An introduction to parsing text in Haskell with Parsec

Parsec makes parsing text very easy in Haskell. I write this as much for myself as for anyone else to have a tutorial and reference which starts from the ground up and works through how each function can be used with examples all the way.

First off, why would you use Parsec as opposed to things like regular expressions for parsing content? Coming from other languages, splitting content into arrays and processing ever smaller chunks using regular expressions and the like can be quite common practise. In Haskell we can go down that route too, but now I've seen the light with Parsec I want to introduce you to a better approach.

Most tutorials get stuck in with a complete example straight off the bat, but I'm going to present the different functions one by one so that they can be used as a reference of reminder (to me as much as anyone!) of how things work. I'll try to keep examples relatively self contained, so it shouldn't be hard to skip to bits, but keep in mind the basic setup I do first. I have also put all of the example code into a file right here that is ready to be `:load` ed straight into `ghci` and played with.

The Basics

Parsec works its way along some stream of text from beginning to end, attempting to match the stream of inputs to some rule or a set of rules. Parsec is also monadic, so we can piece together our different rules in sequence using the convenient do notation. As a general overview, rules work by consuming a character at a time from the input and determining whether they match or not, so when you piece a number of rules in sequence, each rule will consume part of the input until you have no input left, run out of rules, or one of your rules fails to match (resulting in an error).

Let's start with a very basic setup. I import `Parsec` qualified so it's plainly obvious when I'm using `Parsec` functions, and also import `Control.Applicative` so we can play with parsing things in the applicative style later on, and make a short alias for `parseTest` for use in the examples:

```
-- I import qualified so that it's clear which
-- functions are from the parsec library:
import qualified Text.Parsec as Parsec

-- I am the error message infix operator, used later:
import Text.Parsec ((<?>))

-- Imported so we can play with applicative things later.
-- not qualified as mostly infix operators we'll be using.
import Control.Applicative

-- Get the Identity monad from here:
import Control.Monad.Identity (Identity)

-- alias Parsec.parse for more concise usage in my examples:
parse rule text = Parsec.parse rule "(source)" text
```

This will be our base setup, with a simple utility function parse defined which just ignored the second argument of Parsec.parse (which in the real world would be the name of the file whose contents you are parsing, and is just used in error messages by Parsec to give you that piece of extra information).

Parsec comes with a host of building blocks, each of which can be a rule in its own right, or can be combined with others to build more complex rules up. Let's look at some of these basic building blocks and see how they work with the above setup.

Parsec.char

This function returns a rule that matches the current character in the text that we are parsing to whatever character you provide it. Lets have a go in the interactive ghci console:

```
ghci> someText = "Hello Hello Hello World World World"
ghci> parse (Parsec.char 'H') someText
Right 'H'
ghci> parse (Parsec.char 'e') someText
Left "(source)" (line 1, column 1):
unexpected "H"
expecting "e"
```

Here, running `Parsec.char 'H'` returns a rule that will match a single character so long as it's 'H'. If we thus use it in parsing our string, which begins with an H, it is perfectly happy. If we try looking for any letter that isn't 'H', we fail. The result is always of type `Either ParseError res` ; we get back a `Right result` if the rule was successful, and `Left error` if the rule fails. Thus, we can pattern match to see which the case is simply enough:

```
main = do
    let result = parse (Parsec.char 'H') "Hello"
    case result of
        Right v -> putStrLn "success!"
        Left err -> putStrLn ("whoops, error: "++show err)
```

Parsec.string

This function returns a rule that attempts to match the provided string of characters:

```
ghci> parse (Parsec.string "hello") "hello world!"
Right "hello"
ghci> parse (Parsec.string "hello") "howdy"
Left "(source)" (line 1, column 1):
unexpected "o"
expecting "hello"
```

The parser will consume characters from the input one by one until all characters match or one of them is not as expected. Since both of the above attempts begin with 'h', the error complains about an unexpected 'o'. This consuming of characters will become significant when multiple rules are chained together.

`Parsec.oneOf`

Sometimes we want to match a range of characters; that's where Parsec.oneOf comes in handy. Similar to `Parsec.char`, except you provide this function a list of characters you're OK with matching:

```
ghci> parse (Parsec.oneOf "abcde") "allo"
Right 'a'
ghci> parse (Parsec.oneOf "abcde") "chewy"
Right 'c'
ghci> parse (Parsec.oneOf "abcde") "gnaw"
Left "(source)" (line 1, column 1):
unexpected "g"
```

Here, we can see that the parser will *consume* any single character from 'a' to 'e'. We can use ranges here to simplify things, so to match any lowercase letter we'd just use `Parsec.oneOf ['a'..'z']` for instance.

Parsec comes with pre-prepared rules which do just that, for example `Parsec.anyChar`, which will consume one of anything:

```
ghci> parse Parsec.anyChar "blahblah"
Right 'b'
ghci> parse Parsec.anyChar "=-symbols..."
Right "="
```

The rule `Parsec.letter` will happily consume any lower or uppercase letter, `Parsec.lower` will consume any lowercase letter, `Parsec.digit` will consume any number, and `Parsec.alphaNum` any letter or number. All of these can be constructed manually using `Parsec.oneOf` as above, though they come with nicer error messages (which you can add to your own rules; we'll look at that later).

Parsec.noneOf

The opposite of the above, you provide this function a list of characters that aren't allowed and it'll match anything else. Again, ranges are your friend:

```
ghci> parse (Parsec.noneOf ['0'..'9']) "hello"
Right 'h'
ghci> parse (Parsec.noneOf ['0'..'9']) "100"
Left "(source)" (line 1, column 1):
unexpected "1"
```

Parsec.many and Parsec.many1

Let's face it, there are times when you want to parse more than just one letter. `Parsec.many` tries to run whatever rule it is given as an argument over and over until it fails. Even if the rule matches no times, many will return without error, but just give back an empty result. Let's see how it's used:

```
ghci> parse (Parsec.many (Parsec.char 'h')) "hhhheeelllooo!"
Right 'hhhh'
ghci> parse (Parsec.many (Parsec.char 'e')) "hhhheeelllooo!"
Right ''
ghci> parse (Parsec.many Parsec.letter) "hhhheeelllooo!"
Right 'hhhheeelllooo'
```

As we can see, `Parsec.many` can't go wrong, as it'll happily match the rule it's given zero times and just return nothing. It'll go as far as it can and give you back everything that matched. `Parsec.many1` is similar except that the rule it's provided must match at least once for it to return successfully:

```
ghci> parse (Parsec.many1 Parsec.letter) "hello!!"
Right 'hello'
ghci> parse (Parsec.many1 Parsec.letter) "75 hello's!"
Left "(source)" (line 1, column 1):
unexpected "7"
expecting letter
```

Useful when you need to match some set of letters or numbers for instance, but must see at least one.

Parsec.count

If you need to match a specific number of something, `Parsec.count` comes in handy. It takes a number and a parser, and expects to match that parser that number of times (or will fail), returning the result. Here's an example:

```
ghci> parse (Parsec.count 4 Parsec.letter) "ahoythere"
Right "ahoy"
ghci> parse (Parsec.count 4 Parsec.letter) "aho"
Left "(source)" (line 1, column 4):
unexpected end of input
expecting letter
```

Parsec.manyTill

This parser takes in two arguments; the rule it will try matching and the rule it expects to be able to match right after it. As with many, it expects zero or more of the first rule, but it will error on anything that matches neither rule. The below tries matching letters and then expects numbers immediately following them:

```
ghci> parse (Parsec.manyTill Parsec.letter Parsec.digit) "hello12345"
Right "hello"
ghci> parse (Parsec.manyTill Parsec.letter Parsec.digit) "12345"
Right ""
ghci> parse (Parsec.manyTill Parsec.letter Parsec.digit) "hello 12345"
Left "(source)" (line 1, column 6):
unexpected " "
expecting digit or letter
```

It's important to keep in mind the fact that this will consume (and output) all of the first rule, and then consume whatever the second rule matches (but ignore it in the output). When we start stringing together rules in sequence, it becomes increasingly important what we consume and what we leave ready for the next rule to have a go at.

One thing I think is great about Parsec is that it provides useful error information straight off the bat; in this case the string we passed in back at the beginning ("*(source)*") along with the line and column number of the error and some useful message as to what went wrong. We're only dealing with single lines at the moment, but to have this from the word go is really cool.

Combining Rules

Now you have some of the basic rules under your belt, lets talk about how to combine them! Parsec, being monadic, allows you to write parsers using Haskell's *do* notation sugar. Here's an example that puts some simple parsers above in sequence, getting the results of a couple and returning them:

```
-- This looks for letters, then spaces, then digits.
-- we then return letters and digits in a tuple.
myParser :: Parsec.Parsec String () (String,String)
myParser = do
    letters <- Parsec.many1 Parsec.letter
    Parsec.spaces
    digits <- Parsec.many1 Parsec.digit
    return (letters,digits)
```

Note that I have given this parser an explicit type of `Parsec.Parsec String () (String,String)`. The arguments to this type in order are simply input type, some state you'd like threaded through your parsers (we'll use the unit type for now, and so have no meaningful state, and have a quick look at state later), then output type. In this case, we're taking in a String and returning a tuple of two strings. If you use `:type` to inspect the type of rules in ghci, you'll see that they are made from the `ParsecT` type, not `Parsec`. `ParsecT` is just a monad transformer which has the same type parameters as `Parsec.Parsec`, along with a new parameter `m` representing the monad that this wraps. Needless to say, these two types are equivalent:

```
-- I have to import the identity monad to use in the ParsecT definition:
import Control.Monad.Identity (Identity)

myParser1 :: Parsec.ParsecT String () Identity (String,String)
myParser1 = myParser
```

```
myParser2 :: Parsec.Parsec String () (String,String)
myParser2 = myParser
```

When you inspect the types of functions in the `Parsec` package, bare this in mind to help understand what you are dealing with! Every rule is of a similar type, though the return value varies from rule to rule. `Parsec.many`, for instance, returns an array of matches. Have a look yourself in ghci!

Anyway, now we have defined `myParser`, We can use it with our `parse` function as so:

```
ghci> parse myParser "hello 1000"
Right ("hello","1000")
ghci> parse myParser "woohoooo0!!"
Right ("woohoooo","0")
ghci> parse myParser "1000"
Left "(source)" (line 1, column 1):
unexpected "1"
expecting letter
```

Because we have used `Parsec.many1`, we require there to be at least one letter followed by zero or more spaces and finally at least one number. Our rule then wraps these into a tuple for us (but could instead return them packaged in a custom type or any other arrangement).

Say we have a number of these letter/digit pairs, separated by some separator, for example a comma. In this case, we might want to parse them all into a list of tuples of the type seen above. Let's define another rule to parse our separator:

```
mySeparator :: Parsec.Parsec String () ()
mySeparator = do
    Parsec.spaces
    Parsec.char ','
    Parsec.spaces
```

I have once again added an explicit type signature. I do this because, when written independent of any usage in my test file, Haskell cannot determine which types things will be. Note that the only thing returned is the last line, which has a type matching our signature; values from any other parsing done

prior to this return are ignored. We could also add an explicit `return ()` at the end, but `Parsec.spaces` returns this anyway.

This rule matches zero or more spaces, followed by a comma, followed by zero or more spaces. Given that we don't care about grabbing and returning any of the values parsed by these rules, we can *desugar* the above into a one liner:

```
mySeparator = Parsec.spaces >> Parsec.char ',' >> Parsec.spaces
```

Now we have `myParser`, and `mySeparator`, each made from smaller parsing rules. In the same way that we have created them, we can now combine our new rules to create ever larger rules. I'll start with a more verbose rule, built from things we have learned above:

```
--I want to return a list of pairs, this time.
myPairs :: Parsec.Parsec String () [(String,String)]
myPairs = Parsec.many $ do
    pair <- myParser
    mySeparator
    return pair
```

This basically uses `Parsec.many` to parse 0 or more instances of `myParser` followed by `mySeparator`, saving the result of myParser and returning it. Notice that I use do sugar to build up a rule which is then passed as an argument to `Parsec.many`. This desugars to the following, which makes it more clear that everything inside the `do` block is simply one argument to `Parsec.many`:

```
myPairs = Parsec.many (myParser >>= \pair -> mySeparator >> return pair)
```

Given that `Parsec.many` returns a list of whatever is passed in (see the last bit of its type signature), the result is thus a list of `(String,String)` pairs. Let's give it a go:

```
ghci> parse myPairs "hello 1, byebye 2,"
Right [("hello","1"),("byebye","2")]
ghci> parse myPairs ""
Right []
ghci> parse myPairs "hello 1, byebye 2"
```

```
Left "(source)" (line 1, column 18):
unexpected end of input
expecting digit, white space or ","
```

As we can see, as a result of using `Parsec.many`, the parser will happily find no instances, but if the parser successfully begins matching input, failure (for example a missing separator at the end) leads to an error being thrown. As it happens, there are built in functions for parsing the common pattern of items split by some separator.

Parsec.endBy

Takes two arguments, a rule to parse items, and a rule to parse separators. `Parsec.endBy` essentially does exactly as above, and expects a string consisting of rule then separator, returning an array of whatever the rule returns:

```
-- I want to return a list of pairs as above but using a built in helper:
myPairs2a :: Parsec.Parsec String () [(String,String)]
myPairs2a = Parsec.endBy myParser mySeparator
```

Parsec.sepBy

Takes two arguments as `endBy` does, but this time does not expect the separator to come after the final instance of the rule we're parsing.

```
-- I want to return a list of pairs without a final separator:
myPairs2b :: Parsec.Parsec String () [(String,String)]
myPairs2b = Parsec.sepBy myParser mySeparator
```

As such, it does not require a final separator (in fact the presence of one would be an error):

```
ghci> parse myPairs2b "hello 1, bye 2"
Right [("hello","1"),("bye","2")]
```

`Parsec.choice` and `<|>` for matching one of multiple rules

Using `Parsec.choice` or the shorthand infix operator `Parsec.<|>` (also in `Control.Applicative`) we can present more than one rule to parse, and the first rule that **successfully consumes input** is used (even if it later fails; a caveat we'll get to). Let's see how this works in practice by removing the need for a separator at the end of our pairs:

```
--I want to return a list of pairs with an optional end separator.
myPairs2 :: Parsec.Parsec String () [(String,String)]
myPairs2 = Parsec.many $ do
    pair <- myParser
    Parsec.choice [Parsec.eof, mySeparator]
    return pair
```

Now, our rule will consume many letter-digit pairs, each followed by either the end of file (a rule parsec provides) or our defined separator. This can also be written using the infix operator as:

```
import Text.Parsec (<|>)

myPairs3 :: Parsec.Parsec String () [(String,String)]
myPairs3 = Parsec.many $ do
    pair <- myParser
    Parsec.eof <|> mySeparator
    return pair
```

Where I make note to import the `<|>` operator so I don't have to prefix it and make it look ugly in use. Both the infix operator and `Parsec.choice` support as many alternatives as you like, for example `Parsec.choice [rule1, rule2, rule3]` or `rule1 <|> rule2 <|> rule3`. In either case, the first rule in the sequence to consume some input is the rule that is then used. By accepting either end of file or our custom separator, we no longer require the separator at the end:

```
ghci> parse myPairs2 "hello 1, byebye 2,"
Right [("hello","1"),("byebye","2")]
```

```
Right [("hello","1"),("byebye","2")]
ghci> parse myParis2 "hello 1, byebye 2"
```

The important thing to remember here is that the first rule that consumes input is the one that is used. This can lead to unexpected failure. Take the following example:

```
parse (Parsec.string "hello" <|> Parsec.string "howdy") "howdy"
```

Offhand one might expect the parser to attempt to match "hello", and on failure to match that, succeed instead at matching "howdy". In actuality the parsing fails entirely:

```
ghci> parse (Parsec.string "hello" <|> Parsec.string "howdy") "howdy"
Left "(source)" (line 1, column 1):
unexpected "o"
expecting "hello"
```

This is because on attempting to match the string "hello", the rule created from `Parsec.string "hello"` consumes the 'h' successfully, and is thus rule selected for use, before subsequently failing on the next character. Taking another example, this becomes clearer:

```
ghci> parse (Parsec.string "hello" <|> Parsec.string "bye") "bye"
Right "bye"
```

Here, the first rule fails before it can successfully consume any input, and so the second rule is selected without issue. By default, Parsec will not "look ahead" to see whether a rule matches or not, for performance reasons. The first solution to this (and probably most performant) is to parse any input that is common to either match separately, and then try the rest, avoiding the need to perform any lookahead, for example:

```
ghci> parse (Parsec.char 'h' >> (Parsec.string "ello" <|> Parsec.string "owdy")) "
Right "owdy"
```

Noting that since we are ignoring the result of the first parser (which consumes the 'h'), we no longer return the full string. This is simple enough to rectify if necessary; we'll turn our inline notation into a more explicit rule to do so:

```
helloOrHowdy :: Parsec.Parsec String () String
helloOrHowdy = do
    first <- Parsec.char 'h'
    rest <- Parsec.string "ello" <|> Parsec.string "owdy"
    return (first:rest)
```

By manually deciding what to return from our rule, we can choose to return the correct string by appending our initial char to the rest of the string. Errors are now based on the parts of the string that each rule tries to consume rather than the whole thing, increasing their precision but perhaps at the cost of clarity:

```
ghci> parse helloOrHowdy "hello"
Right "hello"
ghci> parse helloOrHowdy "allo"
Left "(source)" (line 1, column 1):
unexpected "a"
expecting "h"
ghci> parse helloOrHowdy "hoops"
Left "(source)" (line 1, column 2):
unexpected "o"
expecting "owdy"
```

The first error comes from `Parsec.char`, the second from the failing `Parsec.string`. We'll show you how to provide your own custom errors instead if you'd prefer, but first lets look at lookahead as a neater way to parse these strings.

Parsec.try

Avoiding lookahead can quickly become unwieldy when rules become more complex. In these cases, we can instruct Parsec to try a rule, and rewind us to the previous state if the rule fails at any point. `Parsec.try` does just this; it catches any failure and rewinds us. Due to the performance implications, it's best to keep the lookahead bounded to as small a region as possible; the less

potential parsing is wrapped in a try function, the better. `Parsec.try` also suppresses any error messages you would otherwise have been given in the wrapped rule, and so can lead to odd and unhelpful error messages if used inappropriately. That said, when used properly it can be taken advantage of to give better errors. Let's give it a go:

```
helloOrHowdy2 :: Parsec.Parsec String () String
helloOrHowdy2 = Parsec.try (Parsec.string "hello") <|> Parsec.string "howdy"
```

This results in the correct parsing, and generally clear errors, but since any errors from failure to parse hello are suppressed, errors will only describe a failure to match the choice operator or a failure to match "howdy", ignoring the failure to match "hello":

```
ghci> parse helloOrHowdy2 "hello"
Right "hello"
ghci> parse helloOrHowdy2 "howdy"
Right "howdy"
ghci> parse helloOrHowdy2 "boo!"
Left "(source)" (line 1, column 1):
unexpected "b"
expecting "hello" or "howdy"
ghci> parse helloOrHowdy2 "hellay"
Left "(source)" (line 1, column 1):
unexpected "e"
expecting "howdy"
```

Creating your own error messages with <?>

Sometimes, often when building up your own rules, you want to use your own custom error messages in case of parsing failure. `<?>` allows you to attach a custom error to any rule quite simply. Let's see it in action:

```
ghci> parse (Parsec.string "hello") "wrongstring"
Left "(source)" (line 1, column 1):

unexpected "w"
expecting "hello"
ghci> parse (Parsec.string "hello" <?> "a common greeting") "wrongstring"
Left "(source)" (line 1, column 1):
```

```
Left "(source)" (line 1, column 1):
unexpected "w"
expecting a common greeting
```

Here, we simply attach a new error message to the rule we create from `Parsec.string`. `<?>` has the lowest precedence possible, which means that anything else will evaluate first. Attaching a new error message to the end of a chain of rules created with `<|>` for instance will result in that error being used if all of the rules fail *without consuming any input*, as then the rule generated from the `<|>` chain has failed. As soon as a rule consumes input, it becomes up to the error reporting of that rule to describe future failure (unless of course a try block surrounds the rule, which suppresses error messages from it). This basic example illustrates the fact:

```
ghci> -- this fails without consuming any input:
ghci> parse (Parsec.string "apple" <|> Parsec.string "bat" <?> "boom!") "cat"
Left "(source)" (line 1, column 1):
unexpected "c"
expecting boom!
ghci> -- this consumes input before failing:
ghci> parse (Parsec.string "apple" <|> Parsec.string "bat" <?> "boom!") "aunty"
Left "(source)" (line 1, column 1):
unexpected "u"
expecting "apple"
```

If you'd like to provide a single custom error message for a rule you've created, you can incase the rule in a `try` to catch any errors that would otherwise originate from it, and instead provide your own in the event that the rule fails. Here's a simple example doing just that:

```
-- here we parse a basic greeting with no custom errors:
greeting :: Parsec.Parsec String () String
greeting = do
    Parsec.char 'h'
    Parsec.string "olla" <|> Parsec.string "ello"
    return "greeting"

--parse the same greeting, but wrap in try and add custom error:
greeting2 :: Parsec.Parsec String () String
greeting2 = Parsec.try greeting <?> "a greeting!"
```

This is not recommended for more significant rules as it would replace precise error messages from sub rules with some general and less helpful error.

However, when building small rules it can be more descriptive to provide your own error over those `Parsec` provides.

Applicative functions for more concise parsing

The module `Control.Applicative` imports several functions, mostly infix operators, that can help make your rules more concise and readable in the right situations. It turns out that we've already been using one such operator, `<|>`, already! Applicative functions often make code shorter, since they are all about being point-free, that is, not making explicit references to the variables being passed around.

Let's convert our original pair parser to applicative style and then run over what each operator does:

```
-- lets start again with our first parser to parse a letter/digit pair:
myParser :: Parsec.Parsec String () (String,String)
myParser = do
    letters <- Parsec.many1 Parsec.letter
    Parsec.spaces
    digits <- Parsec.many1 Parsec.digit
    return (letters,digits)

-- in applicative style:
myParserApp :: Parsec.Parsec String () (String,String)
myParserApp = (,) <$> Parsec.many1 Parsec.letter <*> (Parsec.spaces *> Parsec.many

-- could also be written as:
myParserApp2 :: Parsec.Parsec String () (String,String)
myParserApp2 = liftA2 (,) (Parsec.many1 Parsec.letter) (Parsec.spaces *> Parsec.ma

-- or even (swapping *> for the more familiar >>):
myParserApp :: Parsec.Parsec String () (String,String)
myParserApp2 = liftA2 (,) (Parsec.many1 Parsec.letter) (Parsec.spaces >> Parsec.ma
```

Let's run through the main applicative operators one by one and see what they each actually do:

<$> and <*>

This operator is essentially `fmap` . it takes a function on the left and a rule on the right, and applies the function to the result of the rule (provided the rule succeeds, otherwise we get a parse error instead) before returning. If we want to apply the function to more than one argument, we separate the arguments wih <*>. lets run through a couple of examples in ghci:

```
ghci> -- apply the result to a tuple constructor:
ghci> parse ((,) <$> Parsec.char 'a' <*> Parsec.char 'b') "ab"
Right ('a','b')
ghci> -- put the result into an array:
ghci> parse ((\a b -> [a,b]) <$> Parsec.char 'a' <*> Parsec.char 'b') "ab"
Right "ab"
```

The neat thing about this is that you can chain as many arguments as you need to the function by adding <*> 's to the end.

`liftAx`

A prefix version of the above, `liftAx` takes *x* number of subsequent arguments and applies them to the first. Not as flexible as the infix versions but there may be times when it is more readable. Here are the same examples as above with lift:

```
ghci> -- apply the result to a tuple constructor:
ghci> parse (liftA2 (,) (Parsec.char 'a') (Parsec.char 'b')) "ab"
Right ('a','b')
ghci> -- put the result into an array:
ghci> parse (liftA2 (\a b -> [a,b]) (Parsec.char 'a') (Parsec.char 'b')) "ab"
Right "ab"
```

<* and *>

Sometimes you want to run some rules. discarding the results of all but one.

These operators each take two rules and return the result of the one that the angle bracket points to. Examples:

```
ghci> parse (Parsec.char 'a' <* Parsec.char 'b') "ab"
Right 'a'
ghci> parse (Parsec.char 'a' *> Parsec.char 'b') "ab"
Right 'b'
```

They can also be chained in such a way that you can ignore several rules:

```
ghci> parse (Parsec.char 'a' <* Parsec.char 'b' <* Parsec.char 'c') "abc"
Right 'a'
ghci> parse (Parsec.char 'a' *> Parsec.char 'b' <* Parsec.char 'c') "abc"
Right 'b'
ghci> parse (Parsec.char 'a' *> Parsec.char 'b' *> Parsec.char 'c') "abc"
Right 'c'
```

And often some in handy when you want to do things like trim whitespace or other bits and pieces from around some piece of information.

<$

Runs the rule on the right, and then returns whatever is on the left if the rule succeeds. Let's see it in action along with some equivalent ways to do the same thing:

```
ghci> parse ("greeting!" <$ Parsec.string "hello") "hello"
Right "greeting!"
ghci> parse (Parsec.string "hello" >> return "greeting!") "hello"
Right "greeting!"
ghci> parse (return "greeting!" <* Parsec.string "hello") "hello"
```

As you can see, there are no shortage of different ways to do things! I can see myself using the more obvious second example most often despite it being a few extra characters from the first, but whatever floats you boat!

Handling state

More recently I learned that you can thread state through your parsers as well (thanks Chobbsy!). This is useful for keeping track of things like amount of indentation. Here is a very basic example using state to count a letter:

```
-- matches char 'h', incrementing int state by 1
-- each time one is seen.
hCountParser :: Parsec.Parsec String Int ()
hCountParser = do
    Parsec.char 'h'
    c <- Parsec.getState
    let c' = c+1
    Parsec.putState c'
    return ()

-- parse as many h's as we can, then return the state
-- to see how many there were
Parsec.runParser (Parsec.many hCountParser >> Parsec.getState) 0 "" "hhhhhhhhhhhhe
```

As well as getting and setting, we can use `Parsec.modifyState` to modify state in place. Thus, a simpler version of `hCountParser` would be:

```
hCountParser' :: Parsec.Parsec String Int ()
hCountParser' = do
    Parsec.char 'h'
    Parsec.modifyState (+1)
    return ()
```

It's worth noting however that being a monad transformer, we also have the option of combining a parser with something like the `State` monad for instance in order to thread state through it in an approach that is more consistent with the monad transformer way of doing things. Using the `State` Monad instead, we would have something like:

```
import Control.Monad       (lift)
import Control.Monad.State as S

hCountParser'' :: Parsec.ParsecT String () (S.State Int) ()
hCountParser'' = do
    char 'h'
```

```
    lift $ modify (+1)

-- after running our parser transformer, we get back our unevaluated inner state, w
-- contains our parser result and state ('h' count). We only want the state so
-- we use execState rather than runState or evalState to execute and unwrap the sta
-- providing an initial state to start the ball rolling.
S.execState (Parsec.runParserT (Parsec.many hCountParser2) () "" "hhhhhhhhhhhhello
```

Wrapping Things up

We have wandered through some of the built in functions and rules provided. We've then looked at combining rules to build larger ones, branching by way of allowing a choice of rules, lookahead by way of `try`, and finally adding custom error messages to your rules and had a quick play with maintaining state. Armed with the above, you should be off to a good start!

I encourage you to peruse `Parsec` functions in `ghci` by importing the `Parsec` module under some alias (or qualified as I have done) and using tab complete to get a list of everything that `Parsec` provides. using `:type` on these can provide further insight, and forms the basis for how I explored my way through many of the available functions. The chapter here at *Real World Haskell* is also excellent and runs through far more substantial examples, though note that some small pieces are a little out of date.

I hope that this has been helpful! If I've left anything out, please leave a comment and let me know!

Similar Posts

Haskell: Some Awesome Language Extensions Explained 2015/08/21

A living post, partly for my own reference, containing details of some of the more awesome/useful/interesting looking Haskell language extensions with plenty of examples. I'll update it as I have more to add.

Haskell programming

Haskell: A Look at the Continuation Monad

2015/06/06

Some notes I developed on the continuation monad in Haskell as a result of exploring how they work and how to use them, complete with plenty of examples along the way.

Haskell programming

Haskell: Another Lens Tutorial 2014/08/28

As someone who just got lenses after several days of trying to find out about them, I thought I'd reflect on this new understanding by showing you what a lens really is, starting from scratch.

Haskell programming

Comments

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Cutú Chiqueño • 2 years ago

Thx, as a beginner to Haskell I really enjoyed reading this tutorial. It not just explaines parsec in an easy way but also helped me understand better some infix operators which are also used outside of Parsec. I would really like to see mor of these kind of tutorials on often used libraries in Haskell to lower the burden for people new to this language.

1 • Reply • Share ›

Nick B • 3 years ago

Great article - found it much more practical than most of the other introductions to parsing in Haskell. Thanks!

1 • Reply • Share ›

Simone Trubian • 3 years ago

Thank you James for the this excellent tutorial, I finally got my head round parsing!

1 • Reply • Share ›

James Wilson Admin ➚ Simone Trubian • 3 years ago

Thanks :)

• Reply • Share ›

Evgeny Morozov • a year ago

To make "myPairs3" function work, I changed "import Text.Parsec (<|>)" to "import Text.Parsec.Combinator (choice)"

• Reply • Share ›

John Walker • 2 years ago

This is a truly awesome post:
1. for learning the basics parsec or with a few tweaks megaparsec
2. as an example of how to structure a great tutorial
Thanks heaps !!

• Reply • Share ›

Richard Careaga • 2 years ago

For someone who has been (nonprofessionally) involved with imperative languages for the past 50 years, coming up to speed on a functional language like Haskell can seem like undergoing an elaborate hazing ritual, reminiscent of the ancient Adventure game's twisty little mazes of passages all (alike|different). This post was a major help in understanding why what I eventually hit on

worked.

First, a comment on Ch. 16 of RWH. Beyond the CSV examples is an exercise in catching up with the cumulative changes from GHC 6.10.1. That's not something a beginner should attempt. This 2008 reference badly needs an update.

Second, in my flailing about I came across the original, 2001 Parsec User Guide by the package author, which has a 20-page tutorial with examples covering the basics clearly. It turned out to be not too difficult to add the pragmas, imports and type declarations now needed to make the examples work as described. For the benefit of future pilgrims, I've collected these as userguide.hs at https://github.com/technocr... for the benefit of future pilgrims.

Thanks again for this more advanced look.

 • Reply • Share ›

Severyn Kozak • 2 years ago

Nice article, especially the sections about the applicative style and adding state to your parser.

 • Reply • Share ›

Chobbsy • 3 years ago

"The result is always of type Either; Left result if the rule was successful, and Right error if the rule fails."

I think you have Left and Right backwards here! :)

Also the () in your parser type is for the user state. Parsec lets you keep track of your own state, which can be useful if you need to keep track of anything! Here's a function that helps with this:

http://hackage.haskell.org/...

The best little overview of the Parsec types that I have found is actually in this indentation parsing article, here:

http://spin.atomicobject.co...

"but this time does not expect the separator to some after the final instance of the rule we're parsing"

I think this is a typo! "come after the final instance?"

Good post, thanks for the Parsec review! :)

 • Reply • Share ›

James Wilson Admin ➚ Chobbsy • 3 years ago

Thanks for this! That was an embarrassing mistake :)

I have fixed these, and also added a bit about using state (parsec's own and state monad) at the end. I also updated the example.hs file to work with GHC 7.10 and removed all tabs, so hopefully the tutorial is a little more 'complete' now.

 • Reply • Share ›

Chobbsy ➚ James Wilson • 3 years ago

Wow that was quick! Nicely done!

Also, something seems a bit fishy with the disqus comments -- on the replies on disqus it seems to be using localhost...

https://disqus.com/by/lytnus/

The title "An introduction to parsing text in Haskell with Parsec" links to...

http://localhost:8080/#!/post/haskell-parsec-basics

As opposed to linking to unbui.lt. Might want to double check how it's embedded! I think there is a way to transfer the comments over if you change the URL... Don't quite remember.

Anyway, thanks for the changes! Cheers! :)

^ | v • Reply • Share ›

James Wilson Admin ➚ Chobbsy • 3 years ago

Thanks, I noticed that in the past but assumed it was just me! Turns out disqus associates the first URL that you visit the thread from with it, which is a bit pants if you're just testing it! I think I have fixed up the URL's though Im not sure if the changes will propagate everywhere. Frankly I have never found disqus particularly easy/reliable/stable :-/

^ | v • Reply • Share ›

Anonymous • 3 years ago

"First off, why would you use Parsec as opposed to things like regular expressions for parsing content?"

Reason #1: your input is not regular!

... and also latest studies show that shotgun parsers [1] cause global warming.

[1] http://langsec.org

^ | v • Reply • Share ›

Kaligule • 3 years ago

Thank you so much for this tutorial. It indeed is the best Parsec Tutorial I found. You have a typo there at "stirng" by the way :)

^ | v • Reply • Share ›

James Wilson Admin ➚ Kaligule • 3 years ago

Thankyou :) typo corrected!

^ | v • Reply • Share ›

Björn Döring • 3 years ago

Thanks for this easy to follow post! `Parsec` seems to be simple once you get it, but it's hard to actually find a good introductory tutorial on it!

Just a quick question: Using applicative style, how would you apply a function to the result of your parser before returning it? For instance, `myParser` in your example parses a tuple and returns type `(String, String)`. How could I modify the parser to return a tuple of type `(Integer, Integer)` by using the `read` function (assuming for the moment that the strings can actually be converted to Integers)?

Reply • Share ›

James Wilson Admin ➚ Björn Döring • 3 years ago

To do this sort of thing, I'd just make parsers to do each bit and then throw them together. A simple but not very safe version using read to parse two numbers into a tuple of Ints would involve creating a couple of parsers like so:

```
parseNumber = Parsec.many Parsec.digit >>= \a -> return $ (read a :: Int)
parseNumberWithSpaces = parseNumber <* Parsec.spaces
```

and then applying them applicative style like so:

```
parse ((,) <$> parseNumberWithSpaces <*> parseNumberWithSpaces) "123 456"
```

which would return in this case:

Right (123,456)

Note that if parsing as int fails you get a nasty error, but I'm just keeping things simple and

About Me

My name is James Wilson. I studied Cognitive Science at Hertfordshire University, graduating with a first class degree, and went on to complete a Ph.D in Developmental Robotics at Aberystwyth University. I've been programming since I was a kid, and am always trying out new languages. I have a particular interest in biologically inspired machine learning algorithms, and I love making things. Don't hesitate to get in touch if you want to know more.