Если вы видите что-то необычное, просто сообщите мне.

Applicative Parsing

In part 1 of this series, we prepared ourselves for parsing by going over the basics of the Gherkin Syntax. In this part, we'll be using Regular Expression (Regex) based, applicative parsing to parse the syntax. We'll start by focusing on the fundamentals of this library and building up a vocabulary of combinators to use. We'll make heavy use of the Applicative typeclass. If you need a refresher on that, check out this article.

As we start coding, you can also follow along with the examples on Github here! Most of the code here is in the RegexParser module.

If you're itching to try a monadic approach to parsing, be sure to check out part 3 of this series, where we'll learn about the Attoparsec library. If you want to learn about a wider variety of production utilities, download our Production Checklist. It summarizes many other useful libraries for writing higher level Haskell.

GETTING STARTED

So to start parsing, let's make some notes about our input format. First, we'll treat our input feature document as a single string. We'll remove all empty lines, and then trim leading and trailing whitespace from each line.

```
parseFeatureFromFile :: FilePath -> IO Feature
parseFeatureFromFile inputFile = do
  fileContents <- lines <$> readFile inputFile
  let nonEmptyLines = filter (not . isEmpty) fileContents
  let trimmedLines = map trim nonEmptyLines
  let finalString = unlines trimmedLines
  case parseFeature finalString of
    ...
```

```
...

isEmpty :: String -> Bool
isEmpty = all isSpace

trim :: String -> String
trim input = reverse flippedTrimmed
  where
    trimStart = dropWhile isSpace input
    flipped = reverse trimStart
    flippedTrimmed = dropWhile isSpace flipped
```

This means a few things for our syntax. First, we don't care about indentation. Second, we ignore extra lines. This means our parsers might allow certain formats we don't want. But that's OK because we're trying to keep things simple.

THE RE TYPE

With applicative based parsing, the main data type we'll be working with is called RE, for regular expression. This represents a parser, and it's parameterized by two types:

```
data RE s a = ...
```

The s type refers to the fundamental unit we'll be parsing. Since we're parsing our input as a single String, this will be Char. Then the a type is the result of the parsing element. This varies from parser to parser. The most basic combinator we can use is sym. This parses a single symbol of your choosing:

```
sym :: s - > RE s s

parseLowercaseA :: RE Char Char
parseLowercaseA = sym 'a'
```

To use an RE parser, we call the match function or its infix equivalent =~. These will return a Just value if we can match the entire input string, and Nothing otherwise:

```
>> match parseLowercaseA "a"
Just 'a'
>> "b" =~ parseLowercaseA
Nothing
>> "ab" =~ parseLowercaseA
Nothing -- (Needs to parse entire input)
```

PREDICATES AND STRINGS

Naturally, we'll want some more complicated functionality. Instead of parsing a single input character, we can parse any character that fits a particular predicate by using psym. So if we want to read any character that was not a newline, we could do:

```
parseNonNewline :: RE Char Char
parseNonNewline = psym (/= '\n')
```

The string combinator allows us to match a particular full string and then return it:

```
readFeatureWord :: RE Char String
readFeatureWord = string "Feature"
```

We'll use this for parsing keywords, though we'll often end up discarding the "result".

APPLICATIVE COMBINATORS

Now the RE type is applicative. This means we can apply all kinds of applicative combinators over it. One of these is many, which allows us to apply a single parser several times. Here is one combinator that we'll use a lot. It allows us to read everything up until a newline and return the resulting string:

```
readUntilEndOfLine :: RE Char String
readUntilEndOfLine = many (psym (/= '\n'))
```

Beyond this, we'll want to make use of the applicative <*> operator to combine different parsers. We can also apply a pure function (or constructor) on top of those by using <$>. Suppose we have a data type that stores two characters. Here's how we can build a parser for it:

```
data TwoChars = TwoChars Char Char

parseTwoChars :: RE Char TwoChars
parseTwoChars = TwoChars <$> parseNonNewline <*> parseNonNewline

...

>> match parseTwoChars "ab"
Just (TwoChars 'a' 'b')
```

We can also use <* and *>, which are cousins of the main applicative operator. The first one will parse but then ignore the right hand parse result. The second discards the left side result.

```
parseFirst :: RE Char Char
parseFirst = parseNonNewline <* parseNonNewline

parseSecond :: RE Char Char
parseSecond = parseNonNewline *> parseNonnewline
>> match parseFirst "ab"
Just 'a'
>> match parseSecond "ab"
Just 'b'
>> match parseFirst "a"
Nothing
```

Notice the last one fails because the parser needs to have both inputs! We'll come back to this idea of failure in a second. But now that we know this technique, we can write a couple other useful parsers:

```
readThroughEndOfLine :: RE Char String
readThroughEndOfLine = readUntilEndOfLine <* sym '\n'

readThroughBar :: RE Char String
readThroughBar = readUntilBar <* sym '|'
```

```
readUntilBar :: RE Char String
readUntilBar = many (psym (\c -> c /= '|' && c /= '\n'))
```

The first will parse the rest of the line and then consume the newline character itself. The other parsers accomplish this same task, except with the vertical bar character. We'll need these when we parse the Examples section further down.

ALTERNATIVES: DEALING WITH PARSE FAILURE

We introduced the notion of a parser "failing" up above. Of course, we need to be able to offer alternatives when a parser fails! Otherwise our language will be very limited in its structure. Luckily, the RE type also implements Alternative. This means we can use the <|> operator to determine an alternative parser when one fails. Let's see this in action:

```
parseFeatureTitle :: RE Char String
parseFeatureTitle = string "Feature: " *> readThroughEndOfLine

parseScenarioTitle :: RE Char String
parseScenarioTitle = string "Scenario: " *> readThroughEndOfLine

parseEither :: RE Char String
parseEither = parseFeatureTitle <|> parseScenarioTitle
>> match parseFeatureTitle "Feature: Login\n"
Just "Login"
>> match parseFeatureTitle "Scenario: Login\n"
Nothing
>> match parseEither "Scenario: Login\n"
Just "Login"
```

Of course, if ALL the options fail, then we'll still have a failing parser!

```
>> match parseEither "Random: Login\n"
Nothing
```

We'll need this to introduce some level of choice into our parsing system. For instance, it's up to the user if they want to include a Background as part of their feature. So we need to be able to read the background if it's there or else move onto parsing a scenario.

VALUE PARSER

In keeping with our approach from the last article, we're going to start with smaller elements of our syntax. Then we can use these to build larger ones with ease. To that end, let's build a parser for our Value type, the most basic data structure in our syntax. Let's recall what that looks like:

```
data Value =
  ValueNull |
  ValueBool Bool |
  ValueString String |
  ValueNumber Scientific
```

Since we have different constructors, we'll make a parser for each one. Then we can combine them with alternative syntax:

```
valueParser :: RE Char Value
valueParser =
  nullParser <|>
  boolParser <|>
  numberParser <|>
  stringParser
```

Now our parsers for the null values and boolean values are easy. For each of them, we'll give a few different options about what strings we can use to represent those elements. Then, as with the larger parser, we'll combine them with <|>.

```
nullParser :: RE Char Value
nullParser =
  (string "null" <|>
  string "NULL" <|>
  string "Null") *> pure ValueNull
```

```haskell
boolParser :: RE Char Value
boolParser =
  trueParser *> pure (ValueBool True) <|>
  falseParser *> pure (ValueBool False)
  where
    trueParser = string "True" <|> string "true" <|> string "TRUE"
    falseParser = string "False" <|> string "false" <|> string "FALSE"
```

Notice in both these cases we discard the actual string with *> and then return our constructor. We have to wrap the desired result with pure.

NUMBER AND STRING VALUES

Numbers and strings are a little more complicated since we can't rely on hard-coded formats. In the case of numbers, we'll account for integers, decimals, and negative numbers. We'll ignore scientific notation for now. An integer is simple to parse, since we'll have many characters that are all numbers. We use some instead of many to enforce that there is at least one:

```haskell
numberParser :: RE Char Value
numberPaser = ...
  where
    integerParser = some (psym isNumber)
```

A decimal parser will read some numbers, then a decimal point, and then more numbers. We'll insist there is at least one number after the decimal point.

```haskell
numberParser :: RE Char Value
numberPaser = ...
  where
    integerParser = some (psym isNumber)
    decimalParser =
      many (psym isNumber) <*> sym '.' <*> some (psym isNumber)
```

Finally, for negative numbers, we'll read a negative symbol and then one of the other parsers:

```haskell
numberParser :: RE Char Value
numberPaser = ...
  where
    integerParser = some (psym isNumber)
    decimalParser =
```

```
      many (psym isNumber) <*> sym '.' <*> some (psym isNumber)
    negativeParser = sym '-' <*> (decimalParser <|> integerParser)
```

However, we can't combine these parsers as is! Right now, they all return different results! The integer parser returns a single string. The decimal parser returns two strings and the decimal character, and so on. In general, we'll want to combine each parser's results into a single string and then pass them to the read function. This requires mapping a couple functions over our last two parsers:

```
numberParser :: RE Char Value
numberPaser = ...
  where
    integerParser = some (psym isNumber)
    decimalParser = combineDecimal <$>
      many (psym isNumber) <*> sym '.' <*> some (psym isNumber)
    negativeParser = (:) <$>
      sym '-' <*> (decimalParser <|> integerParser)

    combineDecimal :: String -> Char -> String -> String
    combineDecimal base point decimal = base ++ (point : decimal)
```

Now all our number parsers return strings, so we can safely combine them. We'll map the ValueNumber constructor over the value we read from the string.

```
numberParser :: RE Char Value
numberPaser = (ValueNumber . read) <$>
  (negativeParser <|> decimalParser <|> integerParser)
  where
    ...
```

Note that order matters! If we put the integer parser first, we'll be in trouble! If we encounter a decimal, the integer parser will greedily succeed and parse everything before the decimal point. We'll either lose all the information after the decimal, or worse, have a parse failure.

The last thing we need to do is read a string. We need to read everything in the example cell until we hit a vertical bar, but then ignore any whitespace. Luckily, we have the right combinator for this, and we've even written a trim function already!

```
stringParser :: RE Char Value
stringParser = (ValueString . trim) <$> readUntilBar
```

And now our valueParser will work as expected!

BUILDING AN EXAMPLE TABLE

Now that we can parse individual values, let's figure out how to parse the full example table. We can use our individual value parser to parse a whole line of values! The first step is to read the vertical bar at the start of the line.

```
exampleLineParser :: RE Char [Value]
exampleLineParser = sym '|' *> ...
```

Next, we'll build a parser for each cell. It will read the whitespace, then the value, and then read up through the next bar.

```
exampleLineParser :: RE Char [Value]
exampleLineParser = sym '|' *> ...
  where
    cellParser =
      many isNonNewlineSpace *> valueParser <* readThroughBar

isNonNewlineSpace :: RE Char Char
isNonNewlineSpace = psym (\c -> isSpace c && c /= '\n')
```

Now we read many of these and finish by reading the newline:

```
exampleLineParser :: RE Char [Value]
exampleLineParser =
  sym '|' *> many cellParser <* readThroughEndOfLine
  where
    cellParser =
      many isNonNewlineSpace *> valueParser <* readThroughBar
```

Now, we need a similar parser that reads the title column of our examples. This will have the same structure as the value cells, only it will read normal alphabetic strings instead of values.

```
exampleColumnTitleLineParser :: RE Char [String]
exampleColumnTitleLineParser = sym '|' *> many cellParser <* readThroughEndOfLine
  where
    cellParser =
      many isNonNewlineSpace *> many (psym isAlpha) <* readThroughBar
```

Now we can start building the full example parser. We'll want to read the string, the column titles, and then the value lines.

```
exampleTableParser :: RE Char ExampleTable
exampleTableParser =
  (string "Examples:" *> readThroughEndOfLine) *>
  exampleColumnTitleLineParser <*>
  many exampleLineParser
``
We're not quite done yet. We'll need to apply a function over these results that will produce the final ExampleTable. And the trick is that we want to map up the example keys with their values. We can accomplish this with a simple function. It will return zip the keys over each value list using map:
```haskell
exampleTableParser :: RE Char ExampleTable
exampleTableParser = buildExampleTable <$>
  (string "Examples:" *> readThroughEndOfLine) *>
  exampleColumnTitleLineParser <*>
  many exampleLineParser
  where
    buildExampleTable :: [String] -> [[Value]] -> ExampleTable
    buildExampleTable keys valueLists = ExampleTable keys (map (zip keys) valueLists)
```

STATEMENTS

Now we that we can parse the examples for a given scenario, we need to parse the Gherkin statements. To start with, let's make a generic parser that takes the keyword as an argument. Then our full parser will try each of the different statement keywords:
```

```
parseStatementLine :: String -> RE Char Statement
parseStatementLine signal = ...

parseStatement :: RE Char Statement
parseStatement =
  parseStatementLine "Given" <|>
  parseStatementLine "When" <|>
  parseStatementLine "Then" <|>
  parseStatementLine "And"
```

Now we'll get the signal word out of the way and parse the statement line itself.

```
parseStatementLine :: String -> RE Char Statement
parseStatementLine signal = string signal *> sym ' ' *> ...
```

Parsing the statement is tricky. We want to parse the keys inside brackets and separate them as keys. But we also want them as part of the statement's string. To that end, we'll make two helper parsers. First, nonBrackets will parse everything in a string up through a bracket (or a newline).

```
nonBrackets :: RE Char String
nonBrackets = many (psym (\c -> c /= '\n' && c /= '<'))
```

We'll also want a parser that parses the brackets and returns the keyword inside:

```
insideBrackets :: RE Char String
insideBrackets = sym '<' *> many (psym (/= '>')) <* sym '>'
```

Now to read a statement, we start with non-brackets, and alternate with keys in brackets. Let's observe that we start and end with non-brackets, since they can be empty. Thus we can represent a line a list of non-bracket/bracket pairs, followed by a last non-bracket part. To make a pair, we combine the parser results in a tuple using the (,) constructor enabled by TupleSections:

```
parseStatementLine :: String -> RE Char Statement
parseStatementLine signal = string signal *> sym ' ' *>
  many ((,) <$> nonBrackets <*> insideBrackets) <*> nonBrackets
```

From here, we need a recursive function that will build up our final statement string and the list of keys. We do this with buildStatement.

```
parseStatementLine :: String -> RE Char Statement
parseStatementLine signal = string signal *> sym ' ' *>
  (buildStatement <$>
    many ((,) <$> nonBrackets <*> insideBrackets) <*> nonBrackets)
  where
    buildStatement ::
      [(String, String)] -> String -> (String, [String])
    buildStatement [] last = (last, [])
    buildStatement ((str, key) : rest) rem =
      let (str', keys) = buildStatement rest rem
      in (str <> "<" <> key <> ">" <> str', key : keys)
```

The last thing we need is a final helper that will take the result of buildStatement and turn it into a Statement. We'll call this finalizeStatement, and then we're done!

```
parseStatementLine :: String -> RE Char Statement
parseStatementLine signal = string signal *> sym ' ' *>
  (finalizeStatement . buildStatement <$>
    many ((,) <$> nonBrackets <*> insideBrackets) <*> nonBrackets)
  where
    buildStatement ::
      [(String, String)] -> String -> (String, [String])
    buildStatement [] last = (last, [])
    buildStatement ((str, key) : rest) rem =
      let (str', keys) = buildStatement rest rem
      in (str <> "<" <> key <> ">" <> str', key : keys)

    finalizeStatement :: (String, [String]) -> Statement
    finalizeStatement (regex, variables) = Statement regex variables
```

SCENARIOS

Now that we have all our pieces in place, it's quite easy to write the parser for scenario! First we get the title by reading the keyword and then the rest of the line:

```
scenarioParser :: RE Char Scenario
scenarioParser = string "Scenario: " *> readThroughEndOfLine ...
```

After that, we read many statements, and then the example table. Since the example table might not exist, we'll provide an alternative that is a pure, empty table. We can wrap everything together by mapping the Scenario constructor over it.

```
scenarioParser :: RE Char Scenario
scenarioParser = Scenario <$>
  (string "Scenario: " *> readThroughEndOfLine) <*>
  many (statementParser <* sym '\n') <*>
  (exampleTableParser <|> pure (ExampleTable [] []))
```

We can also make a "Background" parser that is very similar. All that changes is that we read the string "Background" instead of a title. Since we'll hard-code the title as "Background", we can include it with the constructor and map it over the parser.

```
backgroundParser :: RE Char Scenario
backgroundParser = Scenario "Background" <$>
  (string "Background:" *> readThroughEndOfLine) *>
 many (statementParser <* sym '\n') <*>
  (exampleTableParser <|> pure (ExampleTable [] []))
```

FINALLY THE FEATURE

We're almost done! All we have left is to write the featureParser itself! As with scenarios, we'll start with the keyword and a title line:

```
featureParser :: RE Char Feature
featureParser = Feature <$>
  (string "Feature: " *> readThroughEndOfLine) <*>
  ...
```

Now we'll use the optional combinator to parse the Background if it exists, but return Nothing if it doesn't. Then we'll wrap up with parsing many scenarios!

```
featureParser :: RE Char Feature
featureParser = Feature <$>
  (string "Feature: " *> readThroughEndOfLine) <*>
```

```
  pure [] <*>
  (optional backgroundParser) <*>
  (many scenarioParser)
```

Note that here we're ignoring the "description" of a feature we proposed as part of our original syntax and simply giving an empty list of strings. Since there are no keywords for that, it turns out to be painful to deal with it using applicative parsing. When we look at monadic approaches starting next week, we'll see it isn't as hard there.

CONCLUSION

This wraps up our exploration of applicative parsing. We can see how well suited Haskell is for parsing. The functional nature of the language means it's easy to start with small building blocks like our first parsers. Then we can gradually combine them to make something larger. It can be a little tricky to wrap our heads around all the different operators and combinators. But once you understand the ways in which these let us combine our parsers, they make a lot of sense and are easy to use.

You should now move onto part 3 of this series, where we will start learning about monadic parsing. You'll get to see how we use the Attoparsec library to parse this same Gherkin syntax!

To further your knowledge of useful Haskell libraries, download our free Production Checklist! It will tell you about libraries for many tasks, from databases to machine learning!

If you've never written a line of Haskell before, never fear! Download our Beginners Checklist to learn more!

Revision #1
Created 2022-03-11 16:12:29 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick