Parsing Text with Nom

"Parsing" is turning a stream of raw text or binary into some structured data types, i.e. a Rust type that your code can understand and use. This isn't the textbook definition of parsing, but damnit, this is my blog and my opinion. This tutorial is about *nom*, my favourite Rust parsing library. It uses a *parser combinator* approach: you start writing tiny parsers that match, say, a single number or a character. These become building blocks for larger parsers, that match, say, a date or a phone number. By combining many small parsers together, you can build a big parser that decodes a file or stream into nice Rust structs and enums. In this tutorial we'll use Nom to parse the input file to an Advent of Code puzzle.

What are parsers?

Most programs need to read some input before they know what to do. A video game will move a character based on keyboard input, the `cat` command will display different text based on its arguments, and accounting software will output a very good or a very worrying message depending on the spreadsheet it loads.

Program input can come in many different forms. Sometimes the input is already structured, for example, most video game programming environments can recognize when a mouse button is pressed, know which mouse button it was, and call the appropriate code to react to that. But sometimes input is unstructured -- it can't be easily understood by your program. For example, the input could be a text file. Your programming languages probably understands that it's been given a text file. But it doesn't understand what the text *means* or what it should do in response. So programmers use parsers to take this unstructured file and examine it, match it against some logical rules. Once the data has been structured, you can call the appropriate functions to handle it.

For example, in Advent of Code day 5 you're given a list of lines and must find points where two or more lines overlap. The Advent of Code website gives the puzzle input in a text file like this:

```
0,9 -> 5,9
8,0 -> 0,8
9,4 -> 3,4
2,2 -> 2,1
7,0 -> 7,4
6,4 -> 2,0
0,9 -> 2,9
3,4 -> 1,4
0,0 -> 8,8
5,5 -> 8,2
```

Rust doesn't natively know to interpret this file as a list of lines in 2D space. To the Rust compiler, this text file is just a big String. We have to write a *parser* which takes that string and parses it into types Rust can understand, like

```
/// A point in 2D space
pub struct Point {
    pub x: u32,
    pub y: u32,
}

/// A line spanning two points.
pub struct Line(pub Point, pub Point);
```

If you already know the basics of Nom, or you've used parser combinators in other languages, feel free to skip ahead, but for now, let's take a closer look at Nom.

Building parsers with Nom

Nom is a *parser combinator* library. In general, a *combinator* framework gives you

1. A set of tiny primitive tools for doing small, simple tasks.

2. Ways to combine those tools to accomplish more complex tasks. These are called combinators.

For example, you can think of the Bash shell as a combinator approach. You have

1. A set of primitive tools like ls, cd, cat and grep that do one thing, and do it well.
2. Ways to combine tools, for example |, &&, loops and `if` statements.

Another example of the combinator pattern is Rust's Result<T,E> type. Your primitive tools might be closures like `Fn(T) -> Result<V, U>`. The combinators are methods like `map`, `and_then` and `or_else`.

And yet another example (maybe a bit of a stretch) is JSON! JSON defines some primitive types like `number`, `bool` and `string`, as well as some combinators like `Object` or `Array` that can take multiple primitive values and combine them into a composite value. Like a Person data type made from two primitive data types: `{"name": string, "age": number}`. And you can then combine Person values into arrays of people, or objects where the fields are people.

Nom takes a similar approach, using combinators to combine parsers[1]. There are primitive parsers that match one simple value from the input string, and functions (called combinators) which let you combine multiple primitive parsers into a more complex (composite) parser. Let's look at the primitives first.

Primitive parsers

A parser's job is to consume bytes from some *input*, and try converting them into some *output* Rust value. If this conversion succeeds, the parser should return two things:

1. The **remaining input** which wasn't parsed, so that you can continue running other parsers on the rest of the input file
2. The parsed **output** value

If the conversion failed, the parser should return some helpful error.

In Nom, parsers are just functions. Each parser function has three generic types:

- `I` is for *input*. In this tutorial, our input will always be a string, but you can use Nom to parse binary (`Vec<u8>` or `Bytes`) or your own custom type, for example tokens output by a lexer.
- `O` is for *output*. The parser reads some bytes from the input and tries to convert them to its output value. Some parsers will output a number, a string, or some "business logic" type like a Person or User struct. In our Advent of Code above, we'll write parsers that output Points and Lines.
- `E` is for *error* (just like in the stdlib `Result<T,E>`). It's generic because Nom supports different error types with different trade-offs. We'll discuss them later.

Each function's type signature is basically the same: `Fn(I) -> Result<(I, O), E>`. It's just a function that takes the input string, reads some bytes, and tries to convert them into an output value `O`. If that conversion succeeds, it returns the rest of the input (the remainder, part the parser didn't need to read to get a value) and the value, which is why the OK branch is `(I, O)`. If it fails, it just returns the error `E`.

Here's a very simple example parser: `digit1`. Technically it's `nom::character::complete::digit1`[2]. The nom docs are usually really good, so let's take a look at their example.

```rust
use nom::character::complete::newline;

fn parser(input: &str) -> IResult<&str, &str> {
    digit1(input)
}

assert_eq!(parser("21c"), Ok(("c", "21")));
assert_eq!(parser("c1"), Err(Err::Error(Error::new("c1", ErrorKind::Digi
assert_eq!(parser(""), Err(Err::Error(Error::new("", ErrorKind::Digit)))
```

So, we see that `digit1` is just a function. It takes one argument, the input string.

The parser reads characters from the start of the input string, and if the character is a digit, it adds it to the output. When it finds a non-digit character, it terminates. Once it terminates it returns an IResult... what's that?

Remember above, we said that parsers should return `(I, O)` if they succeed (the remaining input string that *wasn't* consumed to get an output value, and the output value itself), and `E` if they fail. Well, `IResult` is just shorthand for this. It's a simple convenience newtype: `type IResult<I, O, E> = Result<(I, O), E>`. The docs explain it well:

> Holds the result of parsing functions. The `Ok` side is a pair containing the remainder of the input (the part of the data that was not parsed) and the produced value. The `Err` side contains an instance of `nom::Err`.

As expected, if the input string *doesn't* start with any digits, Nom returns an error. The example uses Nom's default Error type, which tells you two things:

1. Where in the input string Nom failed.
2. What Nom was trying to parse when it failed.

If you want more detail, you can use E = `VerboseError` instead of the default error type. This has a bit more overhead at runtime, but makes debugging easier.

Let's look at another parser, named `tag`. It matches some specific substring chosen by the programmer. The Nom docs have an example, using `tag` to match the substring "Hello":

```
use nom::bytes::complete::tag;

fn parser(s: &str) -> IResult<&str, &str> {
  tag("Hello")(s)
}

assert_eq!(parser("Hello, World!"), Ok((", World!", "Hello")));
assert_eq!(parser("Something"), Err(Err::Error(Error::new("Something", E
assert_eq!(parser(""), Err(Err::Error(Error::new("", ErrorKind::Tag))));
```

Notice the `tag` function has a different type signature to the `digit1` function! That's because `digit1` is a parser -- it takes an input string and outputs an `IResult`, defined as "the result of parsing functions". But `tag` isn't exactly a parser -- it's a function which *outputs* a parser. It takes one argument, the substring to match on, and returns a parser which matches that string.

- `digit` is a parser which takes one argument (an input string) and returns `IResult`.
- `tag` takes one argument, a string to match, and returns a parser. That parser takes one argument (an input string) and returns `IResult`.

Nom's other primitive parsers fall into these categories: they're either actually parsers, or functions that return a parser. For example:

- `digit0`, `alpha1` and `line_ending` are parsers
- `char` takes a character and returns a parser that matches that character.

OK, so we can now build small parsers that recognize particular characters. But that's not going to be enough to parse the Advent of Code input file. We need a parser that can parse the text "(3,4)" into `Point{x: 3, y: 4}`. To do that, we need to combine these small parsers into a composite parser that can convert text into a Point.

Combining parsers with... combinators

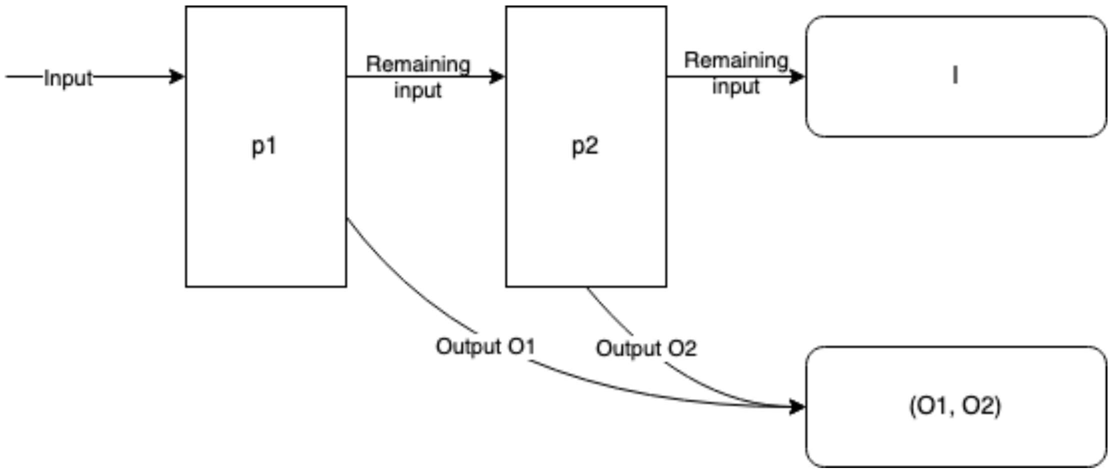
So far we've seen two types of Nom functions:

1. Parsers, e.g. `digit1`
2. Functions that return a parser, e.g. `tag`

We're about to examine a third kind:

3. Functions that take a parser as an argument, and return a new parser, e.g. `pair`.

The `pair` function takes two parsers as arguments (call them `p1` and `p2`). It returns a new parser that chains them together. Each parser takes its input type I (usually string) and outputs (I, O) for the remaining input and the output value it parsed:



So basically it:

1. Runs `p1`
2. If `p1` failed, return the error.
3. Otherwise, the remaining input from `p1` becomes the input for `p2`
4. Run `p2`

As usual, the official docs have a good example:

```
use nom::sequence::pair;
use nom::bytes::complete::tag;

// `pair` gets an object from the first parser, then gets another object
let mut parser = pair(tag("abc"), tag("efg"));

assert_eq!(parser("abcefg"), Ok(("", ("abc", "efg"))));
assert_eq!(parser("abcefghij"), Ok(("hij", ("abc", "efg"))));
assert_eq!(parser(""), Err(Err::Error(("", ErrorKind::Tag))));
assert_eq!(parser("123"), Err(Err::Error(("123", ErrorKind::Tag))));
```

So, that's a very simple parser combinator which takes two parsers and outputs a new parser. Nom is full of these helpful little functions. For example, there's `separated_pair` which takes *three* parsers: it runs all three in sequence and discards the middle parser's output. It's useful for removing separators, e.g.

parsing "1,2" into "1" and "2".

```
fn main() {
    use nom::{character::complete::{char, digit1}, sequence::separated_p

    let mut parser = separated_pair(digit1, char(','), digit1);
    assert_eq!(parser("1,2"), Ok(("", ("1", "2"))));
}
```

So far we've looked at combinators that take multiple parsers as arguments. But here's one that only takes a single parser: it's called `map`. It wraps a parser, then passes the output into a closure.

```
use nom::{
    character::complete::{alpha1, char},
    combinator::map,
    sequence::separated_pair,
    IResult,
};

/// Some example business logic type that our program uses
#[derive(PartialEq, Eq, Debug)]
struct Person {
    first_name: String,
    last_name: String,
}

impl Person {
    fn parse(input: &str) -> IResult<&str, Self> {
        // Parses two words, separated by a space.
        // Note: alpha1 recognizes one or more letters from A to Z.
        let two_words_parser = separated_pair(alpha1, char(' '), alpha1)

        // Defines a new parser which wraps the `two_words_parser`, then
        // passes the resulting pair into a closure.
        let mut person_parser = map(
            two_words_parser,
            |(first_name, last_name)| Self {
                first_name: String::from(first_name),
                last_name: String::from(last_name),
            },
        );

        // Use the parser
        person_parser(input)
    }
}

fn main() {
    let (remaining_input, actual) = Person::parse("Adam Chalmers_").unwr
    let expected = Person {
        first_name: "Adam".to_owned(),
        last_name: "Chalmers".to_owned(),
    };
    assert_eq!(remaining_input, "_");
    assert_eq!(expected, actual);
}
```

The `map` combinator expects a closure that always succeeds. But sometimes you want a closure that might fail. For example, you might have a parser that outputs a string, and then try to convert that string into a number. If the closure returns a Result, we can use `map_res`.

```
use nom::{character::complete::digit1, combinator::map_res, IResult};
use std::str::FromStr;

/// Parse a number from the string, but return it as an actual Rust numb
pub fn parse_numbers(input: &str) -> IResult<&str, u32> {
    let mut parser = map_res(digit1, u32::from_str);
    parser(input)
}

fn main() {
    let (actual_input_remaining, actual_output) = parse_numbers("123").u
```

```
        let expected_output = 1234;
        let expected_input_remaining = "";
        assert_eq!(actual_output, expected_output);
        assert_eq!(actual_input_remaining, expected_input_remaining);
    }
```

There are many other combinators in Nom's modules:

- `nom::sequence` contains the `pair` and `separated_pair` combinators we've already seen. Others include `tuple` which takes any number of different parsers and runs them. `pair` is a special case of `tuple` where the tuple has length 2.
- `nom::multi` has combinators that take a parser and run it many times, e.g. `many0` wraps another parser and runs it 0 or more times, or `separated_list1` which can parse `"1,2,3"` into `vec![1,2,3]`.
- `nom::branch` has combinators like `alt` for trying one of several different parsers (e.g. parsing either "3" or 0003 into the numeric value 3)
- `nom::combinator` has general-purpose combinators e.g. `map_opt` (like `map_res` but the closure returns Option, not Result)

OK! That's the basics of Nom: parsers consume bytes from an input (often a string) and output a Rust value. There are primitive parsers that match simple patterns, like some alphanumeric characters or whitespace. Then there are combinators, which take one or more parsers and produce more complex ones. Those parsers can then become the input for yet more combinators to make even more complex parsers. If you keep applying this process, you can parse some pretty complicated formats, like HTTP or EXIF. Let's take a look at a real-world example I used Nom for: parsing Advent of Code, day 5.

Solving problems with Nom

Above we described the problem: read a text file and parse it into `Line` objects. The key to parsing with Nom is to **start small**. Find a small part of the input that you *do* know how to parse, and unit test it to make sure it works. Then build up more complex parsers from there.

Obviously, we'll need to know how to parse a single 2D point.

```
use nom::{
    character::complete::{char, digit1},
    combinator::{map, map_res},
    sequence::separated_pair,
    IResult,
};
use std::str::FromStr;

/// Parse a `u32` from the start of the input string.
/// (this is copied/pasted from earlier in the blog post!)
pub fn parse_numbers(input: &str) -> IResult<&str, u32> {
    map_res(digit1, u32::from_str)(input)
}

/// A point in 2D space
#[derive(Debug, Eq, PartialEq)]
pub struct Point {
    pub x: u32,
    pub y: u32,
}

impl Point {
    /// Parse a point.
    /// Like all Nom parsers, this has an Input type, Output type and Er
    /// Its Input type is &str (because we're parsing a text file that g
    /// Its Output type is Point (which we can refer to as Self inside a
    /// Its Error type will be the default Nom error type, so we don't h
    /// specify it here.
    fn parse(input: &str) -> IResult<&str, Self> {

        // This parser outputs a (u32, u32).
        // It uses the `parse_numbers` parser
        // and the `separated_pair` combinator discussed earlier.
        let parse_two_numbers = separated_pair(parse_numbers, char(','),

        // Map the (u32, u32) into a Point.
        // Uses the `parse_two_numbers` parser defined on the previous l
        // and the `map` combinator discussed earlier.
        map(parse_two_numbers, |(x, y)| Point { x, y })(input)
    }
}

// Because each parser is just a pure, deterministic function,
// it's very easy to unit test them!

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_parse_point() {
        let tests = [
            ("1,2", Point { x: 1, y: 2 }, ""),
            ("1,2asdf", Point { x: 1, y: 2 }, "asdf"),
        ];
        for (input, expected_output, expected_remaining_input) in tests
            let (remaining_input, output) = Point::parse(input).unwrap()
            assert_eq!(output, expected_output);
            assert_eq!(remaining_input, expected_remaining_input);
        }
    }
}
```

OK, great. Now we can reuse that Point parser for a Line parser. The Advent of Code text file stores lines like this:

> 0,9 -> 5,9

```
/// A line spanning two points.
#[derive(Debug, Eq, PartialEq)]
pub struct Line(pub Point, pub Point);

impl Line {
    /// Parse a Line from the input string.
    fn parse(input: &str) -> IResult<&str, Self> {
        let parse_arrow = tag(" -> ");

        // Parse two points, separated by an arrow
        let parse_points = separated_pair(Point::parse, parse_arrow, Poi

        // If the parse succeeded, put those two points into a Line
        map(parse_points, |(p0, p1)| Line(p0, p1))(input)
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_parse_line() {
        let tests = [
            (
                "0,9 -> 5,9",
                Line(Point { x: 0, y: 9 }, Point { x: 5, y: 9 }),
                "",
            ),
            (
                "0,9 -> 5,9xyz",
                Line(Point { x: 0, y: 9 }, Point { x: 5, y: 9 }),
                "xyz",
            ),
        ];
        for (input, expected_output, expected_remaining_input) in tests
            let (remaining_input, output) = Line::parse(input).unwrap();
            assert_eq!(remaining_input, expected_remaining_input);
            assert_eq!(output, expected_output);
        }
    }
}
```

Now it should be easy to parse the rest. We know each line in the text file should encode one Line object. We can use separated_list1 to match a line, then match and discard a newline, then match a line, then match and discard a newline, etc etc until the end of the file.

```
/// Parse the whole Advent of Code day 5 text file.
pub fn parse_input(s: &str) -> Vec<Line> {
    let (remaining_input, lines) = separated_list1(line_ending, Line::pa
    assert!(remaining_input.is_empty());
    lines
}


#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_parse_file() {
        // Assuming you're logged in, you can download your Day 5 puzzle
        // https://adventofcode.com/2021/day/5/input
        // Then save it under src/data/input.txt
        // The `include_str!` macro reads a file into a String at compil
        // I use it for all my AoC problems!
        let input = include_str!("data/input.txt");
        let lines = parse_input(input);
        assert_eq!(lines.len(), 500);
    }
```

Now we can parse the entire text file! You can see the full code on GitHub, feel free to open PRs if you've got any suggestions.

I find the "start small" approach really productive. Using many small parsers instead of one big parser makes it very easy to understand and test each little step. I can choose any parser here and tell exactly what it does just by looking at the type signature. If I want to know the exact kind of text it parses, I can check the unit tests. This makes it easy to understand the code even years after

I last touched it. I should know, because a few years ago I wrote the fountain crate which parses the Fountain markdown language in Nom, and I still find the code pretty readable[3].

Next steps

I hope this helped you understand Nom and parser combinators! If you want to learn more, I suggest taking a look at Nom's README.md -- it has a list of links to different docs, including the API docs, some design documents and even a "which combinator should I use" guide. Nom's creator Geoffroy Couprie has really gone above and beyond to make some good docs!

I'm still figuring out a good solution for blog comments, but for now you can email me, send me a tweet @adam_chal, reply on the GitHub issue for this blog post or just reply on whichever Reddit/HN site you're using. Thanks for reading!

(oh, and if you liked this, and want to get paid to talk about Rust full time with me: drop your resume to ehfgwbof@pybhqsyner.pbz after rot13 -- currently hiring in the EU and USA, ideally Lisbon or Austin)

1

Nom is inspired by Parsec, the original parser combinator library from Haskell. I first learned about Parsec from its chapter in Real World Haskell and was impressed by how easily the concept could be ported into Rust.

2

There are two kinds of parsers, *complete* for when you can read the whole input into memory, and *streaming* for when you're getting the input bit by bit. For Advent of Code, the input files are always big enough to fit into memory. But if you were streaming, say, really huge datasets that couldn't fit into memory, you'd use *streaming*. Or if you were reading from a streaming protocol like an HTTP or gRPC body, you'd probably use streaming there too.

3

Note the type signatures haver a lot of complicated boilerplate -- that's because I wanted the parsers to support either the default Nom error, or Nom's VerboseError. If you have a suggestion for improving it without breaking backwards compatibility, please let me know!

5 January 2022

#rust #programming #nom #parsing

© A Less Mad March 2022

Powered by Zola, Theme Anpu.