使用 Haskell 编写灵活的 Parser（上）

2017, Jul 26 by Tesla Ice Zhang

为什么我会想到去用 Haskell 写 Parser 呢？因为 Haskell 的 do notation 对这种 Monad 组合子真的太友好了。

另一个原因是因为最近在 CodeWars 上做了不少 Parser 题，刷分巨快(因为 Parser 题都是紫色的)，很快把我推上了 1kyu 。

全文代码如有误欢迎指出，我会诚恳地改正。本文讲的内容不多，主要是介绍，下篇文章会说怎么解析带优先级/结合性的表达式。

首先我说说 Parser Combinator（就是函数式编程中采用的编写 Parser 的方式）的基本使用。 篇幅所限， Parsre Combinator 库的编写就不说了。它其实是将针对某种语法模块的 Parser 单元组合起来成为可以解析复杂语法的 Parser 。 比如我有一个可以解析数字的 Parser 和一个可以解析符号 + 的 Parser ，那么组合他们就可以得到加法运算的 Parser 。大概是这样：

```
plusParser :: Parser Char
plusParser = xxxx


numberParser :: Parser Int
```

```
numberParser = xxxx

data Op a b = Op a b a

plusOperationParser :: Parser (Op a b)
plusOperationParser = do
  a <- numberParser
  plusParser
  b <- numberParser
  return $ Op a '+' b
```

是不是一目了然呢？

语法介绍

首先我有类型及对应构造器 `Op` 表示一个二元运算，然后有解析数字的 `numberParser` 和解析加号的 `plusParser` ， 他们通过最下面那个超级简单的 do notation （就是按从上到下的顺序运行这些 Parser，解析成功则返回解析好的那个东西（比如 `Parser a` 解析后就返回 a，那么同理上面的 `plusParser` 解析完后返回一个 `Char` 。而 `numberParser` 返回的是解析好的 `Int` ， 我们通过那个箭头的语法来捕获它返回的值。后者我们需要保留解析后留下的数字，但前者我们都知道它返回的一定是 `'+'` ， 因此再获取这个返回的值意义不大，因此它就没有箭头，最后用一个 `return` 把返回值拼起来）组合到一起（事实上 do notation 只是 Monad 的系列套餐的一个语法糖，但这是一个非常厉害的语法糖。 篇幅所限，本文不讨论，也不需要读者懂 do notation 展开后到底啥样）。

总结

一个 `Parser a` 解析字符串，返回 a 类型的东西。

`Parser a` 之间可以互相组合，成为一个大 `Parser b` 。

构建最简单的 Parser

首先我们有一个函数 `satisfy` ，通过一个 `Char -> Bool` 构造出一个 `Parser Char` ，也就是给定一个函数，把字符传入， 返回 True 则代表成功解析。举个例子，这个函数接收一个 `Char` ，返回专门解析这个 `Char` 的 Parser：

```
char :: Parser Char
char = satisfy . (==)
```

如果不 `point-free` 并进行 eta 展开的话就是这样（如果看不懂上面的版本就看这个吧）：

```
char c = satisfy (\x -> x == c)
```

而如果我们要编写一个解析字符串 `>>=` 的 Parser，则：

```
bind :: Parser String
bind = do
  char '>'
  char '>'
  char '='
  return ">>="
```

再抽象一下，解析某个长度为 3 的字符串（作为参数）的 Parser：

```
three :: String -> Parser String
three s@[a, b, c] = do
  char a
  char b
  char c
  return s
```

再抽象一下，解析任意长度为 3 的字符串:

```
oneC :: Parser Char
oneC = do
  c <- satisfy $ const True
  return c
--

three' :: Parser String
three' = do
  a <- oneC
  b <- oneC
  c <- oneC
  return [a, b, c]
```

到这里你是不是已经大概清楚怎么使用 Parser Combinator 了？想不想动手试试？

（有毒）具体在 GHCi 中使用

首先，大家可以选择使用我在文章末尾提供的微型实现;

或者安装:

- 版本算正常的 GHC
- 版本算正常的 Cabal

然后执行：

```
$ cabal install parsec
...

$ ghci
...

Prelude> :m +Text.ParserCombinators.ReadP
```

然后剩下的我也不会，我只知道我的 Parser 实现在这个库中的等价物是~~（应该是）~~ `ReadP a`。 所以你们还是直接用我在附录里面放的~~网上找的~~那份独立的吧（雾（其实我还魔改过一点点啦，编码习惯啥的，不是大事））。

用法：

首先写好你的 Parser（假设就是 `myP` ，直接和我给的代码写一起），然后打开 GHCi 加载你的文件，然后运行：

```
parseCode myP "the code you want to parse"
```

就可以看到返回的结果了。具体的例子：

```
Main> parseCode (satisfy (`elem` "jfly")) "j"
'j'
```

```
Main> parseCode (satisfy (`elem` "jfly")) "f"
'f'
Main> parseCode (satisfy (`elem` "jfly")) "g"
*** Exception: Hugh?
```

读者感兴趣的话可以试试实现一个 Parser，解析一个带括号的字符串，返回括号内部的东西。

你学到了什么

- 简易的 Parser Combinator 的使用

预告

- 稍微复杂点的表达式的解析

（附）Parser Combinator 简易实现

成功解析返回解析结果，失败则 `error "Hugh?"`（模仿 dram）。

```
import Data.Char
import Data.List
import Control.Monad
import Control.Applicative

-----------------------------------------------------
--------------- my parser combinator ----------------
-----------------------------------------------------
```

```
newtype Parser val = Parser { parse :: String -> [(val,

parseCode :: Parser a -> String -> a
parseCode m s = case parse m s of
  [(res, [])] -> res
  _           -> error "Hugh?"
--

instance Functor Parser where
  fmap f (Parser ps) = Parser $ \p -> [ (f a, b) | (a,
--

instance Applicative Parser where
  pure = return
  (Parser p1) <*> (Parser p2) = Parser $ \p ->
    [ (f a, s2) | (f, s1) <- p1 p, (a, s2) <- p2 s1 ]
--

instance Monad Parser where
  return a = Parser $ \s -> [(a, s)]
  p >>= f  = Parser $ concatMap (\(a, s1) -> f a `parse
--

instance MonadPlus Parser where
  mzero     = Parser $ const []
  mplus p q = Parser $ \s -> parse p s ++ parse q s
```

```
--

instance Alternative Parser where
  empty   = mzero
  p <|> q = Parser $ \s -> case parse p s of
    [] -> parse q s
    rs -> rs
--

item :: Parser Char
item = Parser $ \s -> case s of
  [     ] -> []
  (h : t) -> [(h, t)]
--

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = item >>= \c -> if p c then return c else em
```

Tweet this

Top

创建一个 issue 以申请评论

Create an issue to apply for commentary

协议/License