Monadic Parser Combinators 介绍

陈逸凡（Neuromancer ）

19th Mar, 2017

1 Intro

提示 本篇假设读者对基本的 Haskell 语法有所了解，主要包括：类型匹配，参数化类型，lambda 表达式，类型构造器，类型类等。不需要读者熟悉 Monad 类型。

1.1 何为 Combinator

在 Parser 构造过程中，将目标 Parser 分解为多个小的 Parser 再组合起来是一种常用的构造方式。将大的问题分解为多个小目标，再把对这些小目标构造出的各种简单 Parser 像乐高积木一样组合，便能最终得到一个复杂而精美的 Parser，这便是“分解——组合”思想的精妙之处。

Combinators 将 Parser 的“组合”抽象成一个 Domain Specific Language (DSL) 。用统一的语言来表示这样的组合过程，并提供了一些最基本的 Parser ，这就使得构造 Parser 变成了一个非常简单而易读（readable）的过程。而且各种 Parser 都可复用同样的函数，也很符合代码复用的目标。

1.2 何为 Monadic

Monad 是函数式编程中常用的一类抽象数据类型/代数概念，这里暂且不提其抽象性质，只谈一谈其在 Parsing 过程中的用途。

假设已经有了一个简单的 Parser 记为 **p**，对于给定输入（这里类型不妨为 String ，后同），**p** 要么将其解析出一个类型为 a 的结果（和剩余的未解

析的输入），要么给出一个错误信息。我们可以将 **p** 的类型其表示为以下用 Haskell 代码表示的形式。

```
data Parser a =
  P { runParser : Input -> ParseResult a }
data ParseResult a = Err ParseErr | Res Input a
```

使用 monad （或者在 Haskell 语境下，将 Parser 定义为 Monad 类型类实例），可以达到这样的效果：

1. 如果成功解析，可以把解析出的数据传递下去并被后续的解析过程利用，利用高阶函数，可以充分重用解析结果类型 a 可用的各类函数而不必一一手动提升
2. 如果解析失败，可以把错误信息传递下去而并不需要特殊处理，并在最后用统一的错误处理过程来解决出错信息，这样相比直接 error 跳出或显式地传递错误信息对重构代码要友好得多
3. 还可以为解析结果附加更多的有用信息，具体做法可以是给 Parser a 类型再套一个 Logger 类型类。值得一提的是，这么做的时候，只需要修改 Parser 的定义和添加一个统一的 Logger 信息处理过程，而几乎不必修改其他部分代码。不过这里还是用当前不带 Logger 的 Parser 类型，目的是只关注本篇主题。

Monad 到底能带来多大的益处，可以通过后面的讲解来详细展示。

2 基本组合子的构造

就像乐高积木一样，Parser Combinator 也会提供一些最简单的砖块以供使用者来构造更复杂的结构。

2.1 固定读取一个字符的 Parser

就跟字面意思一样，这个 Parser 固定从输入流头部取出一个字符作为结果；当输入为空时，就返回错误信息。

```
anyChar :: Parser Char
anyChar = P $ \inp ->
  case inp of
    [] -> Err UnexpectedEof
    x:xs -> Res xs x
```

2.2 永远返回错误信息的 Parser

由于错误信息有多种，所以需要构造多个对应的 Parser，这里举两个例子，有需要的时候可以自己添加更多错误信息到 ParseError 类型中，并构造出对应的组合子。

```
unexpectedChar :: Char -> Parser a
unexpectedChar c = P $ \_ -> Err (UnexpectedChar c)

failed :: Parser a
failed = P $ \_ -> Err Failed
```

2.3 不解析但返回一个值

这个 Parser 实际上并不解析任何东西，构造这个组合子的目的在于将把基本类型的计算过程引入到 Parser 里。具体用法后面会再出现。

```
pureParser :: a -> Parser a
pureParser x = P $ \inp -> Res inp x
```

这就是全部所需的基本 Parser 。就如同所看到的那样，非常简洁。可以类比为这就是一个代数系统的生成元，配合给出的运算子，就可以组合出全部可能的 Parser 。

3 定义基本运算

以上组合子，实际发生解析的只有第一个 anyChar 解析器，要怎样从找一个简陋至极的解析器出发构造更多的解析器呢？这就需要一些运算来调

整 anyChar 了。

3.1 将基本函数映射到 Parser 上

需要一个“提升”操作，使得原本作用在 a 类型上的函数应用到 Parser a 上。

```
mapParser :: (a -> b) -> Parser a -> Parser b
mapParser f p = P $ \inp ->
  case runParser p inp of
    Res inp' x -> Res inp' (f x)
    Err err -> Err err
```

这里，我们用一个基本函数和一个 Parser 构造出一个产生不同结果的 Parser 。如果原 Parser 会解析错误，那么新 Parser 也返回同样的解析错误；如果原 Parser 解析出一个结果，那么新 Parser 就会给出函数作用在结果后的返回值。

3.2 将解析结果提供给之后的解析过程

如果我们需要把解析出来的结果提供给后续的解析过程使用，比如构造带有复杂数据结构的结果，又或者是上下文敏感文法（context sensitive）需要，就需要一个操作来“取出”解析结果。

```
bindParser ::
  (a -> Parser b) -> Parser a -> Parser b
bindParser bf pa = P $ \inp ->
  case runParser pa inp of
    Res inp' x -> runParser (bf x) inp'
    Err err -> Err err
```

这个函数的含义是，给定一个由一个参数决定的 Parser ，和一个 Parser ，将后者的解析结果作为参数提供给前者，得到一个新的 Parser ，——这与之前的“取出结果提供给后续过程”是等价的。同之前的 mapParser 一样，

如果第一个 Parser 就解析错误，那后续过程也不会继续，而是直接返回解析错误。

提示 为方便书写，后面使用交换两个参数位置的 flbindParser 。

```
flbindParser ::
  Parser a -> (a -> Parser b) -> Parser b
flbindParser = flip bindParser
```

3.3 定义为类型类实例

事实上，已经定义的这些运算已经足以定义 Parser 为类型类实例。这样做的目的在于，一来只需要定义基本函数就可以充分利用一系列 Haskell 已定义的衍生函数；二来可以用统一的 API 来提供给一个，比如说，对 Monad 已经有所了解的使用者，让 TA 几乎不看文档就可以直接利用 Monad 类型类的操作符来操作 Parser 。

```
instance Functor Parser where
  fmap :: (a -> b) -> Parser a -> Parser b
  fmap = mapParser

instance Applicative Parser where
  pure :: a -> Parser a
  pure = pureParser
  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
  pf <*> pa = flbindParser pf $ \f ->
              flbindParser pa $ \x ->
              pureParser $ f x

instance Monad Parser where
  (=<<) :: (a -> Parser b) -> Parser a -> Parser b
  (=<<) = bindParser
```

感兴趣的读者可以自行验证在这里 Functor law ，Applicative law 和 Monad law 都得到了满足。这也是我们标题中第一个单词的来源。由类

型类导出的函数会在后面使用时再做出对应解释。

3.4 串联解析器

把两个解析器串联起来，前一个解析完了让第二个继续解析。这样的操作都已经由 Applicative 类型类提供了对应的操作符，分别是 <*> 将前一个解析结果应用到后一个解析结果上作为返回值，*> 只返回后一个 Parser 的解析结果，<* 只返回前一个 Parser 的解析结果。

3.5 并联解析器

如果第一个 Parser 解析失败，那么就尝试用第二个 Parser 解析。

```
(<|>) :: Parser a -> Parser a -> Parser a
p <|> q = P $ \inp ->
case runParser p inp of
Err err -> runParser q inp
res -> res
```

有了并联操作之后，最有趣的一点在于我们可以借此实现解析数量不定的元素的 Parser，实现正则表达式中符号 * / +，或 EBNF 表达式中符号的功能。

```
manyP :: Parser a -> Parser [a]
manyP pa = someP pa <|> pure []

someP :: Parser a -> Parser [a]
someP pa = pa >>= \x ->
manyP pa >>= \xs ->
pure $ x:xs
```

这是一个间接递归调用的例子，many 返回包含 0 或多个元素的列表，some 返回包含至少一个元素的列表。

在定义了并联操作和 some ，many 操作后，实际上我们已经使得 Parser 满足了一个称为 MonadPlus 的类型类 / 代数结构的要求——也就是目前所看到的这些要求。

4 更多操作

通过之前定义的这些基本操作，我们可以定义出更多更复杂的操作。

4.1 只接受限定字符

对 anyChar 解析出的字符进行筛选。

```
satisfyP :: (Char -> Bool) -> Parser Char
satisfyP p = anyChar >>= \c ->
  if p c
    then pure c
    else unexpectedChar c
```

注 '»=' 操作符就是 flbindParser 的中缀版本。

类似的，可以把这里的 Char 类型替换为更一般的类型变量，不过就不能使用 unexpectedChar 作为解析错误信息了。

通过替换这里的判定函数 p ，也可以构造许多特定的筛选函数。如：

```
isP :: Char -> Parser Char
isP c = satisfyP (== c)

digitP :: Parser Char
digitP :: satisfy isDigit
```

4.2 按顺序调用一列解析器

把同一类型的 Parser 放在一个列表中，按序调用后返回一个解析结果列表，这样可以简化不少重复劳动。

```
seqParser :: [Parser a] -> Parser [a]
seqParser [] = pure []
seqParser (p:ps) = p >>= \x ->
                   seqParser ps >>= \xs ->
                   pure $ x:xs
```

这样，我们可以直接由一个字符串生成一个解析指定字符串的 Parser：

```
stringP :: String -> Parser String
stringP s = seqParser $ map isP s
```

4.3 解析被特定符号包裹的数据

这样的 Parser 的应用场景可以是比如读取一个 HTML tag 。

```
betweenP ::
  Parser b -> Parser c -> Parser a -> Parser a
betweenP pl pr pm = pl *> pm <* pr
```

上面这段代码含义就是左中右顺序解析，并忽略掉两侧解析特殊符号的结果，只返回中间的解析结果。

4.4 解析被特定符号分割的数据

使用在解析比如一列逗号分割的字符串中，如 CSV 格式。

```
sepByP :: Parser b -> Parser a -> Parser [a]
sepByP pb pa = ( pa >>= h ->
                 many (pb *> pa) >>= t ->
                 pure $ h : t ) <|> pure []
```

上面这段代码，首先读取 1 个元素，再识别 0 个或多个前面包含分隔符的元素，并把所有解析出元素串起来。如果一个元素都没有识别到，就返回空列表。

4.5 更多更复杂的 Parser

可以看到，在定义基本操作之后，定义其他 Parser 就不再涉及具体的 Parser 类型类的定义。作为 Parser Combinator 的使用者，并不需要知道 Parser a 的具体结构，只需要知道 Parser Combinator 提供的基本组合子和操作符，就可以搭建出一个解析出特定结构的 Parser ，这也就是为什么这些 Combinators 可以被称之为 Embedded Domain Specific *Language* 而不仅仅是一个 Library 。

5 一个 JSON Parser 的例子

下面，就用以 JSON 格式为例，看我们如何方便地构造将字符串解析成用 Haskell 抽象数据结构表示的 JSON 类型的。[1]

```
jsonString :: Parser Chars
jsonString =
  between
    (is $ fromSpecialCharacter DoubleQuote)
    (charTok $ fromSpecialCharacter DoubleQuote)
    (list (spacialHex <|> noneof (listh "\\\"")))
  where
    spacialHex = is '\\' *> (hexu ||| special)
    special =
      do
        c <- character
        case toSpecialCharacter c of
            Full sc -> valueParser
              $ fromSpecialCharacter sc
            Empty -> unexpectedCharParser c

jsonNumber :: Parser Rational
jsonNumber = P $ \inp ->
  case readFloats inp of
      Full (num, inp') -> Result inp' num
```

[1]基于 data61/fp-course 的代码。https://github.com/data61/fp-course

```
      Empty -> ErrorResult $
                case inp of
                  Nil -> UnexpectedEof
                  c :. _ -> UnexpectedChar c

jsonTrue :: Parser Chars
jsonTrue = stringTok "true"

jsonFalse :: Parser Chars
jsonFalse = stringTok "false"

jsonNull :: Parser Chars
jsonNull = stringTok "null"

jsonArray :: Parser (List JsonValue)
jsonArray = betweenSepbyComma '[' ']' jsonValue

jsonObject :: Parser Assoc
jsonObject = betweenSepbyComma '{' '}' singleObject
  where
    singleObject =
      do
        spaces
        s <- jsonString
        spaces
        is ':'
        v <- jsonValue
        spaces
        return (s, v)

jsonValue :: Parser JsonValue
jsonValue = spaces *> (
  (pure JsonNull <* jsonNull) <|>
```

```
  (pure JsonTrue <* jsonTrue) <|>
  (pure JsonFalse <* jsonFalse) <|>
  (JsonArray <$> jsonArray) <|>
  (JsonString <$> jsonString) <|>
  (JsonObject <$> jsonObject) <|>
  (JsonRational False <$> jsonNumber)
  ) <* spaces

readJsonValue :: Filename -> IO (ParseResult JsonValue)
readJsonValue filename =
  do
    inp <- readFile filename
    return (parse jsonValue inp)
```

这个 Parser 的构造仅需要数十行，其中甚至有超过三分之一是对特殊字符 escape 的处理（注，这里略去了一些小的操作函数的定义，但从其名字中也很容易看出来是怎么用组合子构造的。此外还略去了对特殊字符的转换函数）。相比之下，一个用 C 语言手写的递归下降解释器就比这复杂得多了，可读性和代码复用性也要差上不少。

值得一提的是这里对 JSON Number Parser 的构造。我们并没有用已有的组合子，而是自己重新构造了一个 Parser ，因为可以直接利用已有的 read 函数。于是这又体现出 *Embedded* DSL 的另一个特性，即可以与宿主语言无缝结合。

6 更复杂的问题

之前的 JSON Parser 是非常简单的，因为这是一个 LL(1) 文法，简单的递归下降解释器就能完成解析。对于更复杂的问题，Parser Combinators 同样可以解决。

6.1 上下文相关性

如前所述，利用 Monad 的 bind 功能可以将 Parser 的解析结果取出提供给后续的解析过程，这也就解决了上下文相关的问题。当然，这也是递归下降解释器都能做到的事情

6.2 任意向前看文法

之前定义的 <|> 函数实际上已经隐含了无限向前看的能力。第一个 Parser 可以尝试解析前方任意长字符串，失败了就交给第二个 Parser 重新。

6.3 左递归处理

实际上，Parser Combinator 如何处理左递归 left-recursion 直到 2008 年才解决，在 Parser Combinator 概念第一次提出的 19 年后。[2]

对于左递归的处理，可以参考 parsec 库[3]的做法，这里直接引用其中的代码：

```
chainl1 :: (Stream s m t) => ParsecT s u m a
  -> ParsecT s u m (a -> a -> a)
  -> ParsecT s u m a
chainl1 p op = do{ x <- p; rest x }
             where
               rest x    = do{ f <- op
                             ; y <- p
                             ; rest (f x y)
                             }
                         <|> return x

chainr1 :: (Stream s m t) => ParsecT s u m a
  -> ParsecT s u m (a -> a -> a)
```

[2]Frost, Richard A.; Hafiz, Rahmatullah; Callaghan, Paul (2008). "Parser Combinators for Ambiguous Left-Recursive Grammars".

[3]https://github.com/aslatter/parsec

```
    -> ParsecT s u m a
  chainr1 p op = scan
                 where
                   scan = do{ x <- p; rest x }
                   rest x     = do{ f <- op
                                  ; y <- scan
                                  ; return (f x y)
                                  }
                                <|> return x
```

6.4 二义性文法

对于二义性，最直接的方法就是将所有可能结果存放在一个列表中，将 Parser 定义改为 P (Input -> [ParseResult]) （这样的改动并不会带来大范围重构），用一个类似于 <|> 操作符的函数将所有可能结果合并起来。但这样的代价就是可能会导致复杂度急剧上升，用 Memoization 可以使得复杂度降至多项式时间[4]。

7 总结

Parser Combinators 是一项非常简洁的自顶向下 Top-Down 语法分析器构造技术，利用这项技术可以极大减小编写语法分析器时的痛苦。利用一些高效的基础数据结构，可以使得 Parser Combinators 有着可与自底向上解析器一比的效率。

参考文献

[1] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In In ????, pages, 2007.

[4] Frost, Richard A.; Szydlowski, Barbara (1996). "Memoizing Purely Functional Top-Down Backtracking Language Processors"

[2] Richard A Frost and Barbara Szydlowski. Memoizing purely functional top-down backtracking language processors. Science of Computer Programming, 27(3):263 – 288, 1996.

[3] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[4] Daan Leijen. Parsec, a fast combinator parser, 2001.

[5] Tony Morris. Json parser code from fp-course.