Parsing CSS with Parsec (/2014/08/10/parsing-css-with-parsec.html)

AUG 10, 2014

This article is a small introduction to Parsec (http://hackage.haskell.org/package/parsec), the Haskell parser combinator library for Haskell. We'll use it to parse simple CSS file such as the following.

```
.container h1 {
  color: rgba(255, 0, 0, 0.9);
  font-size: 24px;
}
```

First we need to figure out our data structure which will represent the syntax tree. Since this is just an introduction, we'll go easy and ignore features like media queries.

In order to create the structure we need to figure out how to name things. We can look at the grammar definition for CSS 2.1 (http://www.w3.org/TR/CSS2/grammar.html) to figure out how things are named, from which we can tell that the main unit is a `ruleset`, has a `selector` and a list of `declarations`. Let's call it a `rule` instead of a `declaration` to keep things short. Each rule then has a `property` and a `value`.

```
type Selector = String
data Rule = Rule String String deriving Show
data Ruleset = Ruleset Selector [Rule] deriving Show
```

Basic parsec combinators

The way that parsec work is that you build up small parsers and combine them into bigger ones. We could write a parser for a rule, such as `color: red;`, which would first parse a property, then a colon, then some optional spaces and finally a value with an optional semicolon at the end.

Here are some basic parsers from the Parsec library.

- `char` - Parses a single character.
- `string` - Parses an arbitrary string.
- `optional` - Takes a parser and makes it optional.
- `many` - Takes a parser for a single item and makes it into a parser for 0 to N items.
- `many1` - Same as `many`, only that it requires at least one.
- `letter` - Parses any letter.
- `digit` - Parses a digit.

To parse a colon we could do `char ':'`. If that colon was optional, we can just combine it with the `optional` combinator, such as `optional (char ':')`, and so on.

Here's how a simple parser for `Rule` could look like.

Recent articles

- Fibonacci Numbers (/2015/08/08/fibonacci-numbers.html)
- Parsing CSS with Parsec (/2014/08/10/parsing-css-with-parsec.html)
- Lens Tutorial - Stab & Traversal (Part 2) (/2014/08/06/lens-tutorial-stab-traversal-part-2.html)
- Foldable and Traversable (/2014/07/30/foldable-and-traversable.html)
- Building Monad Transformers - Part 1 (/2014/07/22/building-monad-transformers-part-1.html)
- Mutable State in Haskell (/2014/07/20/mutable-state-in-haskell.html)
- Lens Tutorial - Introduction (part 1) (/2014/07/14/lens-tutorial-introduction-part-1.html)
- Using Phantom Types in Haskell for Extra Safety - Part 2 (/2014/07/10/using-phantom-types-in-haskell-for-extra-safety-part-2.html)
- Using Phantom Types for Extra Safety (/2014/07/08/using-phantom-types-for-extra-safety.html)
- Evil Mode: How I Switched From VIM to Emacs (/2014/06/23/evil-mode-how-to-switch-from-vim-to-emacs.html)

Tags

- clojure (1) (/tags/clojure.html)
- testing (1) (/tags/testing.html)
- rspec (1) (/tags/rspec.html)
- ruby (1) (/tags/ruby.html)
- refactoring (1) (/tags/refactoring.html)
- haskell (9) (/tags/haskell.html)
- emacs (1) (/tags/emacs.html)
- lens (2) (/tags/lens.html)
- algorithms (1) (/tags/algorithms.html)

Archives

- 2015 (1) (/2015.html)
- 2014 (13) (/2014.html)
- 2013 (3) (/2013.html)

Google+ (https://plus.google.com/+JakubArnold?rel=author)

```
import Text.Parsec
import Text.Parsec.String

rule :: Parser Rule
rule = do
    p <- many1 letter
    char ':'
    optional (char ' ')

    v <- many1 letter
    optional (char ';')

    return $ Rule p v
```

You might have already noticed that `Parser` is indeed a Monad, which is why we're using the `do` notation, and why we are able to combine many small parsers together. This is where the power of Parsec comes in, because it is very easy to combine small parsers to build something that you can use in the real world.

Now comes the time to test our parser. Parsec defines a function called `parse`, which accepts a parser, a source name and a source string, and returns `Either` a `ParseError` if our parsing failed, or the parsed value.

```
λ> import Text.Parsec
λ> parse (char ':') "test parser" ":"
Right ':'
λ> parse (char ':') "test parser" "a"
Left "test parser" (line 1, column 1):
unexpected "a"
expecting ":"
λ> parse (many1 letter) "test parser" "hello"
Right "hello"
```

We might also need to say something like *parse any number of letters or digits*. This is where the `<|>` combinator comes in, which allows us to say the *or* part. It takes two parsers as arguments and returns a new parser, which tries to parse with the first one, and if it fails tries the second one.

```
λ> parse (many1 $ letter <|> digit) "test parser" "hello123"
Right "hello123"
```

When parsing fails

There is one thing very important to understand here. As the parsers try to parse the input, they consume it. If you use `<|>` to combine two parsers together, and the first parser fails after already consuming some input, the second parser will continue where the first one left off. Here's an example that illustrates this.

```
λ> parse (string "hay" <|> string "hoy") "test parser" "hoy"
Left "test parser" (line 1, column 1):
unexpected "o"
expecting "hay"
```

We're trying to parse either `"hay"` or `"hoy"`, giving it an input of `"hoy"`.

It seems that this should obviously succeed, but it doesn't, because the first parser consumes the first character `"h"` and then it fails.

We could write rewrite this in another way.

```
λ> parse (char 'h' >> (string "ey" <|> string "oy")) "test parser" "hoy"
Right "oy"
```

If we used a do block we could've actually achieved the end result of parsing the whole `"hoy"` string, but there's a much easier way, by using `try`.

Using `try` on any parser makes it backtrack when it fails while consuming a part of the input.

```
λ> parse (try (string "hay") <|> string "hoy") "test parser" "hoy"
Right "hoy"
```

Note that we only need to use `try` with our first parser, since there is nothing left to do if the second parser fails. `try` doesn't affect how the parser works, only what happens when it fails.

The bottom line here is, if you're combining together multiple parsers where one could consume some input and then fail, use `try`. There are cases when you don't need to do this, such as when we did `letter <|> digit`. Since those two parsers don't overlap in their domain, we don't need to use `try` there.

The reason why parsec behaves this way is simply because of performance. Careless usage of `try` can make the parser slower, but since we're just trying to understand how things work, we don't need to worry about this.

The CSS parser

Before we move on any further, let's improve our original `rule` parser. We didn't really account for spaces, since CSS rules can be indented and there can be arbitrary number of spaces after the `:`, and values can have more than just letters, such as `#FFF`.

We can use the `spaces` parser which skips *zero or more whitespace characters*.

```
rule :: Parser Rule
rule = do
    p <- many1 letter
    char ':'
    spaces

    v <- many1 letter
    char ';'

    return $ Rule p v
```

Next we need to figure out how to tell parsec that we also want things other than just letters in the property value. We could use `oneOf` to name all of the symbols we'd like to accept, such as `oneOf "#()%"`, which parses one character out of the given set. But to keep things simple,

let's just say that the value can be *anything but a* `;`. We can use the `noneOf` combinator for that. We'll also make the `;` non-optional to save ourselves some trouble, and accept any number of whitespace after the whole rule definition.

```
rule :: Parser Rule
rule = do
    p <- many1 letter
    char ':'
    spaces

    v <- many1 (noneOf ";")
    char ';'
    spaces

    return $ Rule p v
```

Let's test this out.

```
λ> parse rule "css parser" "background: #fafafa;"
Right (Rule "background" "#fafafa")
λ> parse rule "css parser" "background: rgba(255, 255, 255, 0.3);"
Right (Rule "background" "rgba(255, 255, 255, 0.3)")
```

We can even try parsing multiple rules at once using `many1`.

```
λ> parse (many1 rule) "css parser" "background: rgba(255, 255, 255, 0.3); col
or: red;\nborder: 1px solid black;"
Right [Rule "background" "rgba(255, 255, 255, 0.3)",Rule "color" "red",Rule "
border" "1px solid black"]
```

Now moving onto the parser for a whole ruleset. Let's do the same thing as we did with values and say that *a selector can be any character except for {*. Next we have the `{`, followed by any number of spaces, followed by a list of rules, followed by a closing `}`.

```
ruleset :: Parser Ruleset
ruleset = do
    s <- many1 (noneOf "{")
    char '{'
    spaces

    r <- many1 rule
    char '}'
    spaces

    return $ Ruleset s r
```

Let's test this out.

```
λ> parse ruleset "css parser" "p { color: red; }"
Right (Ruleset "p " [Rule "color" "red"])
λ> parse ruleset "css parser" "p { background: #fafafa;\n color: red; }"
Right (Ruleset "p " [Rule "background" "#fafafa",Rule "color" "red"])
```

And everything seems to be working properly. You might notice that our

selector is being parsed as `"p "` instead of just `"p"`. This is because we were too relaxed on our definition, but that's easy to fix. But first let's do a bit of refactoring.

Refactoring the parser using Applicative

Because the `Parser` monad is also an instance of `Applicative`, we can use a lot of the combinators that `Applicative` gives us to cleanup our code. The most useful ones are `*>` and `<*`, where `*>` takes two parsers, runs the first one, throws away the result, then runs the second one and returns its result (exactly the same as `>>` does for monads). `<*` does the same thing, but the other way around, here are a couple of examples.

The reason why I'm hiding the `<|>` in the import here is because we need the definition from Parsec, not from Applicative.

```
λ> import Control.Applicative hiding ((<|>))
λ> parse (spaces *> string "hello") "test parser" "  hello"
Right "hello"
λ> parse (char '(' *> string "hello" <* char ')') "test parser" "(hello)"
Right "hello"
λ> parse (char ';' <* spaces) "test parser" ";     "
Right ';'
λ> parse (char ';' <* spaces) "test parser" "; \n\n"
Right ';'
```

As you can see the usage is pretty straightforward. We could write the same thing using a `do` notation, but using the Applicative combinators make the code easier to read once you get used to them. You can think of them as pointing in the direction of the result.

Here's how we could refactor our rule parser.

```
rule :: Parser Rule
rule = do
  p <- many1 letter <* char ':' <* spaces
  v <- many1 (noneOf ";") <* char ';' <* spaces

  return $ Rule p v
```

We can even define a helper that would take a parser and apply `<* spaces` to it, since we're using that quite a lot, but this is just a matter of taste.

```
paddedChar c = char c <* spaces

rule :: Parser Rule
rule = do
  p <- many1 letter <* paddedChar ':'
  v <- many1 (noneOf ";") <* paddedChar ';'

  return $ Rule p v
```

Let's do the same thing for our ruleset parser.

```
ruleset :: Parser Ruleset
ruleset = do
    s <- many1 (noneOf "{")
    r <- paddedChar '{' *> many1 rule <* paddedChar '}'

    return $ Ruleset s r
```

Now that we have refactored everything, it's time to make the selector parsing more strict. The new parser will be defined as *a sequence of characters consisting of letters, numbers, dots and hashes, separated by spaces*`.

```
selector :: Parser String
selector = many1 (oneOf ".#" <|> letter <|> digit) <* spaces
```

Then the actual selector for the ruleset will be just many of our `selector` parsers in a row, separated by spaces. We'll use the `sepBy1` combinator for this, which takes a parser specifying the separator and returns a list of parsed values.

```
λ> parse (selector `sepBy1` spaces) "test parser" ".container h1 "
Right [".container","h1"]
```

Now that we've succesfully parsed the selector, we can combine it back into a single string using the `unwords` function from prelude.

```
ruleset :: Parser Ruleset
ruleset = do
    s <- selector `sepBy1` spaces
    r <- paddedChar '{' *> many1 rule <* paddedChar '}'

    return $ Ruleset (unwords s) r
```

And let's test this once again to make sure everything works.

```
λ> parse ruleset "css parser" ".container h1 { color: red; }"
Right (Ruleset ".container h1" [Rule "color" "red"])
```

As you can see, our selector now doesn't contain the trailing spaces.

Closing thoughts

The parser we developed in this article is far from complete, but feel free to extend it to support things like pseudo classes, comments, etc.

While it's not so common to do TDD in Haskell, I'd recommend writing a lot of unit tests for your parser. It's easy to play around in the REPL and test things out, but once you start composing multiple parsers together it gets very tedious to have to check different versions of the string you're parsing every time you make a change. Unlike in regular Haskell code you can't really rely on the type system that much, since you're just working with strings.

upcoming book,
Haskell by Example?

Subscribe to receive updates and free content from the book. You'll also get a discount when the final version of the book is released.

First Name

Email Address

Keep me updated