

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

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

```

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!

Revision #1

Created 11 March 2022 16:17:03 by gasick

Updated 11 March 2022 17:11:16 by gasick