



---

# 使用 Haskell 编写灵活的 Parser (下)

2017, Jul 27 by Tesla Ice Zhang

本篇主要讲 Parser Combinator 的组合，比如二元运算符的解析以及优先级结合性的处理。

## some 与 many

上篇还剩下一个内容，就是 `some` 和 `many`。这两个函数是 `Control.Applicative` 内置的，可以直接拿来用。

比如我们可以有一个解析空字符串的 Parser：

```
-- | one or more
spacesP :: Parser String
spacesP = some $ oneOf " \n\r\t"

-- | zero or more
spaces0P :: Parser String
spaces0P = many $ oneOf " \n\r\t"
```

非常简单，对吧。它们看起来也很直观。

## 其他小工具

---

另外，为下文做准备，我们先加上这几个 Parser。他们的意义看名字就知道，不再介绍。

```
import Data.Char

stringP :: String -> Parser String
stringP [] = return []
stringP (c : cs) = do
    charP c
    stringP cs
    return $ c : cs
--

reservedP :: String -> Parser String
reservedP = tokenP . stringP

natP :: Parser Int
natP = read <$> some digitP

digitP :: Parser Char
digitP = satisfy isDigit

tokenP :: Parser a -> Parser a
tokenP p = do
    a <- p
    spaces0P
```

```
    return a
```

```
--
```

## chainl1 与 chainr1

首先，我们考虑这样两种 Parser。

1. 解析一个运算单元

2. 解析一个运算符

看起来是个半群，不过这并不是重点。我们以整数和加减法作为具体的例子，然后可以考虑写出对应的 Parser。

```
-- the parsers
```

```
numberP :: Parser String
```

```
numberP = do
```

```
    s <- stringP "-" <|> return []
```

```
    cs <- some digitP
```

```
    spaces0P
```

```
    return $ s ++ cs
```

```
--
```

```
plusSymbolP :: Parser String
```

```
plusSymbolP = do
```

```
    stringP "+"
```

```
    spaces0P
```

```
    return "+"
```

## AST

沿用一下上次类似的 AST 表述。

```
{-# LANGUAGE DeriveFunctor #-}
```

```
data OpTree a b = Term b
                | Op (OpTree a b) a (OpTree a b)
    deriving (Show, Eq, Functor)
```

这个也很清晰，比如 1+1 的 AST 就是：


```
Op (Term "1") "+" (Term "1")
```

## 问题

---

但是我们这里有一个巨大的问题。当我们在描述一颗复杂语法树的时候，比如 1+1+1+1 这种好几个二元运算符组合起来，我们需要

```
p = do
  n1 <- numberP
  plusSymbolP
  n2 <- numberP
  plusSymbolP
  n3 <- numberP
  plusSymbolP
  n4 <- numberP
  return $ Op (Op (Op (Term "1") "+" (Term "1"))) "+" (1
```



这种复杂而傻逼的 Parser。

## 抽象

---

为了解决这个问题，我们首先重构一下 `plusSymbolP`，把所有的二元运算抽象成一个新的 Parser。

```
-- | binary operators
binOp :: String -> (b -> b -> b) -> Parser (b -> b -> b)
binOp sym func = do
  stringP sym
  return func
```

当然你也可以不使用 `do notation`，毕竟这个很简单。

```
-- | binary operators
binOp :: String -> (b -> b -> b) -> Parser (b -> b -> b)
binOp s = (stringP s >>) . return
```

上面是半 point-free 的版本。

这个函数的意思是：

1. 拿到运算符和 通过结合运算符左右的两个语法树节点生成新的语法树节点的函数（此处是 `\l r -> Op l "+" r`）
2. 解析运算符
3. 如果解析成功，那么返回这个函数

## 重写

---

于是我们的加法符号解析器，以及剩下的四则运算就可以很简单地重写：

```
addOp = binOp "+" $ \x y -> Op x "+" y
subOp = binOp "-" $ \x y -> Op x "-" y
mulOp = binOp "*" $ \x y -> Op x "*" y
divOp = binOp "/" $ \x y -> Op x "/" y
```

要解析一个 `l op r`（左边 `op` 是任意一个二元运算符）也就可以这样：

```
operatorP :: Parser (a -> a -> a) -> Parser a
operatorP op = do
  l <- numberP
  o <- op
  r <- numberP
  return $ o l r
```

只需要传入 `op` 就能构造一个二元运算符。

## 左结合

---

同理。我们可以进行『不断地尝试往右边解析运算符，直到解析不了，返回表达式链』。

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = do
  a <- p
  rest a
  where
```

```

rest a = (do
  f <- op
  b <- p
  rest $ f a b)
<|> return a
--

```

上面这个实现基本上是你能想到的最简单最朴素好像也是唯一的实现。

比如链状的 + 操作：

```
adds = chainl1 numberP addOp
```

它可以这样工作：

```

Main> parseCode adds "1+1+1+1"
Op (Op (Op (Term "1") "+" (Term "1")) "+" (Term "1")) '+' (Term "1")
Main> parseCode adds "1+1+1"
Op (Op (Term "1") "+" (Term "1")) "+" (Term "1")
Main> parseCode adds "1+1"
Op (Term "1") "+" (Term "1")

```

我们也可以同时加入减法：

```
addAndSubs = chainl1 numberP $ addOp <|> subOp
```

这个读者可以自行尝试。

## 右结合

这里也有一个最简最智障实现：

```
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainr1 p op = scan
  where
    scan = do
      a <- p
      rest a
    rest a = (do
      f <- op
      b <- scan
      rest $ f a b)
    <|> return a
--
```

举例比如乘方：

```
sq = chainr1 numberP $ binOp "^" $ \x y -> Op x "^" y
```

## 优先级

对应地，要加入优先级，也很简单，只要把优先级高的运算（比如乘法）的结果作为优先级低的运算（比如加法）的元素就好了：

```
addP = chainl1 mulP $ binOp "+" $ \x y -> Op x "+" y
mulP = chainl1 oneP $ binOp "*" $ \x y -> Op x "*" y
oneP = numberP
```



上面的 Parser 可以正确解析  $1+1*1$  和  $1*1+1$  这两个执行顺序不同的表达式。

## 括号

---

这个也很简单，我就不说了，可以当作作业。

提示一下，它的作用是把优先级低的运算提高，因此它里面是需要解析一个优先级低的运算，然后这个东西本身可以变成一个最基层的运算单元。其实大家看看括号表达式在 bnf 里面的表示就一目了然了。

## 杂话 关于构建

---

根据上一篇文章在水乎的反馈，在 2017 年，Haskell 工程应该使用 Stack 而不是 Cabal 构建，而且也应该使用 Megaparsec 这个 Parser Combinator 库。不过我都不会用就是了。

## 关于 Applicative

---

而且上面的很多事情都可以用 Applicative + Alternative 做，不需要 Monad。比如我们有使用 Monad 的 Parser：

```
parseUsingMonad = do
  x <- itemP
  o <- operatorP
  y <- itemP
  return $ Op x o y
```

其实它等价于这个使用 Applicative 的 Parser：

```
parseUsingApplicative = Op <$> itemP <*> operatorP <*>
```

---

上面的代码远远简洁于 Monad 的版本。为了让 GHC 在处理 do notation 的时候尽可能转化为 Applicative 操作而不是 Monad 操作，可以使用

```
{-# LANGUAGE ApplicativeDo #-}
```

这个扩展。

## CodeWars Challenges

这里列出一些我认为很有趣的 CodeWars 的 Parser 题。我建立了一个 [collection 来收藏相关题目](#)，感兴趣的同学可以去做一下。

下面分别介绍一下各个题目。

### Operator Parser

---

此题非常新颖，给你一个结合性的 List，里面放的是以优先级排序的二元中缀符号的 Parser，然后你需要把它们组合起来成为一个大的表达式 Parser。这题一看就知道是 foldr/foldl，哈哈。

前文所使用的 OpTree 就是这道题里面的。

### Writing Applicative Parsers From Scratch

---

此题禁止你使用 Control.Applicative, Prelude 的 fmap 和 Data.Functor（我觉得它很 SB 的一点在于它没有禁止你使用 <\$>），然后让你从 0 开始写一个 Applicative 的 Parser。

## Regular Expression Parser

---

这个题目是让你解析正则表达式的，需要特别注意一些结合性的问题（这道题我的解法比较 dirty，特判了两组数据）。做这题的时候我不知道有 `chainr1`，结果写出来的代码很伤。

## Tiny Three Pass Compiler

---

老经典的题，支持很多语言，写起来贼爽。

Tweet this 

Top

创建一个 [issue](#) 以申请评论

[Create an issue](#) to apply for commentary

---

## 协议/License

本作品 使用 Haskell 编写灵活的 Parser (下) 采用 [知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议](#) 进行许可，基于 <http://ice1000.org/2017/07/27/HaskellParsers2/> 上的作品创作。

This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](#).



---

© 2017 Tesla Ice Zhang

 |  | 