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

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, 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.

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