Parsing

The Rust ecosystem has a number of pretty good crates for reading and writing to popular formats, including yaml, json, xml, and toml.

It also has a great framework - Serde - for serialization and deserialization to and from a common data model, as well as to and from user defined structs and enums. In fact Serde's name is just a shortening of Serialization and Deserialization, as that is what it does. The framework itself does not provide parsing capabilities itself. That is a job taken up by a number of other crates. We will take a look at Serde soon, because it is simple to use (as opposed to extend), but first we will cover basic parsing.

For basic needs, rust has a decent regex crate, which offers good performance and most of the features that you would expect.

regex usage

Compiling a Regex

To compile a regular expression, use `Regex::new`.

```
# main.rs
use regex::Regex;
...
let re = Regex::new(r"'([^']+)'\s+\((\d{4})\)")?;
```

Checking for matches with `is_match`

Returns a bool indicating the success or failure.

```
let text = "I categorically deny having triskaidekaphobia.";

let re = Regex::new(r"\b\w{13}\b")?;

if re.is_match(text) {
    println!("thats a match folks");
} else {
    println!("No Match");
}
```

rust playground

Finding with `find`

The `find` method takes a search `&str` and returns an `Option<Match>`. The Match provides the start and end byte range of the leftmost-first match in the text.

```rust
let text = "I categorically deny having triskaidekaphobia.";

let mat = Regex::new(r"\b\w{13}\b")?.find(text).ok_or("Nothing found")?;

assert_eq!(mat.start(), 2);
assert_eq!(mat.end(), 15);
```

rust playground

Find all the matches with `find_iter`

While `find` returns the first match, `find_iter` finds them all, returning an iterator so that you can visit each match in turn. In our example, we look for text which is exactly 13 unicode characters long.

```rust
let text = "Retroactively relinquishing remunerations is reprehensible.";

for mat in Regex::new(r"\b\w{13}\b").unwrap().find_iter(text) {
    println!("{}",  mat.as_str());
}
```

rust playground

Retrieving capture groups with `captures`

In order to find and capture text, you can use capture groups. Here we try and capture all text between two single quotes, followed by at least one space, then a four digit number between parens, which are not part of the capture group.

```rust
// compile regex
let re = Regex::new(r"'([^']+)'\s+\((\d{4})\)")?;
let text = "Not my favorite movie: 'Citizen Kane' (1941).";

// apply regex and get capture groups
let caps = re.captures(text).ok_or("No Captures")?;

assert_eq!(caps.get(1).unwrap().as_str(), "Citizen Kane");
assert_eq!(caps.get(2).unwrap().as_str(), "1941");
assert_eq!(caps.get(0).unwrap().as_str(), "'Citizen Kane' (1941)");

// You can also access groups by index
assert_eq!(&caps[1], "Citizen Kane");
assert_eq!(&caps[2], "1941");
assert_eq!(&caps[0], "'Citizen Kane' (1941)");
```

rust playground

Iterating over captures with `captures_iter`

Lets return to our earlier example and look for text that is again exactly 13 characters long:

```
let text = "Retroactively relinquishing remunerations is reprehensible.";

println!("text to search: '{}'", text);

for mat in Regex::new(r"\b\w{13}\b")?.find_iter(text) {
    println!("{:?}", mat);
}
```

rust playground

Splitting Text with `split` and `splitn`

While `str` has a `split` method, the `regex` crate provides a means to split strings based on a regular expression. The result is an iterator over the substrings of the text delimited by a match of teh regular expression. Actually the crate provides two methods, because there is a `splitn` method that allows you to specify the maximum number of substrings to split the input into.

Here, in our example, we split by one or more tabs and spaces:

```
let re = Regex::new(r"[ \t]+")?;
let fields: Vec<&str> = re.split("a b \t  c\td    e").collect();
assert_eq!(fields, vec!["a", "b", "c", "d", "e"]);
```

Replacing Text with `replace`, `replace_all`, and `replacen`

`regex` provides a triad of methods to support replacing either the leftmost match, in the case of `replace`, all matches, in the case of `replace_all`, or n matches in the case of `replacen`.

These functions are interesting in that they take the text to operate on as a `&str`, and the replacement as any type implementing the Replacer trait. The trait has a method,`replace_append` which receives a reference to the Captures as well as the destination as a mutable reference to a String.

Replacer is implemented generically for any `FnMut` which takes a reference to `Captures`, and returns a type which implements `AsRef<str>`. What this means in practice, is you can either pass in a str to `replace*` methods or a closure that meets the requirements. IF you do pass in a str, you can access capture groups using special syntax. In general, you may index a capture group using `${#}` -- for example `${1}` to reference the first capture group. Non extant group references will be replaced with the empty string.

For example, a simple string replacement for a regular expression matching all characters except 0 or 1 might be:

```
let re = Regex::new("[^01]+")?;
let results = re.replace("1078910", "");
println!("{}", result);
```

rust playground

However, if you add a capture group in the mix, things get a bit more interesting.

```
let re = Regex::new("([^01]+)")?;
let result = re.replace("1078910", " ${1} ");
println!("{}", result);
```

Alternatively, you can use a closure instead of a string as your Replacer.

```
let re = Regex::new("(b[a-z]{2})")?;
let result = re.replace_all("foo bar bla",  |caps: &Captures| {format!("
{}", &caps[1].to_uppercase()) });
println!("{}", result);
```

rust playground

That is just touching the surface. The documentation does a great job of going over the intricacies of this family of functions.

Splitting with `split`

The next interesting method on regex is `split`, which is used for splitting text by the match to the regex.

```
let re = Regex::new(r"[-:]+")?;
let fields: Vec<&str> = re.split("foo:bar-bla").collect();
println!("{:?}",fields);
```

rust playground

Evaluating Multiple regular expressions with RegexSet

Let's say that you want to apply a set of regular expressions to text and find out if any of them match? You can do this with RegexSet.

```
use regex::RegexSet;

fn is_workdir(text: &str) -> bool {
    let re = RegexSet::new(&[
        r"^/dd/shows/(?:\w+/){1,3}user/work\.\w+",
        r"/dd/dept/\w+/user/work\.\w+"
    ]).expect("unable to compile regex");
    re.is_match(text)
}

fn main()-> Result<(), Box<dyn std::error::Error>> {
    let paths = ["/dd/dept/software/user/work.jgerber",
"/dd/shows/FOOBAR/user/work.jerber",];
    for path in paths.iter() {
        println!("path {} is a work dir? {}", &path, is_workdir(&path));
    }
}
```

rust playground

Performance

One thing that we have been doing so far is compiling the regex each invocation. We should make the regex static. There is a great crate called lazy_static for this:

One node. The lazy_static crate exposes a proc_macro. Proc Macros have an import quirk. You have to do the following in the root of your crate (lib.rs)

```
#[macro_use]
extern crate lazy_static
```

Then to use it, do this:

```
use lazy_static;

fn is_workdir(text: &str) -> bool {
    lazy_static!{
        static ref RE: RegexSet =  RegexSet::new(&[
            r"^/dd/shows/(?:\w+/){1,3}user/work\.\w+",
            r"/dd/dept/\w+/user/work\.\w+"
        ]).expect("unable to compile regex");
    }

    RE.is_match(text)
}
```

Homework Part 1

Now that you have taken a look at regex, why not extend your levelspec implementation to validate the inputs? Give it a shot.

Limitations

Of course, regular expressions are great, but they have limitations. Most obviously, they can only be used to parse regular languages. So they might do in a pinch to extract or parse something particular, but you aren't going to be writing a general regex parser in Rust or any other language, to parse, say HTML, or C++; Because HTML and C++ are examples of context free grammars, not regular grammars. Don't believe me? I'll lend you my Dragon book; its just taking up shelf space.

So where do we go from here? We need to use the right tool for the job if we want to be able to parse .

You will be happy to learn that at this point, we wont be cracking open said book. We wont even be looking at Rust bindings for lex and yacc. Instead, we are going to take a look at a crate called "nom", which is one of the most popular parsing frameworks for Rust. It falls into a category of parsers known as parser combinators.

Parser Combinators

A parser combinator is a higher-order function that accepts several parsers as input and returns a new parser as output. Of course higher-order functions are just functions that operate on other functions. We are already familiar with closures in Rust, so that isn't such a big deal. In practice, when employing a parser combinator framework, you write very targeted, small parsers (each responsible for recognizing some small bit), which you combine into bigger, more capable parsers, until, at some point, you can parse whatever you set out to handle in the first place. They are generally simple to build incrementally, simple to test along the way, and surprisingly performant.

Nom is easily the most popular parsing framework in Rust. It is currently on version 5.1.2 and there is a 6.0.0.alpha1 out in the wild as well. You may not have a good feel for Rust crates yet, but let me tell you, that it is pretty amazing to find a crate on the verge of its 6th major version at this point.

Starting Simple

Well lets get started with something really simple. Let's take a look at a trivial parser to recognize text between parentheses. In other words, given a string like `"( this is the input )"`, we want to extract `" this is the input"`.

To achieve this we are going to use 3 parsers:

- The first will parse open parens
- The second will parse input that is not a close paren
- The third will parse close parens.

And because we are interested in extracting the middle bit between the first and third parser, we will use a combinator that takes three parsers, and returns the result from the middle parser, assuming they all match.

Nom ships with a large number of higher order functions which take some input, and return parsers. Our first order of business will be to find one which will help us match an open paren. Since we want to match a single character, the `char` parser is perfect for our first use case. Invoking `char('(')` returns a parser that will attempt to parse out an open paren.

Next we want to parse anything which isn't a close parentheses. To do this we will use `is_not`. This function takes a char as input and returns a parser that will consume input until it encounters the supplied character. So, we want `is_not(')')`.

Lastly, we want to match the close parentheses itself. So, lets go back to `char` and use `char(')')`.

Now we need a way to combine these parsers sequentially, but discarding the results of the char parsers, since we only want to get at the results of the middle parser - characters that are surrounded by parentheses but are not parenthesis. To do this, we will use the `delimited` function, which takes three parsers and returns a parser that applies all three but discards the first and third. This would look something like this:

```
let parser = delimited(
    char('('),
    is_not(')'),
    char(')')
);
```

Now that we have the pieces in mind, lets put them all together into a working program:

```
use nom::{
  IResult,
  sequence::delimited,
  character::complete::char,
  bytes::complete::is_not
};

fn parens(input: &str) -> IResult<&str, &str> {
  delimited(char('('), is_not(")"), char(')'))(input)
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let result = parens("( this is a test )")?;
    println!("{:?}", result);
    Ok(())
}
```

Lets break down `parens` a bit.

Parens takes an input &str and returns an IResult<&str, &str>. IResult is a type alias that simplifies a rather more complex Result type. For now, just understand that that the types that you provide to IResult represent the input and output data types of your parser, not the Ok and Err types. Err is predefined by default for you.

`parens` is made up of three parser instances wrapped in a combinator.

The first parser is `char('(')`. `char()` is a function that takes a character (a unicode code scalar value) and returns a parser which operates on a single character of the data and determines whether it matches the supplied character or not. In this case, we are trying to match an open parentheses.

It should be noted that nom parsers do not have to completely consume the input data. In general if they are successful, they will split the input into two parts: the part that they have no opinion on, and the part that they have successfully parsed.

In the case of `char('(')`, here is an example of how it would handle various inputs:

```
// given input return (remainder, parsed)
"(this is good)" -> Ok("this is good)", "("))

// given unparsable input, return an error with (input, ErrorKind)
// In otherwords, what gave us trouble, and what parser choked
"a(this is good)" -> Err(Error(("a(this is good)", Char)))
```

More on IResult

Here is the type definition for IResult:

```
type IResult<I, O, E = (I, ErrorKind)> = Result<(I, O), Err<E>>;
```

To make it a little more readable, here I will expand the type variable names to indicate what their functions are:

```
type IResult<Input, Output, Error =(I, ErrorKind)> =
Result<(Input,Output), nom::Err<E>>;
```

The IResult has three generic parameters:

- I for the type of the Input data
- O for the type of the output data
- E for the type of the error, which defaults to a tuple of Input data, ErrorKind
    - ErrorKind is an enum which nom defines and which indicates which parser an error has been encountered in.

The return type when IResult is successful is a tuple - (Input, Output). What it really amounts to is actually `(Remainder, Consumed)`

The return type when IResult is unsuccessful is, by default, an instance of nom::Err

When you use IResult, you are defining the return types of the return tuple. `IResult<&str, &str>` means that we expect a tuple of &str, where the first element is the remaining &str that was not parsed, and the second tuple is the successful part.

Of course, you can use IResult with a custom error type, however, that is a bit of a sideshow that we wont be looking at for now.

Exercise

Throughout our exploration of Nom, we will be working towards a simple goal. We want to be able to build a parser to handle cfg files, of the sort we find at work (EG platforms.cfg). Here is an example of one of those files:

```
[cent6_64]
architecture = linux_cent6_x86_64
type = LINUX
version = LINUX_x86_64_2.6
legacy = linux_cent6_x86_64
bits = 64
status = Current
python_version = 2.6

[cent7_64]
architecture = linux_cent7_x86_64
type = LINUX
version = LINUX_x86_64_3.10
legacy = linux_cent7_x86_64
bits = 64
status = Current
python_version = 2.7
```

As you can see, the cfg format is pretty simple. Each file consists of zero or more sections. Each section starts with a `[header]`, between brackets, followed by one or more `key = value` lines.

So, where do we start? Well, it looks like the header and the keys all would typically be deserialized as keys in a map or fields in a struct. That gives me an intuition that they need to be words that start with a letter, followed by letters or numbers. Lets start there.

But first, we will need a project. This project should be a library. Lets call it `cfgparser`.

```
cargo create --lib cfgparser
```

create a parser that parses one or more alphabets followed by numbers and alphabets

The nom::character::complete module has a number of parsers for dealing with character data.

Ok, the first parser listed is `alpha1` and its definition is "`Recognizes one or more lowercase and uppercase ASCII alphabetic characters: a-z, A-Z`". That seems like it will do nicely to start us off. A couple parsers down the list, I see `alphanumeric0` , parses zero or more numbers or alphabetic characters. Great. That is what we need.

Now we need a way of combining them. A combinator. Lets go look at nom::sequence, which provides different combinators for sequencing parsers. We have two parsers that we want to combine serially. So, `pair` seems to fit the bill. Lets hack something together...We will create our first parser -`alphaword` - because it is a word that starts with a letter.

Lets create a file for parsing (`parser.rs`) and a file for housing the parsing tests (`parser_test.rs`). Add the parser module to the lib.rs.

In lib.rs:

```
pub mod parser;
```

Now lets add the parser_test.rs to parser.rs. Open parser and this to the end:

```


#[cfg(test)]
#[path = "./parser_test.rs"]
mod test;
```

This will allow us to load the parser_test contents into our file, but only when we are configured for building tests.

Ok. lets hop over to our parser_test.rs file and add our first test.

```
super::*;

#[test]
fn alphaword_given_a_word_starting_with_a_letter_can_parse() {
    let result = alphaword("a1foo2");
    assert_eq!(result, Ok(("","a1foo2")));
}
```

This is the outcome we want. We want to feed in a string, and get back a tuple with no further characters to parse. Lets figure out how to get there. Back to our parser file.

Lets give it a go.

```
use nom::character::complete::alpha1;
use nom::character::complete::alphanumeric0;
use nom::IResult;

pub fn alphaword(input: &str) -> IResult<&str, &str> {
    pair(alpha1, alphanumeric0)
}
```

Ok. Well we are almost there. `pair` returns a pair, but we want the results of the parser to be combined somehow. And we don't want to have to allocate additionally to do this. This is a bit of a head scratcher. Let's see what Nom has for us. nom::combinator::recognize seems to fit the bill. (I have to admit that this one had me stumped for a bit, until i asked on the forums).

```
use nom::character::complete::alpha1;
use nom::character::complete::alphanumeric0;
use nom::IResult;

pub fn alphaword(input: &str) -> IResult<&str, &str> {
    recognize(
        pair(
            alpha1,
            alphanumeric0
        )
    )(input)
}
```

Lets give our test a go. It should pass.

We have made a good start on this. Now its your turn. Keep on going...