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

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!

Revision #1
Created 2022-03-11 16:19:21 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick