从零开始手写Parser-Combinator

这不是一个教程，只是一个 Haskell 入门学习笔记。

我们的任务是编写一个用于编写 parser 的库。作为最简化的基础内容，一个 parser 应该具有把字符串转化为 AST 的能力。现在我们只有字符串，没有 AST，不妨假设我们的目标 AST 就是一个字符，这个 parser 应该像这样：

```haskell
type Parser = String → Char

parser :: Parser
parser "" = error "Empty string"
parser (x:xs) = x

parse :: Parser → String → Char
parse p = p
```

这个程序的含义应该不需要我解释。它非常简单，但却是我们的基础。在下一章我们将对它进行扩展。

在这个最简单的 parser 的基础上，我们要做一点适当的改动，使得 parser 不仅可以输出字符，还可以输出任何东西。

任何类型作为结果的 parser

所以 `Parser` 类型至少是这样的：

```haskell
type Parser a = String → a
```

上次我们用 `error` 粗暴地对待 parse 过程中的错误，现在我们把错误放到类型里，方便 parser 返回更精确的错误信息：

```haskell
data ParseError = ParseError String

type Parser a = String → Either ParseError a
```

现在我们就可以重写上一章的程序了：

```haskell
parse :: Parser a → String → Either ParseError a
parse p = p
```

```haskell
charParser :: Parser Char
charParser = \s -> case s of
  [] -> Left $ ParseError "Empty string"
  (c:cs) -> Right c
```

很好，但还不够，因为我们希望 parser 可以做更复杂一点的事情，比如说 parse 出一个特定的字符，而不是任意字符。我们希望我们的 parser 有如下的功能：如果字符串开头是 `c`，就直接返回，否则报错。也就是说，我们希望这个 parser 对字符串进行条件判断。我们需要编写一个 `satisfy` 函数，它接收一个谓词，当谓词判断输入为真时，parser 就接收输入：

```haskell
satisfy :: (Char -> Bool) -> Parser Char
satisfy p s = case s of
  [] -> Left $ ParseError "Empty string"
  (c:cs) -> if p c then Right c else Left $ ParseError $ "Expected " ++
show c

cParser :: Parser Char
cParser = satisfy (\c -> c == 'c')
```

非常好！

下一步就是处理更多的字符序列，例如在 parse 出字符 `c` 之后继续 parse 下一个字符 `o`，固然，可以这么写：

```haskell
coParser :: Parser String
coParser [] = Left $ ParseError "Empty string"
coParser (c:cs) =
  if c == 'c'
  then case cs of
    [] -> Left $ ParseError "Only one c"
    (d:ds) -> if d == 'o' then Right "co" else Left $ ParseError $
"Expected 'o' but got " ++ show d
  else Left $ ParseError $ "Expected 'c' but got " ++ show c
```

但既然我们已经有了 `satisfy`，为什么把两个 parser 组合起来呢？要组合多个 parser，一个 parser 在处理完字符串之后需要将剩余的字符串保存下来给下一个 parser 用。

可组合的 parser

现在我们修改 `Parser` 类型的定义，并添加 `regularParse` 函数使其忽略结果中剩余的字符串部分：

```haskell
type Parser a = String → (String, Either ParseError a)

regularParse :: Parser a → String → Either ParseError a
regularParse p = snd . p
```

相应的，`satisfy` 函数可以修改成如下：

```haskell
satisfy :: (Char → Bool) → Parser Char
satisfy p input = case input of
  [] → ([], Left $ ParseError "unexpected end of input")
  (x:xs) → (xs, if p x then Right x else Left $ ParseError $ "unexpected
" ++ [x])
```

现在我们就可以愉快地组合啦！

```haskell
combinate :: Parser a → (a → Parser b) → Parser b
combinate p f input = case p input of
  (input', Left err) → (input', Left err)
  (input', Right a) → case f a input' of
    (input'', Left err) → (input'', Left err)
    (input'', Right b) → (input'', Right b)

coParser :: Parser Char
coParser = combinate (satisfy (== 'c')) (\x → satisfy (== 'o'))
```

现在组合 parser 的工作就非常简单啦！只不过有个缺陷，就是组合得到的 `coParser` 忽略了前一个 parser 的结果，而直接取了后一个 parser 的结果作为最终结果。这不算什么问题，改起来也不困难，这里就不多赘述了。

观察到 `satisfy` 函数的写法和 `combinate` 的写法后，为 `Parser` 类型实现 `Monad` typeclass 的冲动就自然产生了。接下来，为了让 `Parser` 更好用，我们为它实现一系列简单的 typeclass。

ParserMonad

虚晃一枪！在写更多的代码之前，我们要做好充足的准备，那就是让我们的 parser 更加通用。首先，为 `ParseError` 类型注入更多的信息。考虑到遇见一个解析错误时，我们通常希望看到的报错信息是：我们希望解析器看到的是什么，而解析器实际看到的是什么。所以这个类型被改成这个样子：

```haskell
data ParseError = ParseError {
    expected :: String
```

```
  , met :: String
} deriving (Show)
```

对于 `Parser` 类型，我们过去用 typealias 表示，现在更正式地用 record 定义成如下形式：

```
newtype Parsec s a = Parsec {
    runParser :: [s] → ([s], Either ParseError a)
}

regularParse :: Parsec s a → [s] → Either ParseError a
regularParse p = snd . runParser p
```

注意，我们为 parser 增加了一个类型参数 `s`。这是因为一个 parser 不一定从字符串获取输入，也有可能从 token 流之类的数据结构获取输入，所以这里把输入参数类型泛化。当然，多数情况我们还是考虑作为字符串的输入，所以我们的主角是 `CharParser` 类型：

```
type CharParser = Parsec Char
```

`Functor`、`Applicative`、`Monad` 的实现都是平凡的，这里直接贴出代码：

```
instance Functor (Parsec s) where
  fmap f (Parsec p) = Parsec $ \input → case p input of
    (input, Left a) → (input, Left a)
    (input, Right b) → (input, Right (f b))

instance Applicative (Parsec s) where
  pure a = Parsec $ \input → (input, Right a)
  Parsec p <*> Parsec q = Parsec $ \input → case p input of
    (input, Left a) → (input, Left a)
    (input, Right f) → case q input of
      (input, Left a) → (input, Left a)
      (input, Right b) → (input, Right (f b))

instance Monad (Parsec s) where
  return = pure
  Parsec p >>= f = Parsec $ \input → case p input of
    (input, Left a) → (input, Left a)
    (input, Right b) → runParser (f b) input
```

手动实现一遍这些 typeclass 有助于进一步理解 parser 的结构。实现了 `Monad` 之后，组合 parser 的代码就更简洁了：

```
coParser :: CharParser Char
coParser = satisfy (== 'c') >> satisfy (== 'o')
```

更进一步的，我们把 `satisfy (== 'c')` 简化为 `char 'c'`，这个 parser 表示解析单个字符：

```
char :: Char -> CharParser Char
char c = satisfy (== c)
```

继续这个思路，我们希望有这样一个 parser 能够直接解析一个指定的字符串：

```
string :: String -> CharParser ()
string [] = return ()
string (c : cs) = char c >> string cs
```

可以用 `regularParse (string "combinator") "combinator"` 测试一下，这个 parser 确实能工作。但是，到目前为止，我们的 parser 至少有两个问题：

- `satisfy` 的报错信息很模糊，我们通过错误信息只知道遇见了什么字符，而不知道我们需要什么字符
- `string` 只能获得 Unit，而事实上我们更希望它能得到一个字符串

参考资料

https://jakewheat.github.io/intro_to_parsing

http://book.realworldhaskell.org/read/using-parsec.html

（本文定价1元）