Learning Parser Combinators With Rust

18 Apr 2019

This article teaches the fundamentals of parser combinators to people who are already Rust programmers. It assumes no other knowledge, and will explain everything that isn't directly related to Rust, as well as a few of the more unexpected aspects of using Rust for this purpose. It will not teach you Rust if you don't already know it, and, if so, it probably also won't teach you parser combinators very well. If you would like to learn Rust, I recommend the book The Rust Programming Language (https://doc.rust-lang.org/book/).

Beginner's Mind

There comes a point in the life of every programmer when they find themselves in need of a parser.

The novice programmer will ask, "what is a parser?"

The intermediate programmer will say, "that's easy, I'll write a regular expression."

The master programmer will say, "stand back, I know lex and yacc."

The novice has the right idea.

Not that regular expressions aren't great. (But please don't try writing a complicated parser as a regular expression.) Not that there's no joy to be had in employing powerful tools like parser and lexer generators that have been honed to perfection over millennia. But learning about parsers from the ground up is *fun*. It's also something you'll be missing out on if you stampede directly towards regular expressions or parser generators, both of which are only abstractions over the real problem at hand. In the beginner's mind, as the man said (https://en.wikipedia.org/wiki/Shunry%C5%AB_Suzuki#Quotations), there are many possibilities. In the expert's mind, there's only the one the expert got used to.

In this article, we're going to learn how to build a parser from the ground up using a technique common in functional programming languages known as *parser combinators*. They have the advantage of being remarkably powerful once you grasp the basic idea of them, while at the same time staying very close to first principles, as the only abstractions here will be the ones you build yourself on top of the basic combinators - all of which you'll also have to build before you get to use them.

How To Work Through This Article

It's highly recommended that you start a fresh Rust project and type the code snippets into `src/lib.rs` as you read (you can paste it directly from the page, but typing it in is better, as the act of doing so automatically ensures you read the code in its entirety). Every piece of code you're going to need is introduced by the article in order. Mind that it sometimes introduces *changed* versions of functions you've written previously, and that in these cases you should replace the old version with the new one.

The code was written for `rustc` version 1.34.0 using the 2018 edition of the language. You should be able to follow along using any version of the compiler that is more recent, as long as you make sure you're using the 2018 edition (check that your `Cargo.toml` contains `edition = "2018"`). The code needs no external dependencies.

To run the tests introduced in the article, as you might expect, you use `cargo test`.

The Xcruciating Markup Language

We're going to write a parser for a simplified version of XML. It looks like this:

```
<parent-element>
  <single-element attribute="value" />
</parent-element>
```

XML elements open with the symbol `<` and an identifier consisting of a letter followed by any number of letters, numbers and `-`. This is followed by some whitespace, and an optional list of attribute pairs: another identifier as defined previously, followed by a `=` and a double quoted string. Finally, there is either a closing `/>` to signify a single element with no children, or a `>` to signify there is a sequence of child elements following, and finally a closing tag starting with `</`, followed by an identifier which must match the opening tag, and a final `>`.

That's all we're going to support. No namespaces, no text nodes, none of the rest, and *definitely* no schema validation. We're not even going to bother supporting escape quotes for those strings - they start at the first double quote and they end at the next one, and that's it. If you want double quotes inside your actual strings, you can take your unreasonable demands somewhere else.

We're going to parse those elements into a struct that looks like this:

```
 #[derive(Clone, Debug, PartialEq, Eq)]
struct Element {
    name: String,
    attributes: Vec<(String, String)>,
    children: Vec<Element>,
}
```

No fancy types, just a string for a name (that's the identifier at the start of each tag), attributes as pairs of strings (identifier and value), and a list of child elements that look exactly the same as the parent.

(If you're typing along, make sure you include those derives. You're going to need them later.)

Defining The Parser

Well, then, it's time to write the parser.

Parsing is a process of deriving structure from a stream of data. A parser is something which teases out that structure.

In the discipline we're about to explore, a parser, in its simplest form, is a function which takes some input and returns either the parsed output along with the remainder of the input, or an error saying "I couldn't parse this."

It turns out that's also, in a nutshell, what a parser looks like in its more complicated forms. You might complicate what the input, the output and the error all mean, and if you're going to have good error messages you'll need to, but the parser stays the same: something that consumes input and either produces some parsed output along with what's left of the input or lets you know it couldn't parse the input into the output.

Let's write that down as a function type.

```
Fn(Input) -> Result<(Input, Output), Error>
```

More concretely, in our case, we'll want to fill out the types so we get something like this, because what we're about to do is convert a string into an `Element` struct, and at this point we don't want to get into the intricacies of error reporting, so we'll just return the bit of the string that we couldn't parse as an error:

```
Fn(&str) -> Result<(&str, Element), &str>
```

We use a string slice because it's an efficient pointer to a piece of a string, and we can slice it up further however we please, "consuming" the input by slicing off the bit that we've parsed and returning the remainder along with the result.

It might have been cleaner to use `&[u8]` (a slice of bytes, corresponding to characters if we restrict ourselves to ASCII) as the input type, especially because string slices behave a little differently from most slices - especially in that you can't index them with a single number `input[0]`, you have to use a slice `input[0..1]`. On the other hand, they have a lot of methods that are useful for parsing strings that slices of bytes don't have.

In fact, in general we're going to be relying on those methods rather than indexing it like that, because, well, Unicode. In UTF-8, and all Rust strings are UTF-8, these indexes don't always correspond to single characters, and it's better for all parties concerned that we ask the standard library to just please deal with this for us.

Our First Parser

Let's try writing a parser which just looks at the first character in the string and decides whether or not it's the letter `a`.

```
fn the_letter_a(input: &str) -> Result<(&str, ()), &str> {
  match input.chars().next() {
      Some('a') => Ok((&input['a'.len_utf8()..], ())),
      _ => Err(input),
  }
}
```

First, let's look at the input and output types: we take a string slice as input, as we've discussed, and we return a `Result` of either `(&str, ())` or the error type `&str`. The pair of `(&str, ())` is the interesting bit: as we've talked about, we're supposed to return a tuple with the next bit of input to parse and the result. The `&str` is the next input, and the result is just the unit type `()`, because if this parser succeeds, it could only have had one result (we found the letter `a`), and we don't particularly need to return the letter `a` in this case, we just need to indicate that we succeeded in finding it.

And so, let's look at the code for the parser itself. We start by getting the first character of the input: `input.chars().next()`. We weren't kidding about leaning on the standard library to avoid giving us Unicode headaches - we ask it to get us an iterator `chars()` over the characters of the string, and we pull the first item off it. This will be an item of type `char`, wrapped in an `Option`, so `Option<char>`, where `None` means we tried to pull a `char` off an empty string.

To make matters worse, a `char` isn't necessarily even what you think of as a character in Unicode. That would most likely be what Unicode calls a "grapheme cluster (http://www.unicode.org/glossary/#grapheme_cluster)," which can be composed of several `char`s, which in fact represent "scalar values (http://www.unicode.org/glossary/#unicode_scalar_value)," about two levels down from grapheme clusters. However, that way lies madness, and for our purposes we honestly aren't even likely to see any `char`s outside the ASCII set, so let's leave it there.

We pattern match on `Some('a')`, which is the specific result we're looking for, and if that matches, we return our success value: `Ok((&input['a'.len_utf8()..], ()))`. That is, we remove the bit we just parsed (the `'a'`) from the string slice and return the rest, along with our parsed value, which is just the empty `()`. Ever mindful of the Unicode monster, we ask the standard library for the length of `'a'` in UTF-8 before slicing - it's 1, but never, ever presume about the Unicode monster.

If we get any other `Some(char)`, or if we get `None`, we return an error. As you'll recall, our error type right now is just going to be the string slice at the point where parsing failed, which is the one that we got passed in as `input`. It didn't start with `a`, so that's our error. It's not a great error, but at least it's marginally better than just "something is wrong somewhere."

We don't actually need this parser to parse XML, though, but the first thing we need to do is look for that opening `<`, so we're going to need something very similar. We're also going to need to parse `>`, `/` and `=` specifically, so maybe we can make a function which builds a parser for the character we want?

A Parser Builder

Let's even get fancy about this: let's write a function that produces a parser for a static string of *any* length, not just a single character. It's even sort of easier that way, because a string slice is already a valid UTF-8 string slice, and we don't have to think about the Unicode monster.

```
fn match_literal(expected: &'static str)
    -> impl Fn(&str) -> Result<(&str, ()), &str>
{
    move |input| match input.get(0..expected.len()) {
        Some(next) if next == expected => {
            Ok((&input[expected.len()..], ()))
        }
        _ => Err(input),
    }
}
```

Now this looks a bit different.

First of all, let's look at the types. Instead of our function looking like a parser, now it takes our `expected` string as an argument, and *returns* something that looks like a parser. It's a function that returns a function - in other words, a *higher order* function. Basically, we're writing a function that *makes* a function like our `the_letter_a` function from before.

So, instead of doing the work in the function body, we return a closure that does the work, and that matches our type signature for a parser from previously.

The pattern match looks the same, except we can't match on our string literal directly because we don't know what it is specifically, so we use a match condition `if next == expected` instead. Otherwise it's exactly the same as before, it's just inside the body of a closure.

Testing Our Parser

Let's write a test for this to make sure we got it right.

```
#[test]
fn literal_parser() {
    let parse_joe = match_literal("Hello Joe!");
    assert_eq!(
        Ok(("", ())),
        parse_joe("Hello Joe!")
    );
    assert_eq!(
        Ok((" Hello Robert!", ())),
        parse_joe("Hello Joe! Hello Robert!")
    );
    assert_eq!(
        Err("Hello Mike!"),
        parse_joe("Hello Mike!")
    );
}
```

First, we build the parser: `match_literal("Hello Joe!")`. This should consume the string `"Hello Joe!"` and return the remainder of the string, or it should fail and return the whole string.

In the first case, we just feed it the exact string it expects, and we see that it returns an empty string and the `()` value that means "we parsed the expected string and you don't really need it returned back to you."

In the second, we feed it the string `"Hello Joe! Hello Robert!"`, and we see that it does indeed consume the string `"Hello Joe!"` and returns the remainder of the input: `" Hello Robert!"` (leading space and all).

In the third, we feed it some incorrect input, `"Hello Mike!"`, and note that it does indeed reject the input with an error. Not that Mike is incorrect as a general rule, he's just not what this parser was looking for.

Exercises ☝️

- Can you find a method on the `str` type in the standard library that would let you write `match_literal()` without having to do the somewhat cumbersome `get` indexing?

A Parser For Something Less Specific

So that lets us parse `<`, `>`, `=` and even `</` and `/>`. We're practically done already!

The next bit after the opening `<` is the element name. We can't do this with a simple string comparison. But we *could* do it with a regular expression…

…but let's restrain ourselves. It's going to be a regular expression that would be very easy to replicate in simple code, and we don't really need to pull in the `regex` crate just for this. Let's see if we can write our own parser for this using nothing but Rust's standard library.

Recalling the rule for the element name identifier, it's as follows: one alphabetical character, followed by zero or more of either an alphabetical character, a number, or a dash `-`.

```rust
fn identifier(input: &str) -> Result<(&str, String), &str> {
    let mut matched = String::new();
    let mut chars = input.chars();

    match chars.next() {
        Some(next) if next.is_alphabetic() => matched.push(next),
        _ => return Err(input),
    }

    while let Some(next) = chars.next() {
        if next.is_alphanumeric() || next == '-' {
            matched.push(next);
        } else {
            break;
        }
    }

    let next_index = matched.len();
    Ok((&input[next_index..], matched))
}
```

As always, we look at the type first. This time we're not writing a function to build a parser, we're just writing the parser itself, like our first time. The notable difference here is that instead of a result type of `()`, we're returning a `String` in the tuple along with the remaining input. This `String` is going to contain the identifier we've just parsed.

With that in mind, first we create an empty `String` and call it `matched`. This is going to be our result value. We also get an iterator over the characters in `input`, which we're going to start pulling apart.

The first step is to see if there's a letter up front. We pull the first character off the iterator and check if it's a letter: `next.is_alphabetic()`. Rust's standard library is of course here to help us with the Unicodes - this is going to match letters in any alphabet, not just ASCII. If it's a letter, we push it into our `matched` `String`, and if it's not, well, clearly we're not looking at an element identifier, so we return immediately with an error.

For the second step, we keep pulling characters off the iterator, pushing them onto the `String` we're building, until we find one that isn't either `is_alphanumeric()` (that's like `is_alphabetic()` except it also matches numbers in any alphabet) or a dash `'-'`.

The first time we see something that doesn't match those criteria, that means we're done parsing, so we break out of the loop and return the `String` we've built, remembering to slice off the bit we've consumed from the `input`. Likewise if the iterator runs out of characters, which means we hit the end of the input.

It's worth noting that we don't return with an error when we see something that isn't alphanumeric or a dash. We've already got enough to make a valid identifier once we've matched that first letter, and it's perfectly normal for there to be more things to parse in the input string after we've parsed our identifier, so we just stop parsing and return our result. It's only if we can't find even that first letter that we actually return an error, because in that case there was definitely not an identifier here.

Remember that `Element` struct we're going to parse our XML document into?

```rust
struct Element {
    name: String,
    attributes: Vec<(String, String)>,
    children: Vec<Element>,
}
```

We actually just finished the parser for the first part of it, the `name` field. The `String` our parser returns goes right in there. It's also the right parser for the first part of every `attribute`.

Let's test that.

```
#[test]
fn identifier_parser() {
    assert_eq!(
        Ok(("", "i-am-an-identifier".to_string())),
        identifier("i-am-an-identifier")
    );
    assert_eq!(
        Ok((" entirely an identifier", "not".to_string())),
        identifier("not entirely an identifier")
    );
    assert_eq!(
        Err("!not at all an identifier"),
        identifier("!not at all an identifier")
    );
}
```

We see that in the first case, the string `"i-am-an-identifier"` is parsed in its entirety, leaving only the empty string. In the second case, the parser returns `"not"` as the identifier, and the rest of the string is returned as the remaining input. In the third case, the parser fails outright because the first character it finds is not a letter.

Combinators

So now we can parse the opening `<`, and we can parse the following identifier, but we need to parse *both*, in order, to be able to make progress here. So the next step will be to write another parser builder function, but one that takes two *parsers* as input and returns a new parser which parses both of them in order. In other words, a parser *combinator*, because it combines two parsers into a new one. Let's see if we can do that.

```
fn pair<P1, P2, R1, R2>(parser1: P1, parser2: P2) -> impl Fn(&str) -> Result<(&st
where
    P1: Fn(&str) -> Result<(&str, R1), &str>,
    P2: Fn(&str) -> Result<(&str, R2), &str>,
{
    move |input| match parser1(input) {
        Ok((next_input, result1)) => match parser2(next_input) {
            Ok((final_input, result2)) => Ok((final_input, (result1, result2))),
            Err(err) => Err(err),
        },
        Err(err) => Err(err),
    }
}
```

It's getting slightly complicated here, but you know what to do: start by looking at the types.

First of all, we have four type variables: `P1`, `P2`, `R1` and `R2`. That's Parser 1, Parser 2, Result 1 and Result 2. `P1` and `P2` are functions, and you'll notice that they follow the well established pattern of parser functions: just like the return value, they take a `&str` as input and return a `Result` of a pair of the remaining input and the result, or an error.

But look at the result types of each function: `P1` is a parser that produces an `R1` if successful, and `P2` likewise produces an `R2`. And the result of the final parser - the one returned from our function - is `(R1, R2)`. So the job of this parser is to first run the parser `P1` on the input, keep its result, then run `P2` on the input that `P1` returned, and if both of those worked out, we combine the two results into a tuple `(R1, R2)`.

Looking at the code, we see that this is exactly what it does, too. We start by running the first parser on the input, then the second parser, then we combine the two results into a tuple and return that. If either of those parsers fail, we return immediately with the error it gave.

This way, we should be able to combine our two parsers from before, `match_literal` and `identifier`, to actually parse the first bit of our first XML tag. Let's write a test to see if it works.

```rust
#[test]
fn pair_combinator() {
    let tag_opener = pair(match_literal("<"), identifier);
    assert_eq!(
        Ok(("/>", ((), "my-first-element".to_string()))),
        tag_opener("<my-first-element/>")
    );
    assert_eq!(Err("oops"), tag_opener("oops"));
    assert_eq!(Err("!oops"), tag_opener("<!oops"));
}
```

It seems to work! But look at that result type: `((), String)`. It's obvious that we only care about the right hand value here, the `String`. This is going to be the case rather a lot - some of our parsers only match patterns in the input without producing values, and so their outputs can be safely ignored. To accommodate this pattern, we're going to use our `pair` combinator to write two other combinators: `left`, which discards the result of the first parser and only returns the second, and its opposite number, `right`, which is the one we'd have wanted to use in our test above instead of `pair` - the one that discards that `()` on the left hand side of the pair and only keeps our `String`.

Enter The Functor

But before we go that far, let's introduce another combinator that's going to make writing these two a lot simpler: `map`.

This combinator has one purpose: to change the type of the result. For instance, let's say you have a parser that returns `((), String)` and you wanted to change it to return just that `String`, you know, just as an arbitrary example.

To do that, we pass it a function that knows how to convert from the original type to the new one. In our example, that's easy: `|(_left, right)| right`. More generalised, it would look like `Fn(A) -> B` where `A` is the original result type of the parser and `B` is the new one.

```
fn map<P, F, A, B>(parser: P, map_fn: F) -> impl Fn(&str) -> Result<(&str, B), &st
where
    P: Fn(&str) -> Result<(&str, A), &str>,
    F: Fn(A) -> B,
{
    move |input| match parser(input) {
        Ok((next_input, result)) => Ok((next_input, map_fn(result))),
        Err(err) => Err(err),
    }
}
```

And what do the types say? `P` is our parser. It returns `A` on success. `F` is the function we're going to use to map `P` into our return value, which looks the same as `P` except its result type is `B` instead of `A`.

In the code, we run `parser(input)` and, if it succeeds, we take the `result` and run our function `map_fn(result)` on it, turning the `A` into a `B`, and that's our converted parser done.

Actually, let's indulge ourselves and shorten this function a bit, because this `map` thing turns out to be a common pattern that `Result` actually implements too:

```
fn map<P, F, A, B>(parser: P, map_fn: F) -> impl Fn(&str) -> Result<(&str, B), &st
where
    P: Fn(&str) -> Result<(&str, A), &str>,
    F: Fn(A) -> B,
{
    move |input|
        parser(input)
            .map(|(next_input, result)| (next_input, map_fn(result)))
}
```

This pattern is what's called a "functor" in Haskell and its mathematical sibling, category theory. If you've got a thing with a type `A` in it, and you have a `map` function available that you can pass a function from `A` to `B` into to turn it into the same kind of thing but with the type `B` in it instead, that's a functor. You see this a lot of places in Rust, such as in `Option` (https://doc.rust-lang.org/std/option/enum.Option.html#method.map), `Result` (https://doc.rust-lang.org/std/result/enum.Result.html#method.map), `Iterator` (https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.map) and even `Future` (https://docs.rs/futures/0.1.26/futures/future/trait.Future.html#method.map), without it being explicitly named as such. And there's a good reason for that: you can't really express a functor as a generalised thing in Rust's type system, because it lacks higher kinded types, but that's another story, so let's just note that these are functors, and you just need to look for the `map` function to spot one.

Time For A Trait

You might have noticed by now that we keep repeating the shape of the parser type signature: `Fn(&str) -> Result<(&str, Output), &str>`. You may be getting as sick of reading it written out full like that as I'm getting of writing it, so I think it's time to introduce a trait, to make things a little more readable, and to let us add some extensibility to our parsers.

But first of all, let's make a type alias for that return type we keep using:

```
type ParseResult<'a, Output> = Result<(&'a str, Output), &'a str>;
```

So that now, instead of typing that monstrosity out all the time, we can just type `ParseResult<String>` or similar. We've added a lifetime there, because the type declaration requires it, but a lot of the time the Rust compiler should be able to infer it for you. As a rule, try leaving the lifetime out and see if rustc gets upset, then just put it in if it does.

The lifetime `'a`, in this case, refers specifically to the lifetime of the *input*.

Now, for the trait. We need to put the lifetime in here as well, and when you're using the trait the lifetime is usually always required. It's a bit of extra typing, but it beats the previous version.

```
trait Parser<'a, Output> {
    fn parse(&self, input: &'a str) -> ParseResult<'a, Output>;
}
```

It has just the one method, for now: the `parse()` method, which should look familiar: it's the same as the parser function we've been writing.

To make this even easier, we can actually implement this trait for any function that matches the signature of a parser:

```
impl<'a, F, Output> Parser<'a, Output> for F
where
    F: Fn(&'a str) -> ParseResult<Output>,
{
    fn parse(&self, input: &'a str) -> ParseResult<'a, Output> {
        self(input)
    }
}
```

This way, not only can we pass around the same functions we've been passing around so far as parsers fully implementing the `Parser` trait, we also open up the possibility to use other kinds of types as parsers.

But, more importantly, it saves us from having to type out those function signatures all the time. Let's rewrite the `map` function to see how it works out.

```
fn map<'a, P, F, A, B>(parser: P, map_fn: F) -> impl Parser<'a, B>
where
    P: Parser<'a, A>,
    F: Fn(A) -> B,
{
    move |input|
        parser.parse(input)
            .map(|(next_input, result)| (next_input, map_fn(result)))
}
```

One thing to note here in particular: instead of calling the parser as a function directly, we now have to go `parser.parse(input)`, because we don't know if the type `P` is a function, we just knows that it implements `Parser`, and so we have to stick with the interface `Parser` provides. Otherwise, the function body looks exactly the same, and the types look a lot tidier. There's the new lifetime `'a'` for some extra noise, but overall it's quite an improvement.

If we rewrite the `pair` function in the same way, it's even more tidy now:

```
fn pair<'a, P1, P2, R1, R2>(parser1: P1, parser2: P2) -> impl Parser<'a, (R1, R2)
where
    P1: Parser<'a, R1>,
    P2: Parser<'a, R2>,
{
    move |input| match parser1.parse(input) {
        Ok((next_input, result1)) => match parser2.parse(next_input) {
            Ok((final_input, result2)) => Ok((final_input, (result1, result2))),
            Err(err) => Err(err),
        },
        Err(err) => Err(err),
    }
}
```

Same thing here: the only changes are the tidied up type signatures and the need to go `parser.parse(input)` instead of `parser(input)`.

Actually, let's tidy up `pair`'s function body too, the same way we did with `map`.

```
fn pair<'a, P1, P2, R1, R2>(parser1: P1, parser2: P2) -> impl Parser<'a, (R1, R2)
where
    P1: Parser<'a, R1>,
    P2: Parser<'a, R2>,
{
    move |input| {
        parser1.parse(input).and_then(|(next_input, result1)| {
            parser2.parse(next_input)
                .map(|(last_input, result2)| (last_input, (result1, result2)))
        })
    }
}
```

The `and_then` method on `Result` is similar to `map`, except that the mapping function doesn't return the new value to go inside the `Result`, but a new `Result` altogether. The code above is identical in effect to the previous version written out with all those `match` blocks. We're going to get back to `and_then` later, but in the here and now, let's actually get those `left` and `right` combinators implemented, now that we have a nice and tidy `map`.

Left And Right

With `pair` and `map` in place, we can write `left` and `right` very succinctly:

```
fn left<'a, P1, P2, R1, R2>(parser1: P1, parser2: P2) -> impl Parser<'a, R1>
where
    P1: Parser<'a, R1>,
    P2: Parser<'a, R2>,
{
    map(pair(parser1, parser2), |(left, _right)| left)
}

fn right<'a, P1, P2, R1, R2>(parser1: P1, parser2: P2) -> impl Parser<'a, R2>
where
    P1: Parser<'a, R1>,
    P2: Parser<'a, R2>,
{
    map(pair(parser1, parser2), |(_left, right)| right)
}
```

We use the `pair` combinator to combine the two parsers into a parser for a tuple of their results, and then we use the `map` combinator to select just the part of the tuple we want to keep.

Rewriting our test for the first two pieces of our element tag, it's now just a little bit cleaner, and in the process we've gained some important new parser combinator powers.

We have to update our two parsers to use `Parser` and `ParseResult` first, though. `match_literal` is the more complicated one:

```
fn match_literal<'a>(expected: &'static str) -> impl Parser<'a, ()> {
    move |input: &'a str| match input.get(0..expected.len()) {
        Some(next) if next == expected => Ok((&input[expected.len()..], ())),
        _ => Err(input),
    }
}
```

In addition to changing the return type, we also have to make sure the input type on the closure is `&'a str`, or rustc gets upset.

For `identifier`, just change the return type and you're done, inference takes care of the lifetimes for you:

```
fn identifier(input: &str) -> ParseResult<String> {
```

And now the test, satisfyingly absent that ungainly `()` in the result.

```
#[test]
fn right_combinator() {
    let tag_opener = right(match_literal("<"), identifier);
    assert_eq!(
        Ok(("/>", "my-first-element".to_string())),
        tag_opener.parse("<my-first-element/>")
    );
    assert_eq!(Err("oops"), tag_opener.parse("oops"));
    assert_eq!(Err("!oops"), tag_opener.parse("<!oops"));
}
```

One Or More

Let's continue parsing that element tag. We've got the opening `<`, and we've got the identifier. What's next? That should be our first attribute pair.

No, actually, those attributes are optional. We're going to have to find a way to deal with things being optional.

No, wait, hold on, there's actually something we have to deal with even *before* we get as far as the first optional attribute pair: whitespace.

Between the end of the element name and the start of the first attribute name (if there is one), there's a space. We need to deal with that space.

It's even worse than that - we need to deal with *one or more spaces*, because `<element  attribute="value"/>` is valid syntax too, even if it's a bit over the top with the spaces. So this seems to be a good time to think about whether we could write a combinator that expresses the idea of *one or more* parsers.

We've dealt with this already in our `identifier` parser, but it was all done manually there. Not surprisingly, the code for the general idea isn't all that different.

```rust
 fn one_or_more<'a, P, A>(parser: P) -> impl Parser<'a, Vec<A>>
where
    P: Parser<'a, A>,
{
    move |mut input| {
        let mut result = Vec::new();

        if let Ok((next_input, first_item)) = parser.parse(input) {
            input = next_input;
            result.push(first_item);
        } else {
            return Err(input);
        }

        while let Ok((next_input, next_item)) = parser.parse(input) {
            input = next_input;
            result.push(next_item);
        }

        Ok((input, result))
    }
}
```

First of all, the return type of the parser we're building from is `A`, and the return type of the combined parser is `Vec<A>` - any number of `A`s.

The code does indeed look very similar to `identifier`. First, we parse the first element, and if it's not there, we return with an error. Then we parse as many more elements as we can, until the parser fails, at which point we return the vector with the elements we collected.

Looking at that code, how easy would it be to adapt it to the idea of *zero* or more? We just need to remove that first run of the parser:

```
fn zero_or_more<'a, P, A>(parser: P) -> impl Parser<'a, Vec<A>>
where
    P: Parser<'a, A>,
{
    move |mut input| {
        let mut result = Vec::new();

        while let Ok((next_input, next_item)) = parser.parse(input) {
            input = next_input;
            result.push(next_item);
        }

        Ok((input, result))
    }
}
```

Let's write some tests to make sure those two work.

```
#[test]
fn one_or_more_combinator() {
    let parser = one_or_more(match_literal("ha"));
    assert_eq!(Ok(("", vec![(), (), ()])), parser.parse("hahaha"));
    assert_eq!(Err("ahah"), parser.parse("ahah"));
    assert_eq!(Err(""), parser.parse(""));
}

#[test]
fn zero_or_more_combinator() {
    let parser = zero_or_more(match_literal("ha"));
    assert_eq!(Ok(("", vec![(), (), ()])), parser.parse("hahaha"));
    assert_eq!(Ok(("ahah", vec![])), parser.parse("ahah"));
    assert_eq!(Ok(("", vec![])), parser.parse(""));
}
```

Note the difference between the two: for `one_or_more`, finding an empty string is an error, because it needs to see at least one case of its sub-parser, but for `zero_or_more`, an empty string just means the zero case, which is not an error.

At this point, it's reasonable to start thinking about ways to generalise these two, because one is an exact copy of the other with just one bit removed. It might be tempting to express `one_or_more` in terms of `zero_or_more` with something like this:

```
fn one_or_more<'a, P, A>(parser: P) -> impl Parser<'a, Vec<A>>
where
    P: Parser<'a, A>,
{
    map(pair(parser, zero_or_more(parser)), |(head, mut tail)| {
        tail.insert(0, head);
        tail
    })
}
```

Here, we run into Rust Problems, and I don't even mean the problem of not having a `cons` method for `Vec`, but I know every Lisp programmer reading that bit of code was thinking it. No, it's worse than that: it's ownership.

We own that parser, so we can't go passing it as an argument twice, the compiler will start shouting at you that you're trying to move an already moved value. So can we make our combinators take references instead? No, it turns out, not without running into another whole set of borrow checker troubles - and we're not going to even go there right now. And because these parsers are functions, they don't do anything so straightforward as to implement `Clone`, which would have saved the day very tidily, so we're stuck with a constraint that we can't repeat our parsers easily in combinators.

That isn't necessarily a *big* problem, though. It means we can't express `one_or_more` using combinators, but it turns out those two are usually the only combinators you need anyway which tend to reuse parsers, and, if you wanted to get really fancy, you could write a combinator that takes a `RangeBound` in addition to a parser and repeats it according to a range: `range(0..)` for `zero_or_more`, `range(1..)` for `one_or_more`, `range(5..=6)` for exactly five or six, wherever your heart takes you.

Let's leave that as an exercise for the reader, though. Right now, we're going to be perfectly fine with just `zero_or_more` and `one_or_more`.

Another exercise might be to find a way around those ownership issues - maybe by wrapping a parser in an `Rc` to make it clonable?

A Predicate Combinator

We now have the building blocks we need to parse that whitespace with `one_or_more`, and to parse the attribute pairs with `zero_or_more`.

Actually, hold on a moment. We don't really want to parse the whitespace *then* parse the attributes. If you think about it, if there are no attributes, the whitespace is optional, and we could encounter an immediate `>` or `/>`. But if there's an attribute, there *must* be whitespace first. Lucky for us, there must also be whitespace between each attribute, if there are several, so what we're really looking at here is a sequence of *zero or more* occurrences of *one or more* whitespace items followed by the attribute.

We need a parser for a single item of whitespace first. We can go one of three ways here.

One, we can be silly and use our `match_literal` parser with a string containing just a single space. Why is that silly? Because whitespace is also line breaks, tabs and a whole number of strange Unicode characters which render as whitespace. We're going to have to lean on Rust's standard library again, and of course `char` has an `is_whitespace` method just like it had `is_alphabetic` and `is_alphanumeric`.

Two, we can just write out a parser which consumes any number of whitespace characters using the `is_whitespace` predicate much like we wrote our `identifier` earlier.

Three, we can be clever, and we do like being clever. We could write a parser `any_char` which returns a single `char` as long as there is one left in the input, and a combinator `pred` which takes a parser and a predicate function, and combine the two like this: `pred(any_char, |c| c.is_whitespace())`. This has the added bonus of making it really easy to write the final parser we're going to need too: the quoted string for the attribute values.

The `any_char` parser is straightforward as a parser, but we have to remember to be mindful of those UTF-8 gotchas.

```
fn any_char(input: &str) -> ParseResult<char> {
    match input.chars().next() {
        Some(next) => Ok((&input[next.len_utf8()..], next)),
        _ => Err(input),
    }
}
```

And the `pred` combinator also doesn't hold any surprises to our now seasoned eyes. We invoke the parser, then we call our predicate function on the value if the parser succeeded, and only if that returns true do we actually return a success, otherwise we return as much of an error as a failed parse would.

```
fn pred<'a, P, A, F>(parser: P, predicate: F) -> impl Parser<'a, A>
where
    P: Parser<'a, A>,
    F: Fn(&A) -> bool,
{
    move |input| {
        if let Ok((next_input, value)) = parser.parse(input) {
            if predicate(&value) {
                return Ok((next_input, value));
            }
        }
        Err(input)
    }
}
```

And a quick test to make sure everything is in order:

```
#[test]
fn predicate_combinator() {
    let parser = pred(any_char, |c| *c == 'o');
    assert_eq!(Ok(("mg", 'o')), parser.parse("omg"));
    assert_eq!(Err("lol"), parser.parse("lol"));
}
```

With these two in place, we can write our `whitespace_char` parser with a quick one-liner:

```
fn whitespace_char<'a>() -> impl Parser<'a, char> {
    pred(any_char, |c| c.is_whitespace())
}
```

And, now that we have `whitespace_char`, we can also express the idea we were heading towards, *one or more whitespace*, and its sister idea, *zero or more whitespace*. Let's indulge ourselves in some brevity and call them `space1` and `space0` respectively.

```
fn space1<'a>() -> impl Parser<'a, Vec<char>> {
    one_or_more(whitespace_char())
}

fn space0<'a>() -> impl Parser<'a, Vec<char>> {
    zero_or_more(whitespace_char())
}
```

Quoted Strings

With all that sorted, can we now, at last, parse those attributes? Yes, we just need to make sure we have all the individual parsers for the components of the attributes. We've got `identifier` already for the attribute name (though it's tempting to rewrite it using `any_char` and `pred` plus our `*_or_more` combinators). The `=` is just `match_literal("=")`. We're short one quoted string parser, though, so let's build that. Fortunately, we've already got all the combinators we need to do it.

```
fn quoted_string<'a>() -> impl Parser<'a, String> {
    map(
        right(
            match_literal("\""),
            left(
                zero_or_more(pred(any_char, |c| *c != '"')),
                match_literal("\""),
            ),
        ),
        |chars| chars.into_iter().collect(),
    )
}
```

The nesting of combinators is getting slightly annoying at this point, but we're going to resist refactoring everything to fix it just for now, and instead focus on what's going on here.

The outermost combinator is a `map`, because of the aforementioned annoying nesting, and it's a terrible place to start if we're going to understand this one, so let's try and find where it really starts: the first quote character. Inside the `map`, there's a `right`, and the first part of the `right` is the one we're looking for: the `match_literal("\"")`. That's our opening quote.

The second part of that `right` is the rest of the string. That's inside the `left`, and we quickly note that the *right* hand argument of that `left`, the one we ignore, is the other `match_literal("\"")` - the closing quote. So the left hand part is our quoted string.

We take advantage of our new `pred` and `any_char` here to get a parser that accepts *anything but another quote*, and we put that in `zero_or_more`, so that what we're saying is as follows:

- one quote
- followed by zero or more things that are *not* another quote
- followed by another quote

And, between the `right` and the `left`, we discard the quotes from the result value and get our quoted string back.

But wait, that's not a string. Remember what `zero_or_more` returns? A `Vec<A>` for the inner parser's return type `A`. For `any_char`, that's `char`. What we've got, then, is not a string but a `Vec<char>`. That's where the `map` comes in: we use it to turn a `Vec<char>` into a `String` by leveraging the fact that you can build a `String` from an `Iterator<Item = char>`, so we can just call `vec_of_chars.into_iter().collect()` and, thanks to the power of type inference, we have our `String`.

Let's just write a quick test to make sure that's all right before we go on, because if we needed that many words to explain it, it's probably not something we should leave to our faith in ourselves as programmers.

```
#[test]
fn quoted_string_parser() {
    assert_eq!(
        Ok(("", "Hello Joe!".to_string())),
        quoted_string().parse("\"Hello Joe!\"")
    );
}
```

So, now, finally, I swear, let's get those attributes parsed.

At Last, Parsing Attributes

We can now parse whitespace, identifiers, `=` signs and quoted strings. That, finally, is all we need for parsing attributes.

First, let's write a parser for an attribute pair. We're going to be storing them as a `Vec<(String, String)>`, as you may recall, so it feels like we'd need a parser for that `(String, String)` to feed to our trusty `zero_or_more` combinator. Let's see if we can build one.

```
fn attribute_pair<'a>() -> impl Parser<'a, (String, String)> {
    pair(identifier, right(match_literal("="), quoted_string()))
}
```

Without even breaking a sweat! To summarise: we already have a handy combinator for parsing a tuple of values, `pair`, so we use that with the `identifier` parser, yielding a `String`, and a `right` with the `=` sign, whose value we don't want to keep, and our fresh `quoted_string` parser, which gets us the other `String`.

Now, let's combine that with `zero_or_more` to build that vector - but let's not forget that whitespace in between them.

```
fn attributes<'a>() -> impl Parser<'a, Vec<(String, String)>> {
    zero_or_more(right(space1(), attribute_pair()))
}
```

Zero or more occurrences of the following: one or more whitespace characters, then an attribute pair. We use `right` to discard the whitespace and keep the attribute pair.

Let's test it.

```
#[test]
fn attribute_parser() {
    assert_eq!(
        Ok((
            "",
            vec![
                ("one".to_string(), "1".to_string()),
                ("two".to_string(), "2".to_string())
            ]
        )),
        attributes().parse(" one=\"1\" two=\"2\"")
    );
}
```

Tests are green! Ship it!

Actually, no, at this point in the narrative, my rustc was complaining that my types are getting terribly complicated, and that I need to increase the max allowed type size to carry on. It's a good chance you're getting the same error at this point, and if you are, you need to know how to deal with it. Fortunately, in these situations, rustc generally gives good advice, so when it tells you to add `#![type_length_limit = "…some big number…"]` to the top of your file, just do as it says. Actually, just go ahead and make it `#![type_length_limit = "16777216"]`, which is going to let us carry on a bit further into the stratosphere of complicated types. Full steam ahead, we're type astronauts now!

So Close Now

At this point, things seem like they're just about to start coming together, which is a bit of a relief, as our types are fast approaching NP-completeness. We just have the two versions of the element tag to deal with: the single element and the parent element with children, but we're feeling pretty confident that once we have those, parsing the children will be just a matter of `zero_or_more`, right?

So let's do the single element first, deferring the question of children for a little bit. Or, even better, let's first write a parser for everything the two have in common: the opening `<`, the element name, and the attributes. Let's see if we can get a result type of `(String, Vec<(String, String)>)` out of a couple of combinators.

```
fn element_start<'a>() -> impl Parser<'a, (String, Vec<(String, String)>)> {
    right(match_literal("<"), pair(identifier, attributes()))
}
```

With that in place, we can quickly tack the tag closer on it to make a parser for the single element.

```
fn single_element<'a>() -> impl Parser<'a, Element> {
    map(
        left(element_start(), match_literal("/>")),
        |(name, attributes)| Element {
            name,
            attributes,
            children: vec![],
        },
    )
}
```

Hooray, it feels like we're within reach of our goal - we're actually constructing an `Element` now!

Let's test this miracle of modern technology.

```
#[test]
fn single_element_parser() {
    assert_eq!(
        Ok((
            "",
            Element {
                name: "div".to_string(),
                attributes: vec![("class".to_string(), "float".to_string())],
                children: vec![]
            }
        )),
        single_element().parse("<div class=\"float\"/>")
    );
}
```

…and I think we just ran out of stratosphere.

The return type of `single_element` is so complicated that the compiler will grind away for a very long time until it runs into the very large type size limit we gave it earlier, asking for an even larger one. It's clear we can no longer ignore this problem, as it's a rather trivial parser and a compilation time of several minutes - maybe even several hours for the finished product - seems mildly unreasonsable.

Before proceeding, you'd better comment out those two functions and tests while we fix things…

To Infinity And Beyond

If you've ever tried writing a recursive type in Rust, you might already know the solution to our little problem.

A very simple example of a recursive type is a singly linked list. You can express it, in principle, as an enum like this:

```
enum List<A> {
    Cons(A, List<A>),
    Nil,
}
```

To which rustc will, very sensibly, object that your recursive type `List<A>` has an infinite size, because inside every `List::<A>::Cons` is another `List<A>`, and that means it's `List<A>`s all the way down into infinity. As far as rustc is concerned, we're asking for an infinite list, and we're asking it to be able to *allocate* an infinite list.

In many languages, an infinite list isn't a problem in principle for the type system, and it's actually not for Rust either. The problem is that in Rust, as mentioned, we need to be able to *allocate* it, or, rather, we need to be able to determine the *size* of a type up front when we construct it, and when the type is infinite, that means the size must be infinite too.

The solution is to employ a bit of indirection. Instead of our `List::Cons` being an element of `A` and another *list* of `A`, instead we make it an element of `A` and a *pointer* to a list of `A`. We know the size of a pointer, and it's the same no matter what it points to, and so our `List::Cons` now has a fixed and predictable size no matter the size of the list. And the way to turn an owned thing into a pointer to an owned thing on the heap, in Rust, is to `Box` it.

```
enum List<A> {
    Cons(A, Box<List<A>>),
    Nil,
}
```

Another interesting feature of `Box` is that the type inside it can be abstract. This means that instead of our by now incredibly complicated parser function types, we can let the type checker deal with a very succinct `Box<dyn Parser<'a, A>>` instead.

That sounds great. What's the downside? Well, we might be losing a cycle or two to having to follow that pointer, and it could be that the compiler loses some opportunities to optimise our parser. But recall Knuth's admonition about premature optimisation: it's going to be fine. You can afford those cycles. You're here to learn about parser combinators, not to learn about hand written hyperspecialised SIMD parsers (https://github.com/lemire/simdjson) (although they're exciting in their own right).

So let's proceed to implement `Parser` for a *boxed* parser function in addition to the bare functions we've been using so far.

```
 struct BoxedParser<'a, Output> {
    parser: Box<dyn Parser<'a, Output> + 'a>,
}

impl<'a, Output> BoxedParser<'a, Output> {
    fn new<P>(parser: P) -> Self
    where
        P: Parser<'a, Output> + 'a,
    {
        BoxedParser {
            parser: Box::new(parser),
        }
    }
}

impl<'a, Output> Parser<'a, Output> for BoxedParser<'a, Output> {
    fn parse(&self, input: &'a str) -> ParseResult<'a, Output> {
        self.parser.parse(input)
    }
}
```

We create a new type `BoxedParser` to hold our box, for the sake of propriety. To create a new `BoxedParser` from any other kind of parser (including another `BoxedParser`, even if that would be pointless), we provide a function `BoxedParser::new(parser)` which does nothing more than put that parser in a `Box` inside our new type. Finally, we implement `Parser` for it, so that it can be used interchangeably as a parser.

This leaves us with the ability to put a parser function in a `Box`, and the `BoxedParser` will work as a `Parser` just as well as the function. Now, as previously mentioned, that means moving the boxed parser to the heap and having to deref a pointer to get to it, which can cost us *several precious nanoseconds*, so we might actually want to hold off on boxing *everything*. It's enough to just box some of the more popular combinators.

An Opportunity Presents Itself

But, just a moment, this presents us with an opportunity to fix another thing that's starting to become a bit of a bother.

Remember the last couple of parsers we wrote? Because our combinators are standalone functions, when we nest a nontrivial number of them, our code starts getting a little bit unreadable. Recall our `quoted_string` parser:

```
fn quoted_string<'a>() -> impl Parser<'a, String> {
    map(
        right(
            match_literal("\""),
            left(
                zero_or_more(pred(any_char, |c| *c != '"')),
                match_literal("\""),
            ),
        ),
        |chars| chars.into_iter().collect(),
    )
}
```

It would read a lot better if we could make those combinators methods on the parser instead of standalone functions. What if we could declare our combinators as methods on the `Parser` trait?

The problem is that if we do that, we lose the ability to lean on `impl Trait` for our return types, because `impl Trait` isn't allowed in trait declarations.

…but now we have `BoxedParser`. We can't declare a trait method that returns `impl Parser<'a, A>`, but we most certainly *can* declare one that returns `BoxedParser<'a, A>`.

The best part is that we can even declare these with default implementations, so that we don't have to reimplement every combinator for every type that implements `Parser`.

Let's try it out with `map`, by extending our `Parser` trait as follows:

```
trait Parser<'a, Output> {
    fn parse(&self, input: &'a str) -> ParseResult<'a, Output>;

    fn map<F, NewOutput>(self, map_fn: F) -> BoxedParser<'a, NewOutput>
    where
        Self: Sized + 'a,
        Output: 'a,
        NewOutput: 'a,
        F: Fn(Output) -> NewOutput + 'a,
    {
        BoxedParser::new(map(self, map_fn))
    }
}
```

That's a lot of `'a`s, but, alas, they're all necessary. Luckily, we can still reuse our old combinator functions unchanged - and, as an added bonus, not only do we get a nicer syntax for applying them, we also get rid of the explosive `impl Trait` types by boxing them up automatically.

Now we can improve our `quoted_string` parser slightly:

```
fn quoted_string<'a>() -> impl Parser<'a, String> {
    right(
        match_literal("\""),
        left(
            zero_or_more(pred(any_char, |c| *c != '"')),
            match_literal("\""),
        ),
    )
    .map(|chars| chars.into_iter().collect())
}
```

It's now more obvious at first glance that the `.map()` is being called on the result of the `right()`.

We could also give `pair`, `left` and `right` the same treatment, but in the case of these three, I think it reads easier when they're functions, because they mirror the structure of `pair`'s output type. If you disagree, it's entirely possible to add them to the trait just like we did with `map`, and you're very welcome to go ahead and try it out as an exercise.

Another prime candidate, though, is `pred`. Let's add a definition for it to the `Parser` trait:

```
fn pred<F>(self, pred_fn: F) -> BoxedParser<'a, Output>
where
    Self: Sized + 'a,
    Output: 'a,
    F: Fn(&Output) -> bool + 'a,
{
    BoxedParser::new(pred(self, pred_fn))
}
```

This lets us rewrite the line in `quoted_string` with the `pred` call like this:

```
zero_or_more(any_char.pred(|c| *c != '"')),
```

I think that reads a little nicer, and I think we'll leave the `zero_or_more` as it is too - it reads like "zero or more of `any_char` with the following predicate applied," and that sounds about right to me. Once again, you can also go ahead and move `zero_or_more` and `one_or_more` into the trait if you prefer to go all in.

In addition to rewriting `quoted_string`, let's also fix up the `map` in `single_element`:

```rust
fn single_element<'a>() -> impl Parser<'a, Element> {
    left(element_start(), match_literal("/>")).map(|(name, attributes)| Element {
        name,
        attributes,
        children: vec![],
    })
}
```

Let's try and uncomment back `element_start` and the tests we commented out earlier and see if things got better. Get that code back in the game and try running the tests…

…and, yep, compilation time is back to normal now. You can even go ahead and remove that type size setting at the top of your file, you're not going to need it any more.

And that was just from boxing two `map`s and a `pred` - *and* we got a nicer syntax out of it!

Having Children

Now let's write the parser for the opening tag for a parent element. It's almost identical to `single_element`, except it ends in a `>` rather than a `/>`. It's also followed by zero or more children and a closing tag, but first we need to parse the actual opening tag, so let's get that done.

```rust
fn open_element<'a>() -> impl Parser<'a, Element> {
    left(element_start(), match_literal(">")).map(|(name, attributes)| Element {
        name,
        attributes,
        children: vec![],
    })
}
```

Now, how do we get those children? They're going to be either single elements or parent elements themselves, and there are zero or more of them, so we have our trusty `zero_or_more` combinator, but what do we feed it? One thing we haven't dealt with yet is a multiple choice parser: something that parses *either* a single element *or* a parent element.

To get there, we need a combinator which tries two parsers in order: if the first parser succeeds, we're done, we return its result and that's it. If it fails, instead of returning an error, we try the second parser *on the same input*. If that succeeds, great, and if it doesn't, we return the error too, as that means both our parsers have failed, and that's an overall failure.

```
fn either<'a, P1, P2, A>(parser1: P1, parser2: P2) -> impl Parser<'a, A>
where
    P1: Parser<'a, A>,
    P2: Parser<'a, A>,
{
    move |input| match parser1.parse(input) {
        ok @ Ok(_) => ok,
        Err(_) => parser2.parse(input),
    }
}
```

This allows us to declare a parser `element` which matches either a single element or a parent element (and, for now, let's just use `open_element` to represent it, and we'll deal with the children once we have `element` down).

```
fn element<'a>() -> impl Parser<'a, Element> {
    either(single_element(), open_element())
}
```

Now let's add a parser for the closing tag. It has the interesting property of having to match the opening tag, which means the parser has to know what the name of the opening tag is. But that's what function arguments are for, yes?

```
fn close_element<'a>(expected_name: String) -> impl Parser<'a, String> {
    right(match_literal("</"), left(identifier, match_literal(">")))
        .pred(move |name| name == &expected_name)
}
```

That `pred` combinator is proving really useful, isn't it?

And now, let's put it all together for the full parent element parser, children and all:

```
fn parent_element<'a>() -> impl Parser<'a, Element> {
    pair(
        open_element(),
        left(zero_or_more(element()), close_element(…oops)),
    )
}
```

Oops. How do we pass that argument to `close_element` now? I think we're short one final combinator.

We're so close now. Once we've solved this one last problem to get `parent_element` working, we should be able to replace the `open_element` placeholder in the `element` parser with our new `parent_element`, and that's it, we have a fully working XML parser.

Remember I said we'd get back to `and_then` later? Well, later is here. The combinator we need is, in fact, `and_then`: we need something that takes a parser, and a function that takes the result of a parser and returns a *new* parser, which we'll then run. It's a bit like `pair`, except instead of just collecting both results in a tuple, we thread them through a function. It's also just how `and_then` works with `Result`s and `Option`s, except it's a bit easier to follow because `Result`s and `Option`s don't really *do* anything, they're just things that hold some data (or not, as the case may be).

So let's try writing an implementation for it.

```
fn and_then<'a, P, F, A, B, NextP>(parser: P, f: F) -> impl Parser<'a, B>
where
    P: Parser<'a, A>,
    NextP: Parser<'a, B>,
    F: Fn(A) -> NextP,
{
    move |input| match parser.parse(input) {
        Ok((next_input, result)) => f(result).parse(next_input),
        Err(err) => Err(err),
    }
}
```

Looking at the types, there are a lot of type variables, but we know `P`, our input parser, which has a result type of `A`. Our function `F`, however, where `map` had a function from `A` to `B`, the crucial difference is that `and_then` takes a function from `A` to *a new parser* `NextP`, which has a result type of `B`. The final result type is `B`, so we can assume that whatever comes out of our `NextP` will be the final result.

The code is a bit less complicated: we start by running our input parser, and if it fails, it fails and we're done, but if if succeeds, now we call our function `f` on the result (of type `A`), and what comes out of `f(result)` is a new parser, with a result type of `B`. We run *this* parser on the next bit of input, and we return the result directly. If it fails, it fails there, and if it succeeds, we have our value of type `B`.

One more time: we run our parser of type `P` first, and if it succeeds, we call the function `f` with the result of parser `P` to get our next parser of type `NextP`, which we then run, and that's the final result.

Let's also add it straight away to the `Parser` trait, because this one, like `map`, is definitely going to read better that way.

```
 fn and_then<F, NextParser, NewOutput>(self, f: F) -> BoxedParser<'a, NewOutput>
where
    Self: Sized + 'a,
    Output: 'a,
    NewOutput: 'a,
    NextParser: Parser<'a, NewOutput> + 'a,
    F: Fn(Output) -> NextParser + 'a,
{
    BoxedParser::new(and_then(self, f))
}
```

OK, now, what's it good for?

First of all, we can *almost* implement `pair` using it:

```
 fn pair<'a, P1, P2, R1, R2>(parser1: P1, parser2: P2) -> impl Parser<'a, (R1, R2)>
where
    P1: Parser<'a, R1> + 'a,
    P2: Parser<'a, R2> + 'a,
    R1: 'a + Clone,
    R2: 'a,
{
    parser1.and_then(move |result1| parser2.map(move |result2| (result1.clone(), r
}
```

It looks very neat, but there's a problem: `parser2.map()` consumes `parser2` to create the wrapped parser, and the function is a `Fn`, not a `FnOnce`, so it's not allowed to consume `parser2`, just take a reference to it. Rust Problems, in other words. In a higher level language where these things aren't an issue, this would have been a really neat way to define `pair`.

What we can do with it even in Rust, though, is use that function to lazily generate the right version of our `close_element` parser, or, in other words, we can get it to pass that argument into it.

Recalling our failed attempt:

```
fn parent_element<'a>() -> impl Parser<'a, Element> {
    pair(
        open_element(),
        left(zero_or_more(element()), close_element(…oops)),
    )
}
```

Using `and_then`, we can now get this right by using that function to build the right version of `close_element` on the spot.

```
fn parent_element<'a>() -> impl Parser<'a, Element> {
    open_element().and_then(|el| {
        left(zero_or_more(element()), close_element(el.name.clone())).map(move |ch
            let mut el = el.clone();
            el.children = children;
            el
        })
    })
}
```

It looks a bit more complicated now, because the `and_then` has to go on `open_element()`, where we find out the name that goes into `close_element`. This means that the rest of the parser after `open_element` all has to be constructed inside the `and_then` closure. Moreover, because that closure is now the sole recipient of the `Element` result from `open_element`, the parser we return also has to carry that info forward.

The inner closure, which we `map` over the generated parser, has a reference to the `Element` (`el`) from the outer closure. We have to `clone()` it because we're in a `Fn` and thus only have a reference to it. We take the result of the inner parser (our `Vec<Element>` of children) and add that to our cloned `Element`, and we return that as our final result.

All we need to do now is go back to our `element` parser and make sure we change `open_element` to `parent_element`, so it parses the whole element structure instead of just the start of it, and I believe we're done!

Are You Going To Say The M Word Or Do I Have To?

Remember we talked about how the `map` pattern is called a "functor" on Planet Haskell?

The `and_then` pattern is another thing you see a lot in Rust, in generally the same places as `map`. It's called `flat_map` (https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.flat_map) on `Iterator`, but it's the same pattern as the rest.

The fancy word for it is "monad." If you've got a thing `Thing<A>`, and you have an `and_then` function available that you can pass a function from `A` to `Thing<B>` into, so that now you have a new `Thing<B>` instead, that's a monad.

The function might get called instantly, like when you have an `Option<A>`, we already know if it's a `Some(A)` or a `None`, so we apply the function directly if it's a `Some(A)`, giving us a `Some(B)`.

It might also be called lazily. For instance, if you have a `Future<A>` that is still waiting to resolve, instead of `and_then` immediately calling the function to create a `Future<B>`, instead it creates a new `Future<B>` which contains both the `Future<A>` and the function, and which then waits for `Future<A>` to finish. When it does, it calls the function with the result of the `Future<A>`, and Bob's your uncle[1], you get your `Future<B>` back. In other words, in the case of a `Future` you can think of the function you pass to `and_then` as a *callback function*, because it gets called with the result of the original future when it completes. It's also a little more interesting than that, because it returns a *new* `Future`, which may or may not have already been resolved, so it's a way to *chain* futures together.

As with functors, though, Rust's type system isn't currently capable of expressing monads, so let's only note that this pattern is called a monad, and that, rather disappointingly, it's got nothing at all to do with burritos, contrary to what they say on the internets, and move on.

Whitespace, Redux

Just one last thing.

We should have a parser capable of parsing some XML now, but it's not very appreciative of whitespace. Arbitrary whitespace should be allowed between tags, so that we're free to insert line breaks and such between our tags (and whitespace should in principle be allowed between identifiers and literals, like `< div / >`, but let's skip that).

We should be able to put together a quick combinator for that with no effort at this point.

```
 fn whitespace_wrap<'a, P, A>(parser: P) -> impl Parser<'a, A>
where
    P: Parser<'a, A>,
{
    right(space0(), left(parser, space0()))
}
```

If we wrap `element` in that, it will ignore all leading and trailing whitespace around `element`, which means we're free to use as many line breaks and as much indentation as we like.

```
 fn element<'a>() -> impl Parser<'a, Element> {
    whitespace_wrap(either(single_element(), parent_element()))
}
```

We're Finally There!

I think we did it! Let's write an integration test to celebrate.

```
#[test]
fn xml_parser() {
    let doc = r#"
        <top label="Top">
            <semi-bottom label="Bottom"/>
            <middle>
                <bottom label="Another bottom"/>
            </middle>
        </top>"#;
    let parsed_doc = Element {
        name: "top".to_string(),
        attributes: vec![("label".to_string(), "Top".to_string())],
        children: vec![
            Element {
                name: "semi-bottom".to_string(),
                attributes: vec![("label".to_string(), "Bottom".to_string())],
                children: vec![],
            },
            Element {
                name: "middle".to_string(),
                attributes: vec![],
                children: vec![Element {
                    name: "bottom".to_string(),
                    attributes: vec![("label".to_string(), "Another bottom".to_str
                    children: vec![],
                }],
            },
        ],
    };
    assert_eq!(Ok(("", parsed_doc)), element().parse(doc));
}
```

And one that fails because of a mismatched closing tag, just to make sure we got that bit right:

```
#[test]
fn mismatched_closing_tag() {
    let doc = r#"
        <top>
            <bottom/>
        </middle>"#;
    assert_eq!(Err("</middle>"), element().parse(doc));
}
```

The good news is that it returns the mismatched closing tag as the error. The bad news is that it doesn't actually *say* that the problem is a mismatched closing tag, just *where* the error is. It's better than nothing, but, honestly, as error messages go, it's still terrible. But turning this into a thing that actually gives good errors is the topic for another, and probably at least as long, article.

Let's focus on the good news: we wrote a parser from scratch using parser combinators! We know that a parser forms both a functor and a monad, so you can now impress people at parties with your daunting knowledge of category theory[2].

Most importantly, we now know how parser combinators work from the ground up. Nobody can stop us now!

Victory Puppies

Further Resources

First of all, I'm guilty of explaining monads to you in strictly Rusty terms, and I know that Phil Wadler would be very upset with me if I didn't point you towards his seminal paper (https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf) which goes into much more exciting detail about them - including how they relate to parser combinators.

The ideas in this article are extremely similar to the ideas behind the `pom` (https://crates.io/crates/pom) parser combinator library, and if this has made you want to work with parser combinators in the same style, I can highly recommend it.

The state of the art in Rust parser combinators is still `nom` (https://crates.io/crates/nom), to the extent that the aforementioned `pom` is clearly derivatively named (and there's no higher praise than that), but it takes a very different approach from what we've built here today.

Another popular parser combinator library for Rust is `combine` (https://crates.io/crates/combine), which may be worth a look as well.

The seminal parser combinator library for Haskell is Parsec (http://hackage.haskell.org/package/parsec).

Finally, I owe my first awareness of parser combinators to the book *Programming in Haskell* (http://www.cs.nott.ac.uk/~pszgmh/pih.html) by Graham Hutton, which is a great read and has the positive side effect of also teaching you Haskell.

Licence

Footnotes

1: He isn't really your uncle.

2: Please don't be that person at parties.
