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

Parsing with Haskell

Haskell is an excellent language for all your parsing needs. The functional nature of the language makes it easy to compose different building blocks together without worrying about nasty side effects and unforeseen consequences. Since the language is so well-suited for parsing, there are several different libraries out there. Each of them differ a bit in their approaches. We'll explore three libraries in this series.

- Parsing Primer: Gherkin Syntax
- Applicative Parsing
- Attoparsec
- Megaparsec

Parsing Primer: Gherkin Syntax

Haskell is a truly awesome language for parsing. Haskell expressions tend to compose in simple ways with very clearly controlled side effects. This provides an ideal environment in which to break down parsing into simpler tasks. Thus there are many excellent parsing libraries out there.

In this series, we'll be taking a tour of some of these libraries. But before we look at specific code, it will be useful to establish a common example for what we're going to be parsing. In this first part, I'll introduce Gherkin Syntax, the language behind the Cucumber framework. We'll go through the language specifics, then show the basics of how we set ourselves up for success in Haskell.

If you're already familiar with Gherkin syntax, feel free to move on to part 2 of this series! Or if you want to challenge yourself with more libraries to learn, you can download our Production Checklist. It'll give you a survey of libraries for many different tasks!

Like some of our other series, there's a Github Repository that has all the code for these tutorials! You can observe some Gherkin examples in this directory. We'll compose our Haskell types in this module.

GHERKIN BACKGROUND

Cucumber is a framework for Behavior Driven Development. Under BDD, we first describe all the general behaviors we want our code to perform in plain language. This paradigm is an alternative to Test Driven Development. There, we use test cases to determine our next programming objectives. But BDD can do both of these if we can take behavior descriptions and automatically create tests from them! This would allow less technical members of a project team to effectively write tests!

The main challenge of this is formalizing a language for describing these behaviors. If we have a formal language, then we can parse it. If we can parse it into a reasonable structure, then we can turn that structure into runnable test code. This series will focus on the second part of this problem: turning Gherkin Syntax into a data structure (a Haskell data structure, in our case).

GHERKIN SYNTAX

Gherkin syntax has many complexities, but for these articles we'll be focusing on the core elements of it. The behaviors you want to test are broken down into a series of features. We describe each feature in its own .feature file. So our overarching task is to read input from a single file and turn it into a Feature object.

We begin our description of a feature with the Feature keyword (obviously). We'll give it a title, and then give it an indented description (our example will be a simple banking app):

```
Feature: Registering a User
  As a potential user
  I want to be able to create an account with a username,
    email and password
  So that I can start depositing money into my account
```

Each feature then has a series of scenarios. These describe specific cases of what can happen as part of this feature. Each scenario begins with the Scenario keyword and a title.

```
Scenario: Successful registration
  ...

Scenario: Email is already taken
  ...

Scenario: Username is already taken
  ...
```

Each scenario then has a series of Gherkin statements. These statements begin with one of the keywords Given, When, Then, or And. You should use Given statements to describe pre-conditions of the scenario. Then you'll use When to describe the particular action a user is taking to initiate

the scenario. And finally, you'll use Then to describe the after effects.

```
Scenario: Email is already taken
  Given there is already an account with the email "test@test.com"
  When I register an account with username "test",
    email "test@test.com" and password "1234abcd!?"
  Then it should fail with an error:
    "An account with that email already exists"
You can supplement any of these cases with a statement beginning with And.

Scenario: Email is already taken
  Given there is already an account with the email "test@test.com"
  And there is already an account with the username "test"
  When I register an account with username "test",
    email "test@test.com" and password "1234abcd!?"
  Then it should fail with an error:
    "An account with that email already exists"
  And there should still only be one account with
    the email "test@test.com"
```

Gherkin syntax does not enforce that you use the different keywords in a semantically sound way. We could start every statement with Given and it would still work. But obviously you should do whatever you can to make your tests sound correct.

We can also fill in statements with variables in angle brackets. We'll then follow the scenario with a table of examples for those variables:

```
Scenario: Successful Registration
  Given There is no account with username <username>
    or email <email>
  When I register the account with username <username>,
    email <email> and password <password>
  Then it should successfully create the account
    with <username>, <email>, and <password>
  Examples:
    | username | email              | password      |
    | john doe | john@doe.com       | ABCD1234!?    |
    | jane doe | jane.doe@gmail.com | abcdefgh1.aba |
    | jackson  | jackson@yahoo.com  | cadsw4ll0p/   |
```

We can also create a Background for the whole feature. This is a scenario-like description of preconditions that exist for every scenario in that feature. This can also have an example table:

```
Feature: User Log In
  ...

Background:
  Given: There is an existing user with username <username>,
    email <email> and password <password>
  Examples:
    | username | email              | password      |
    | john doe | john@doe.com       | ABCD1234!?    |
    | jane doe | jane.doe@gmail.com | abcdefgh1.aba |
```

And that's the whole language we're going to be working with!

HASKELL DATA STRUCTURES

Let's appreciate now how easy it is to create data structures in Haskell to represent this syntax. We'll start with a description of a Feature. It has a title, description (which we'll treat as a list of multiple lines), the background, and then a list of scenarios. We'll also treat the background like a "nameless" scenario that may or may not exist:

```
data Feature = Feature
  { featureTitle :: String
  , featureDescription :: [String]
  , featureBackground :: Maybe Scenario
  , featureScenarios :: [Scenario]
  }
```

Now let's describe what a Scenario is. It's main components are its title and a list of statements. We'll also observe that we should have some kind of structure for the list of examples we'll provide:

```
data Scenario = Scenario
  { scenarioTitle :: String
  , scenarioStatements :: [Statement]
  , scenarioExamples :: ExampleTable
```

```
  }
```

This ExampleTable will store a list of possible keys as well as list of tuple maps. Each tuple will contain keys and values. At the scale we're likely to be working at, it's not worth it to use a full Map:

```
data ExampleTable = ExampleTable
  { exampleTableKeys :: [String]
  , exampleTableExamples :: [[(String, Value)]]
  }
```

Now we'll have to define what we mean by a Value. We'll keep it simple and only use literal bools, strings, numbers, and a null value:

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

And finally we'll describe a statement. This will have the string itself, as well as a list of variable keywords to interpolate:

```
data Statement = Statement
  { statementText :: String
  , statementExampleVariables :: [String]
  }
```

And that's all there is too it! We can put all these types in a single file and feel pretty good about that. In Java or C++, we would want to make a separate file (or two!) for each type and there would be a lot more boilerplate involved.

GENERAL PARSING APPROACH

Another reason we'll see that Haskell is good for parsing is the ease of breaking problems down into smaller pieces. We'll have one function for parsing an example table, a different function for parsing a statement, and so on. Then gluing these together will actually be slick and simple!

CONCLUSION

Now you can move on to part 2 where we'll actually look at how to start parsing this. The first library we'll use is the regex-applicative parsing library. We'll see how we can get a lot of what we want without even using a monadic context!

For some more ideas on parsing libraries you can use, check out our free Production Checklist. It will tell you about different libraries for parsing as well as a great many other tasks, from data structures to web APIs!

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

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"
```haskell
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.

```
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:

```
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!

Attoparsec

In part 2 of this series we looked at the Regex-based Applicative Parsing library. We took a lot of smaller combinators and put them together to parse our Gherkin syntax (check out part 1 for a quick refresher on that).

This week, we'll look at a new library: Attoparsec. Instead of trying to do everything with a purely applicative structure, this library uses a monadic approach. This approach is much more common. It results in syntax that is simpler to read and understand. It will also make it easier for us to add certain features.

To follow along with the code for this article, take a look at the AttoParser module on Github! For some more excellent ideas about useful libraries, download our Production Checklist! It includes material on libraries for everything from data structures to machine learning!

Finally, if you already know about Attoparsec, feel free to move onto part 4 and learn about Megaparsec!

THE PARSER TYPE

In applicative parsing, all our parsers had the type RE Char. This type belonged to the Applicative typeclass but was not a Monad. For Attoparsec, we'll instead be using the Parser type, a full monad. So in general we'll be writing parsers with the following types:

```
featureParser :: Parser Feature
scenarioParser :: Parser Scenario
statementParser :: Parser Statement
exampleTableParser :: Parser ExampleTable
valueParser :: Parser Value
```

PARSING VALUES

The first thing we should realize though is that our parser is still an Applicative! So not everything needs to change! We can still make use of operators like *> and <|>. In fact, we can leave our value parsing code almost exactly the same! For instance, the valueParser, nullParser, and boolParser expressions can remain the same:

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

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

boolParser :: Parser 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"
```

If we wanted, we could make these more "monadic" without changing their structure. For instance, we can use return instead of pure (since they are identical). We can also use >> instead of *> to perform monadic actions while discarding a result. Our value parser for numbers changes a bit, but it gets simpler! The authors of Attoparsec provide a convenient parser for reading scientific numbers:

```
numberParser :: Parser Value
numberParser = ValueNumber <$> scientific
```

Then for string values, we'll use the takeTill combinator to read all the characters until a vertical bar or newline. Then we'll apply a few text functions to remove the whitespace and get it back to a String. (The Parser monad we're using parses things as Text rather than String).

```
stringParser :: Parser Value
stringParser = (ValueString . unpack . strip) <$>
  takeTill (\c -> c == '|' || c == '\n')
```

PARSING EXAMPLES

As we parse the example table, we'll switch to a more monadic approach by using do-syntax. First, we establish a cellParser that will read a value within a cell.

```
cellParser = do
  skipWhile nonNewlineSpace
  val <- valueParser
  skipWhile (not . barOrNewline)
  char '|'
  return val
```

Each line in our statement refers to a step of the parsing process. So first we skip all the leading whitespace. Then we parse our value. Then we skip the remaining space, and parse the final vertical bar to end the cell. Then we'll return the value we parsed.

It's a lot easier to keep track of what's going on here compared to applicative syntax. It's not hard to see which parts of the input we discard and which we use. If we don't assign the value with <- within do-syntax, we discard the value. If we retrieve it, we'll use it. To complete the exampleLineParser, we parse the initial bar, get many values, close out the line, and then return them:

```
exampleLineParser :: Parser [Value]
exampleLineParser = do
  char '|'
  cells <- many cellParser
  char '\n'
  return cells
  where
    cellParser = ...
```

Reading the keys for the table is almost identical. All that changes is that our cellParser uses many letter instead of valueParser. So now we can put these pieces together for our exampleTableParser:

```
exampleTableParser :: Parser ExampleTable
exampleTableParser = do
  string "Examples:"
  consumeLine
  keys <- exampleColumnTitleLineParser
  valueLists <- many exampleLineParser
  return $ ExampleTable keys (map (zip keys) valueLists)
```

We read the signal string "Examples:", followed by consuming the line. Then we get our keys and values, and build the table with them. Again, this is much simpler than mapping a function like buildExampleTable like in applicative syntax.

STATEMENTS

The Statement parser is another area where we can improve the clarity of our code. Once again, we'll define two helper parsers. These will fetch the portions outside brackets and then inside brackets, respectively:

```
nonBrackets :: Parser String
nonBrackets = many (satisfy (\c -> c /= '\n' && c /= '<'))

insideBrackets :: Parser String
insideBrackets = do
  char '<'
  key <- many letter
  char '>'
  return key
```

Now when we put these together, we can more clearly see the steps of the process outlined in do-syntax. First we parse the “signal” word, then a space. Then we get the “pairs” of non-bracketed and bracketed portions. Finally, we'll get one last non-bracketed part:

```
parseStatementLine :: Text -> Parser Statement
parseStatementLine signal = do
  string signal
  char ' '
  pairs <- many ((,) <$> nonBrackets <*> insideBrackets)
  finalString <- nonBrackets
  ...
```

Now we can define our helper function buildStatement and call it on its own line in do-syntax. Then we'll return the resulting Statement. This is much easier to read than tracking which functions we map over which sections of the parser:

```
parseStatementLine :: Text -> Parser Statement
parseStatementLine signal = do
  string signal
  char ' '
  pairs <- many ((,) <$> nonBrackets <*> insideBrackets)
  finalString <- nonBrackets
  let (fullString, keys) = buildStatement pairs finalString
  return $ Statement fullString keys
  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)
```

SCENARIOS AND FEATURES

As with applicative parsing, it's now straightforward for us to finish everything off. To parse a scenario, we read the keyword, consume the line to read the title, and read the statements and examples:

```
scenarioParser :: Parser Scenario
scenarioParser = do
  string "Scenario: "
```

```
  title <- consumeLine
  statements <- many (parseStatement <* char '\n')
  examples <- (exampleTableParser <|> return (ExampleTable [] []))
  return $ Scenario title statements examples
```

Again, we provide an empty ExampleTable as an alternative if there are no examples. The parser for Background looks very similar. The only difference is we ignore the result of the line and instead use Background as the title string.

```
backgroundParser :: Parser Scenario
backgroundParser = do
  string "Background:"
  consumeLine
  statements <- many (parseStatement <* char '\n')
  examples <- (exampleTableParser <|> return (ExampleTable [] []))
  return $ Scenario "Background" statements examples
```

Finally, we'll put all this together as a feature. We read the title, get the background if it exists, and read our scenarios:

```
featureParser :: Parser Feature
featureParser = do
  string "Feature: "
  title <- consumeLine
  maybeBackground <- optional backgroundParser
  scenarios <- many scenarioParser
  return $ Feature title maybeBackground scenarios
```

FEATURE DESCRIPTION

One extra feature we'll add now is that we can more easily parse the “description” of a feature. We omitted them in applicative parsing, as it's a real pain to implement. It becomes much simpler when using a monadic approach. The first step we have to take though is to make one parser for all the main elements of our feature. This approach looks like this:

```
featureParser :: Parser Feature
featureParser = do
  string "Feature: "
  title <- consumeLine
  (description, maybeBackground, scenarios) <- parseRestOfFeature
  return $ Feature title description maybeBackground scenarios

parseRestOfFeature :: Parser ([String], Maybe Scenario, [Scenario])
parseRestOfFeature = ...
```

Now we'll use a recursive function that reads one line of the description at a time and adds to a growing list. The trick is that we'll use the choice combinator offered by Attoparsec.

We'll create two parsers. The first assumes there are no further lines of description. It attempts to parse the background and scenario list. The second reads a line of description, adds it to our growing list, and recurses:

```
parseRestOfFeature :: Parser ([String], Maybe Scenario, [Scenario])
parseRestOfFeature = parseRestOfFeatureTail []
  where
    parseRestOfFeatureTail prevDesc = do
      (fullDesc, maybeBG, scenarios) <- choice [noDescriptionLine prevDesc, descriptionLine
prevDesc]
      return (fullDesc, maybeBG, scenarios)
```

So we'll first try to run this noDescriptionLineParser. It will try to read the background and then the scenarios as we've always done. If it succeeds, we know we're done. The argument we passed is the full description:

```
where
  noDescriptionLine prevDesc = do
    maybeBackground <- optional backgroundParser
    scenarios <- some scenarioParser
    return (prevDesc, maybeBackground, scenarios)
```

Now if this parser fails, we know that it means the next line is actually part of the description. So we'll write a parser to consume a full line, and then recurse:

```
descriptionLine prevDesc = do
  nextLine <- consumeLine
  parseRestOfFeatureTail (prevDesc ++ [nextLine])
```

And now we're done! We can parse descriptions!

CONCLUSION

That wraps up our exploration of Attoparsec. Now you can move on to the fourth and final part of this series where we'll learn about Megaparsec. We'll find that it's syntactically very similar to Attoparsec with a few small exceptions. We'll see how we can use some of the added power of monadic parsing to enrich our syntax.

To learn more about cool Haskell libraries, be sure to check out our Production Checklist! It'll tell you a little bit about libraries in all kinds of areas like databases and web APIs.

If you've never written Haskell at all, download our Beginner's Checklist! It'll give you all the resources you need to get started on your Haskell journey!

Megaparsec

In part 3 of this series, we explored the Attoparsec library. It provided us with a clearer syntax to work with compared to applicative parsing, which we learned in part 2. This week, we'll explore one final library: Megaparsec.

This library has a lot in common with Attoparsec. In fact, the two have a lot of compatibility by design. Ultimately, we'll find that we don't need to change our syntax a whole lot. But Megaparsec does have a few extra features that can make our lives simpler.

To follow the code examples here, head to the Github repository and take a look at the MegaParser module on Github! To learn about more awesome libraries you can use in production, make sure to download our Production Checklist! But never fear if you're new to Haskell! Just take a look at our Beginners checklist and you'll know where to get started!

A DIFFERENT PARSER TYPE

To start out, the basic parsing type for Megaparsec is a little more complicated. It has two type parameters, e and s, and also comes with a built-in monad transformer ParsecT.

```
data ParsecT e s m a = ParsecT ...


type Parsec e s = ParsecT e s Identity
```

The e type allows us to provide some custom error data to our parser. The s type refers to the input type of our parser, typically some variant of String. This parameter also exists under the hood in Attoparsec. But we sidestepped that issue by using the Text module. For now, we'll set up our own type alias that will sweep these parameters under the rug:

```
type MParser = Parsec Void Text
```

TRYING OUR HARDEST

Let's start filling in our parsers. There's one structural difference between Attoparsec and Megaparsec. When a parser fails in Attoparsec, its default behavior is to backtrack. This means it acts as though it consumed no input. This is not the case in Megaparsec! A naive attempt to repeat our nullParser code could fail in some ways:

```
nullParser :: MParser Value
nullParser = nullWordParser >> return ValueNull
  where
    nullWordParser = string "Null" <|> string "NULL" <|> string "null"
```

Suppose we get the input "NULL" for this parser. Our program will attempt to select the first parser, which will parse the N token. Then it will fail on U. It will move on to the second parser, but it will have already consumed the N! Thus the second and third parser will both fail as well!

We get around this issue by using the try combinator. Using try gives us the Attoparsec behavior of backtracking if our parser fails. The following will work without issue:

```
nullParser :: MParser Value
nullParser = nullWordParser >> return ValueNull
  where
    nullWordParser =
      try (string "Null") <|>
      try (string "NULL") <|>
      try (string "null")
```

Even better, Megaparsec also has a convenience function string' for case insensitive parsing. So our null and boolean parsers become even simpler:

```
nullParser :: MParser Value
nullParser = M.string' "null" >> return ValueNull

boolParser :: MParser Value
boolParser =
  (trueParser >> return (ValueBool True)) <|>
```

```
  (falseParser >> return (ValueBool False))
    where
      trueParser = M.string' "true"
      falseParser = M.string' "false"
```

Unlike Attoparsec, we don't have a convenient parser for scientific numbers. We'll have to go back to our logic from applicative parsing, only this time with monadic syntax.

```
numberParser :: MParser Value
numberParser = (ValueNumber . read) <$>
  (negativeParser <|> decimalParser <|> integerParser)
  where
    integerParser :: MParser String
    integerParser = M.try (some M.digitChar)

    decimalParser :: MParser String
    decimalParser = M.try $ do
      front <- many M.digitChar
      M.char '.'
      back <- some M.digitChar
      return $ front ++ ('.' : back)

    negativeParser :: MParser String
    negativeParser = M.try $ do
      M.char '-'
      num <- decimalParser <|> integerParser
      return $ '-' : num
```

Notice that each of our first two parsers use try to allow proper backtracking. For parsing strings, we'll use the satisfy combinator to read everything up until a bar or newline:

```
stringParser :: MParser Value
stringParser = (ValueString . trim) <$>
  many (M.satisfy (not . barOrNewline))
```

And then filling in our value parser is easy as it was before:

```
valueParser :: MParser Value
valueParser =
```

```
  nullParser <|>
  boolParser <|>
  numberParser <|>
  stringParser
```

FILLING IN THE DETAILS

Aside from some trivial alterations, nothing changes about how we parse example tables. The Statement parser requires adding in another try call when we're grabbing our pairs:

```
parseStatementLine :: Text -> MParser Statement
parseStatementLine signal = do
  M.string signal
  M.char ' '
  pairs <- many $ M.try ((,) <$> nonBrackets <*> insideBrackets)
  finalString <- nonBrackets
  let (fullString, keys) = buildStatement pairs finalString
  return $ Statement fullString keys
  where
    buildStatement  = ...
```

Otherwise, we'll fail on any case where we don't use any keywords in the statement! But it's otherwise the same. Of course, we also need to change how we call our parser in the first place. We'll use the runParser function instead of Attoparsec's parseOnly. This takes an extra argument for the source file of our parser to provide better messages.

```
parseFeatureFromFile :: FilePath -> IO Feature
parseFeatureFromFile inputFile = do
  ...
  case runParser featureParser finalString inputFile of
    Left s -> error (show s)
    Right feature -> return feature
```

But nothing else changes in the structure of our parsers. It's very easy to take Attoparsec code and Megaparsec code and re-use it with the other library!

ADDING SOME STATE

One bonus we do get from Megaparsec is that its monad transformer makes it easier for us to use other monadic functionality. Our parser for statement lines has always been a little bit clunky. Let's clean it up a little bit by allowing ourselves to store a list of strings as a state object. Here's how we'll change our parser type:

```
type MParser = ParsecT Void Text (State [String])
```

Now whenever we parse a key using our brackets parser, we can append that key to our existing list using modify. We'll also return the brackets along with the string instead of merely the keyword:

```
insideBrackets :: MParser String
insideBrackets = do
  M.char '<'
  key <- many M.letterChar
  M.char '>'
  modify (++ [key]) -- Store the key in the state!
  return $ ('<' : key) ++ ['>']
```

Now instead of forming tuples, we can concatenate the strings we parse!

```
parseStatementLine :: Text -> MParser Statement
parseStatementLine signal = do
  M.string signal
  M.char ' '
  pairs <- many $ M.try ((++) <$> nonBrackets <*> insideBrackets)
  finalString <- nonBrackets
  let fullString = concat pairs ++ finalString
  ...
```

And now how do we get our final list of keys? Simple! We get our state value, reset it, and return everything. No need for our messy buildStatement function!

```
parseStatementLine :: Text -> MParser Statement
parseStatementLine signal = do
```

```
  M.string signal
  M.char ' '
  pairs <- many $ M.try ((++) <$> nonBrackets <*> insideBrackets)
  finalString <- nonBrackets
  let fullString = concat pairs ++ finalString
  keys <- get
  put []
  return $ Statement fullString keys
```

When we run this parser at the start, we now have to use runParserT instead of runParser. This returns us an action in the State monad, meaning we have to use evalState to get our final result:

```
parseFeatureFromFile :: FilePath -> IO Feature
parseFeatureFromFile inputFile = do
  ...
  case evalState (stateAction finalString) [] of
    Left s -> error (show s)
    Right feature -> return feature
  where
    stateAction s = runParserT featureParser inputFile s
```

BONUSES OF MEGAPARSEC

As a last bonus, let's look at error messages in Megaparsec. When we have errors in Attoparsec, the parseOnly function gives us an error string. But it's not that helpful. All it tells us is what individual parser on the inside of our system failed:

```
>> parseOnly nullParser "true"
Left "string"
>> parseOnly "numberParser" "hello"
Left "Failed reading: takeWhile1"
```

These messages don't tell us where within the input it failed, or what we expected instead. Let's compare this to Megaparsec and runParser:

```
>> runParser nullParser "true" ""
Left (TrivialError
```

```
  (SourcePos {sourceName = "true", sourceLine = Pos 1, sourceColumn = Pos 1} :| [])
  (Just EndOfInput)
  (fromList [Tokens ('n' :| "ull")]))
>> runParser numberParser "hello" ""
Left (TrivialError
  (SourcePos {sourceName = "hello", sourceLine = Pos 1, sourceColumn = Pos 1} :| [])
    (Just EndOfInput)
    (fromList [Tokens ('-' :| ""),Tokens ('.' :| ""),Label ('d' :| "igit")]))
```

This gives us a lot more information! We can see the string we're trying to parse. We can also see the exact position it fails at. It'll even give us a picture of what parsers it was trying to use. In a larger system, this makes a big difference. We can track down where we've gone wrong either in developing our syntax, or conforming our input to meet the syntax. If we customize the e parameter type, we can even add our own details into the error message to help even more!

CONCLUSION

This wraps up our exploration of parsing libraries in Haskell! In the past few weeks, we've learned about Applicative parsing, Attoparsec, and Megaparsec. The first provides useful and intuitive combinators for when our language is regular. It allows us to avoid using a monad for parsing and the baggage that might bring. With Attoparsec, we saw an introduction to monadic style parsing. This provided us with a syntax that was easier to understand and where we could see what was happening. Finally in this part, we explored Megaparsec. This library has a lot in common syntactically with Attoparsec. But it provides a few more bells and whistles that can make many tasks easier.

Ready to explore some more areas of Haskell development? Want to get some ideas for new libraries to learn? Download our Production Checklist! It'll give you a quick summary of some tools in areas ranging from data structures to web APIs!

Never programmed in Haskell before? Want to get started? Check out our Beginners Checklist! It has all the tools you need to start your Haskell journey!