \mid \ldots \mid 9
\end{aligned}
$$

Типы классического нисходящего парсера

```
data Result v = Parsed v String
              | NoParse
```

Типы для правил грамматики выражений:

```
pE :: String -> Result Int
pM :: String -> Result Int
pP :: String -> Result Int
pD :: String -> Result Int
```

Классический нисходящий парсер выражений

```
pE s = alt1 where
    alt1 = case pM s of
             Parsed vleft s' ->
               case s' of
                 ('+':s'') ->
                   case pE s'' of
                     Parsed vright s''' ->
                       Parsed (vleft + vright) s'''
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    alt2 = case pM s of
             Parsed v s' -> Parsed v s'
             NoParse -> NoParse
```

Проблемы классического нисходящего парсинга

- При откатах одно и то же может вычисляться несколько раз
- В худшем случае экспоненциальное время работы
- Факторизация грамматики – выход, но не очень хороший

- Решение: запоминать промежуточные результаты в таблице

Таблица

```
data Derivs = Derivs { dvE :: Result Int,
                       dvM :: Result Int,
                       dvP :: Result Int,
                       dvD :: Result Int,
                       dvC :: Result Char }
```

Типы

```
data Result v = Parsed v Derivs
              | NoParse


pE :: Derivs -> Result Int
pM :: Derivs -> Result Int
pP :: Derivs -> Result Int
pD :: Derivs -> Result Int
```

Парсер для выражения

```
pE d = alt1 where
    alt1 = case dvM d of
             Parsed vleft d' ->
               case dvC d' of
                 Parsed '+' d'' ->
                   case dvE d'' of
                     Parsed vright d''' ->
                       Parsed (vleft + vright) d'''
                     _ -> alt2
                 _ -> alt2
             _ -> alt2

    alt2 = dvM d
```

Парсер

```
parse :: String -> Derivs
parse s = d where
    d    = Derivs add mult prim dec chr
    add  = pE d
    mult = pM d
    prim = pP d
    dec  = pD d
    chr  = case s of
             (c:s') -> Parsed c (parse s')
             []     -> NoParse
```

Монадический Packrat-парсер

```
newtype Parser v = Parser (Derivs -> Result v)

instance Monad Parser where

  (Parser p1) >>= f2 = Parser pre
          where pre d = post (p1 d)
                post (Parsed v d') = p2 d'
                  where Parser p2 = f2 v
                post (NoParse) = NoParse

  return x = Parser (\d -> Parsed x d)

  fail msg = Parser (\d -> NoParse)
```

Монадический Packrat-парсер

Альтернатива:

```
(<|>) :: Parser v -> Parser v -> Parser v
(Parser p1) <|> (Parser p2) = Parser pre
                where pre d = post d (p1 d)
                      post d NoParse = p2 d
                      post d r = r

Parser pE = (do vleft <- Parser dvM
                char '+'
                vright <- Parser dvE
                return (vleft + vright))
        <|> (do Parser dvM)
```

Преимущества и недостатки

- Линейное время работы, простота, выразительность, но:

- Можно реализовать только детерминированные парсеры
- Нет состояний
- Потребляет достаточно много памяти
- Проблемы с левой рекурсией

Борьба с левой рекурсией: Warth et al

- Первый раз, когда наткнулись на левую рекурсию, запомнить в таблице ошибку анализа — посадить семечко
- Откатиться во входном потоке в начало правила и применить правило еще раз — прорастить семечко

- Определение левой рекурсии:
  - В таблице наряду с вычисленным значением храним галочку: леворекурсивно ли это правило
  - Перед применением правила, выставляем галочку в false
  - Если в таблице нет вычисленного значения, значит попали в левую рекурсию: выставляем галочку в true, садим семечко

Warth et al: неявная рекурсия

Пример: разбираем "4 – 3"относительно грамматики

```
X :: E
E :: X - Num | Num
```

При обработке X посадили семечко: $X \to E \to 4$
При проращивании семечка (повторном вызове парсера X) получаем уже лежащее в таблице значение $(X \to E \to 4)$
Решение:

- Храним стек вызовов правил
- При проращивании семечка проращиваем все вовлеченные в левую рекурсию правила

Проблемы с подходом Warth et al

Проблемы с ассоциативностью, если есть и левая, и правая рекурсия

```
E :: E - E | Num
```

"1 – 2 – 3"разберется как "1 – (2 – (3))". Однако если переписать грамматику следующим образом, ассоциативность будет правильной

```
E :: E - E (- E)? | Num
```

Частичное решение: ограничить правую рекурсию максимум одним шагом в глубину

Литература

- Monadic Parser Combinators:
  `http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf`
- Packrat Parsing:
  `http://www.brynosaurus.com/pub/lang/packrat-icfp02.pdf`
- Packrat Parsers Can Support Left Recursion:
  `http://www.vpri.org/pdf/tr2007002_packrat.pdf`
- Проблемы с предыдущим подходом (Direct Left-Recursive Parsing Expression Grammars):
  `http://tratt.net/laurie/research/pubs/papers/tratt_direct_left_recursive_parsing_expression_grammars.pdf`