Parsing Scheme with nom

Leon Rische

[2017-09-05 Tue 10:55]

Contents

1 Update, 2018-12-11

With version 4 of nom a lot of things changed so the code snippets won't work anymore and some descriptions might be outdated.

2 Introduction

While trying to build a scheme interpreter in rust, I had a hard time getting nom to do what I want and understanding the difference between all the macros it provides.

Don't get me wrong, it's an amazing framework, but most of the example parsers are hard to understand, embedded in lots of unrelated code or outdated (e.g. using the `chain!` macro that has been replaced by `do_parse!`).

This is my attempt to create some kind of tutorial that starts with the simplest building blocks, explores subtle differences between macros and some mistakes to avoid, and hopefully results in working parser for an R5RS Scheme.

3 Setup

1. Make sure you know how to set up and run rust projects crate, at the time of writing this, the newest version is 3.2.0

2. Take a look at chapter 7 of R5RS to see what the target grammar looks like.

3. Create a new cargo (binary) project and install the following dependencies

language=toml,label= ,caption= ,captionpos=b,numbers=none [dependencies] nom = "3.2.0" rustyline = "1.0.0"

`nom` is a parser combinator framework, `rustyline` an implementation of readline that we're going to use to create a simple REPL for experimenting with parsers.

language=rust,label= ,caption= ,captionpos=b,numbers=none // nom defines a ton of macros, make them available here [macro$_u se]externcratenom; externcraterustyline;$

use rustyline::error::ReadlineError; use rustyline::Editor;

fn main() let mut rl = Editor::<()>::new(); if let Err$_) = rl.load_history("history.txt")println!("No$

The above code is a copy of the example on the rustyline github page.

`cargo run` now yields a nice REPL ("RPL" would be more accurate).

```
No previous history.
>> foo
Line: foo
>> bar
Line: bar
```

```
>> baz
Line: baz
>>
CTRL-C
```

4 Keywords

A good place to start is the section on syntactic and expression keywords. We are going to use three nom macros to parse these, `named!`, `tag!` and `alt!`.

Let's see what the nom docs have to say about them:

> `named!`: Makes a function from a parser combination
>
> `tag!`, `tag_s!`: recognizes a specific suite of characters or bytes. `tag!("hello")` matches "hello"
>
> `alt!`: try a list of parsers and return the result of the first successful one

The heading describes `alt!` as a "Choice Combinator" which sounds pretty smart and could be useful for name-dropping ;)

For now, don't think too much about what the input and output types of these functions are, we'll get to it later.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(syntactic$_k$*eyword*, *tag*!(”*else*”));

fn parse(line: str) let res = syntactic$_k$*eyword*(*line*.*as*$_b$*ytes*()); *println*!(”*Parsed*:?”, *res*);

fn main() // ... match readline Ok(line) => rl.add$_h$*istory*$_e$*ntry*(*line*); *parse*(*line*); , //...

First, we create a new parser `syntactic_keyword` that parses only the string (`tag!`) "else", then a helper function that takes a line, feeds it to the parser and prints out the result.

`.as_bytes()` is necessary because nom works on slices of bytes (`u8`) and `&str` is a slice of `chars` (to support unicode).

In case you didn't know, `{:#?}` is a formatting literal that can be used to pretty-print the debug version of some variable.

`cargo run` [1]

```
>> else
Parsed Done(
    [],
```

[1] I reformatted the output to keep it short

```
    [ 101, 108, 115, 101 ]
)
>> foo
Parsed Error(Tag)
>> elsefoo
Parsed Done(
    [ 102, 111, 111 ],
    [ 101, 108, 115, 101 ]
)
>> els
Parsed Incomplete(Size(4))
>>
```

What appears to have happened?

All our parsers return a `IResult`, which, according to the docs, can be one of the following types:

> `Done(I, O)` indicates a correct parsing, the first field containing the rest of the unparsed data, the second field contains the parsed data
>
> `Error(Err<E>)` contains a `Err`, an enum that can indicate an error code, a position in the input, and a pointer to another error, making a list of errors in the parsing tree
>
> `Incomplete(Needed)` `Incomplete` contains a `Needed`, an enum than can represent a known quantity of input data, or unknown

This explains all the kinds of output we are seeing in the REPL.

`[ 101, 108, 115, 101 ]` is just a list of the bytes in `"else"`, `101` is ASCII for 'e', etc.

- "else" can be parsed fully, the `I` of `Done` is empty.
- "foo" can not be parsed and returns an `Error`
- "elsefoo" can be parsed up to "foo", the result is a `Done` with `I` = "foo" (as bytes) and `O` = "else"
- "els" could be parsed, but there is something missing (a fourth byte)

4.1 Other Return Types

Let's add the remaining keywords:

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(syntactic$_k eyword, alt!(expression_k eyword|tag!("else")|tag!("=>")|tag!("define")|tag!("unquote")|t$ splicing")));

named!(expression$_k eyword, alt!(tag!("quote")|tag!("lambda")|tag!("if")|tag!("set!")|tag!("begin")$ ")|$tag!("let")|tag!("do")|tag!("delay")|tag!("quasiquote")$)));

This gives us a first glimpse at the power of parser combinators, because `alt!` can combine any kind of parser (as long as the result types are the same) we can create a second parser `expression_keyword` and combine it with the =tag!=s without any problems.

Getting slices of bytes as a result is not that useful (in this case), but luckily there is an easy solution.

language=rust,label= ,caption= ,captionpos=b,numbers=none [derive(Debug)] enum SyntacticKeyword Else, Arrow, Define, Unquote, UnquoteSplicing, Expression(ExpressionKeyword)

[derive(Debug)] enum ExpressionKeyword Quote, Lambda, If, Set, Begin, Cond, And, Or, Case, Let, LetStar, LetRec, Do, Delay, Quasiquote

Because we are using `{:#?}` to output the results, both enums need to implement the `Debug` trait. `#[derive(Debug)]` tells rust to do this for us.

The last step is to change the result type from `&[u8]` (a slice of bytes) to `SyntacticKeyword` or `ExpressionKeyword` using `map!` and closures `|arg1, arg2, ...| body`.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(syntactic$_k eyword < SyntacticKeyword >, alt!(map!(expression_k eyword, |e|SyntacticKeyword :: Expression(e))|map!(tag!("else"), |_|SyntacticKeyword :: Else)|map!(tag!("=>$ "), $|_|SyntacticKeyword :: Arrow)|map!(tag!("define"), |_|SyntacticKeyword :: Define)|map!(tag!("unquote"), |_|SyntacticKeyword :: Unquote)|map!(tag!("unquote-$ splicing"), $|_|SyntacticKeyword :: UnquoteSplicing))$));

named!(expression$_k eyword < ExpressionKeyword >, alt!(map!(tag!("quote"), |_|ExpressionKeyu$ Quote)|map!(tag!("lambda"), $|_|ExpressionKeyword :: Lambda)|//...map!(tag!("quasiquote"), |_|Expre$ Quasiquote)));

The description of `map!` is accurate but doesn't help that much. . .

> `map!`: maps a function on the result of a parser

. . . but its type signature tells the whole story:

language=rust,label= ,caption= ,captionpos=b,numbers=none map!(I -> IResult<I,O>, O -> P) => I -> IResult<I, P>

Thanks to `tag!` we already have parsers with the type signature `&[u8] -> IResult<&[u8], &[u8]>` and want `syntactic_keyword` to be a parser with the type signature `&[u8] -> IResult<&[u8], SyntacticKeyword>` ("try to parse a slice of bytes and ideally return a `SyntacticKeyword`").

This is exactly what `map!` does, the only missing piece is the second argument, a function `O -> P`, in our case `&[u8] -> SyntacticKeyword`[2].

In most cases we don't need the value for `O` because the result doesn't depend on it, so we can "throw it away" with the `_` placeholder.

The one special case is

language=rust,label= ,caption= ,captionpos=b,numbers=none map!(expression$_k eyword, |e|Syntact$
$Expression(e))$

`expression_keyword` where we need to wrap the resulting `ExpressionKeyword` in a `SyntacticKeyword::Expression(...)`.

Because combining `map!` and `alt!` is so common, there is a special syntax for `alt!` with a builtin `map!`:

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(syntactic$_k eyword < SyntacticKeyword >, alt_c omplete!(expression_k eyword => |e|SyntacticKeyword :: Expression(e)|tag!(''else'') => |_|SyntacticKeyword :: Else|tag!('' => '') => |_|SyntacticKeyword :: Arrow|tag!(''define'') => |_|SyntacticKeyword :: Define|tag!(''unquote$ $splicing'') => |_|SyntacticKeyword :: UnquoteSplicing|tag!(''unquote'') => |_|SyntacticKeyword :: Unquote));$

4.2 Problem 1: Early Returns

Playing around with the REPL quickly leads to some unexpected results:

```
>> else
Parsed Done([], Else)
>> =>
Parsed Done([], Arrow)
>> let
Parsed Done(
    [],
    Expression(Let)
)
>> letrec
Parsed Done(
    [114, 101, 99],
    Expression(Let)
```

[2]Or `&[u8] -> Expressionkeyword` for `expression_keyword`

```
)
>>
```

"letrec" parses to `Expression(Let)` with `[114, 101, 99]` ("rec") as remaining input?

Remember the documentation for `alt!`:

[...] return the result of the first successful one [...]

`Done` with some remaining input still counts as "successful" so `alt!` doesn't even try out the alternative for "letrec". There is a similar problem for "let" / "let*" and "unquote" / "unquote-splicing".

An easy fix is to change the order inside `alt!` so that the longest versions come first.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(syntactic$_k eyword < SyntacticKeyword >, alt!(//...tag!("unquote-splicing") =>$
$|_| SyntacticKeyword :: UnquoteSplicing)|tag!("unquote") => |_| SyntacticKeyword :: Unquote)));$

named!(expression$_k eyword < ExpressionKeyword >, alt!(//...tag!("letrec") =>$
$|_| ExpressionKeyword :: LetRec)|tag!("let*") => |_| ExpressionKeyword :: LetStar)|tag!("let") =>$
$|_| ExpressionKeyword :: Let)|//...))$

4.3 Problem 2: Incomplete

```
>> letrec
Parsed Done(
    [],
    Expression(LetRec)
)
>> let*
Parsed Done(
    [],
    Expression(LetStar)
)
>> let
Parsed Incomplete(Size(6))
>>
```

We successfully fixed "let*" and "letrec" but now "let" won't work because the "letrec" branch sees it, starts to parse it, notices it is incomplete and `alt!` happily returns that as a result.

Again there is an easy solution:

`alt_complete!`: is equivalent to the `alt!` combinator, except that it will not return `Incomplete` when one of the constituting parsers returns `Incomplete`. Instead, it will try the next alternative in the chain.

The same problem is hidden in `syntactic_keyword`, too, so we need to change both `alt!` to `alt_complete`.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(syntactic$_{k}eyword < SyntacticKeyword >, alt_{c}omplete!(//...)$);

named!(expression$_{k}eyword < ExpressionKeyword >, alt_{c}omplete!(//...)$)

```
>> let
Parsed Done(
    [],
    Expression(Let)
)
>> letrec
Parsed Done(
    [],
    Expression(LetRec)
)
>> let*
Parsed Done(
    [],
    Expression(LetStar)
)
>> le
Parsed Error(Alt)
>>
CTRL-C
```

4.4 Conclusion

Our parser is now able to parse all the scheme keywords (syntactic or expression) which is a good start.

There is one small problem left, it might be hard to spot because it only affects the way parsing errors are reported. After switching to `alt_complete!`, the result no longer contains the information if the parser failed because the input was incomplete or if there simply was no matching parser which might be useful for reporting parser errors later on.

5 Integers

5.1 Mapping over Results

Nom works with slices of bytes (`&[u8]`) so we need some way to convert these to strings and then parse them into integers.

Rust already provides a method for the first part: `std::str::from_utf8`.

It's type signature looks like this:

language=rust,label= ,caption= ,captionpos=b,numbers=none [u8] -> Result<str, Utf8Error>

We need is convert `&[u8] -> &str`, what is up with that `Result<>` thingy around it?

The problem is that there are some byte sequences that are not valid as UTF-8 sequences.

If our string is just made up of bytes from `0` to `127` (ASCII), everything works fine.

language=rust,label= ,caption= ,captionpos=b,numbers=none fn main() let input = [49, 50, 51]; // ASCII for "123" println!(":?", std::str::from$_u tf8(input)); // => Ok("123")$

`255` is a valid byte value but must not appear in a sequence.

language=rust,label= ,caption= ,captionpos=b,numbers=none fn main() let input = [49, 50, 51, 255]; println!(":?", std::str::from$_u tf8(input)); // => Err(Utf8Errorvalid_u p_t o : 3, error_l en : Some(1))$

There are a ton of other cases where parsing bytes to a string could go wrong, but the one above has to do for now…

Now that we know why there is a `Result<>` around the stuff we want, how do we use `from_utf8` with nom? `map!` from Part 1 won't work and using `.unwrap()` or `.expect()` would be very inelegant.

The solution is surprisingly simple, nom already includes a variation of `map!` that works with functions that return `Result=s and =nom::digit`, a parser that recognizes one or more of the characters '0'…'9'.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(integer<str>, map$_r es!(nom :: digit, std :: str :: from_u tf8));$

5.2 Parsing Integers

Of course `&str` is not what we really want, we still need to parse it to one of the integer types, for now just `i64`.

One way to do this is to use `str.parse::<i64>()`[3] which returns a `Result`, too, so we need to use `map_res!` again.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(integer<i64>, map$_res!(map_res!(nom :: digit, std :: str :: from_utf8), |s : str|s.parse ::< i64 > ()));$

Rust seems to have a hard time figuring out the type of `s` inside the closure (for good reasons, I am sure), so we need set it to `&str` by hand.

To try out our new parser, just change the `parse()` function from Part 1 to use it instead of `syntactic_keyword`.

language=rust,label= ,caption= ,captionpos=b,numbers=none fn parse(line: str) // let res = syntactic$_keyword(line.as_bytes()); let res = integer(line.as_bytes()); println!("Parsed:?"$

Valid values for `i64` range from

$$-2^{63}$$

to

$$2^{63} - 1$$

[4], so an easy way to see how `map_res!` handles errors would be to use

$$2^{63}$$

or higher as input.

```
>> 1
Parsed Done([], 1)
>> 2
Parsed Done([], 2)
>> 3
Parsed Done([], 3)
>> 0004
Parsed Done([], 4)
>> 9223372036854775808
Parsed Error(MapRes)https://www.youtube.com/watch?v=je8UCmQ45h4
```

Funfact, this piece of code panics for the same reason:

language=rust,label= ,caption= ,captionpos=b,numbers=none fn main() println!(":?", std::i64::MIN); println!(":?", -std::i64::MIN); // -9223372036854775808 // thread 'main' panicked at 'attempt to negate with overflow', ...

[3] `::<i64>` is an alternative way of defining which type we want to parse to, usually this is already set by the type of the variable in a assignment, e.g. =let res: i64 = str.parse()=

[4] If you are wondering why the range is assymetric, take a look at how two's complement is defined.

5.3 Processing Signs

As a last step, we'll add support for signed integers (like `-42`). To do that, we need some way to express "An optional '-' followed by one or more digits" as a parser.

`opt!(parser)` makes `parser` optional, so `opt!(tag("-"))` gives us the first part and we already know that `nom::digit` matches one or more digits, the only thing missing is some way to chain them together.

`do_parse!(opt!(tag"-") » digit » ())` creates a parser that matches the desired pattern and returns `()` (no result).

The last piece of the puzzle is `recognize!(parser)` which returns the input if its child parser was successful.

Putting all of them together, get:

language=rust,label= ,caption= ,captionpos=b,numbers=none // Top of the file use nom::digit;

// ...

named!(integer<i64>, map$_r es!(map_r es!(recognize!(do_p arse!(opt!(tag!("-")) >> digit >> ())), std :: str :: from_u tf8), |s : str|s.parse ::< i64 > ()));$

nom has a problem recognizing module paths inside macros, so `nom::digit` won't work inside the `do_parse!`.

Most of the macros take parsers as inputs and return parsers, so we can make our parser less messy by creating a special `integer_literal` parser.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(integer$_l iteral, recognize!(do_p arse!(opt!(tag!("-")) >> digit >> ())));$

named!(integer<i64>, map$_r es!(map_r es!(integer_l iteral, std :: str :: from_u tf8), |s : str|s.parse ::< i64 > ()));$

```
>> -123
Parsed Done([], -123)
>> -0
Parsed Done([], 0)
>> 0
Parsed Done([], 0)
>> 123
Parsed Done([], 123)
>>
```

This has to do for now, in the next part I'll try to handle binary, octal and hex numbers.

5.4 Spec

In addition to decimal integers like those we handled in Part 2, R5RS includes literals for binary, octal and hexadecimal numbers.

- `#b11` (binary)
- `#o17` (octal)
- `#d19` (decimal)
- `#xaf` (hexadecimal)

They are made up of a **radix specifier** (`#b`, `#o`, `#d`, `#x`, none), a **sign** (`+`, `-`, none) and a non-empty **sequence of digits** with given radix.

If there is no **radix specifier**, the default radix is 10 and if there is no sign, the integer is positive (obviously).

5.5 Digit Sequences

In the last part, we used `nom::digit` to match sequences of decimal digits.

There are two other variants of this, `nom::oct_digit` and `nom::hex_digit`.

Sadly there is no `bin_digit` so we need to write it ourselves.

Looking through the list of nom macros, one might assume something like `many1!(one_of!("01"))` would be a good to do so, but `many1!(...)` returns a list of results instead of just a matching sequence of bytes.

`take_while!` sounds more like what we want and has a variant that only matches sequences that are non-empty:

> [...] returns the longest list of bytes for which the function is true. [...]

The signature of `take_while!` looks like this:

language=rust,label= ,caption= ,captionpos=b,numbers=none take$_{w}hile!(T- > bool) => [T]- > IResult < [T],[T] >$

We are working with byte slices, so `T` is `u8`, so what we need is a function that takes a `u8` byte and returns `true` iff it is a binary digit.

language=rust,label= ,caption= ,captionpos=b,numbers=none fn is$_{b}in_{d}igit(char : u8)- > bool//Just'0'wouldbeachar, //puttingbinfrontmarksitasabytechar == b'0'||char == b'1'$

Now we can build our own `bin_digit` parser:

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(bin$_{d}igit, take_{w}hile1!(is_{b}in_{d}$

5.6 More Signs

In addition to `-`, `+` can be used as a sign, too, so we need a way to handle this.

To keep the integers parsers as dry as possible, we'll extract this into its own parser:

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(sign, recognize!(opt!(one$_o f!(" + -"))))$;

The only new thing here is `one_of!(str)`. According to the docs, it …

> … matches one of the provided characters. `one_of!("abc")` could recognize 'a', 'b', or 'c'.

Just using `opt!(one_of!("+-"))` would lead to problems once we use it inside of `do_parse!(sign » digit » ())`, because it's return type (`Option<...>`) is different, so we have to wrap `recognize!` around it to get a sequence of bytes instead.

5.7 Parsing Numbers with Radix

Next we need some way to parse these digit sequences. `str::parse::<i64>()` won't do this time, because there is no way to tell it which radix (2, 8, 10 or 16) to use.

Instead, we can use `i64::from_str_radix(src: &str, radix: u32)` which returns a `Result`, too, so we can just swap the two functions inside the `map_res!` from Part 2.

Doing this for all new variants (and for decimal integers, to keep things consistent) we can build new parsers `integer_literal2`, `integer_literal8`, …, that match sequences signed binary, octal, decimal and hexadecimal numbers.

language=rust,label= ,caption= ,captionpos=b,numbers=none // Top of the file use nom::digit, oct$_d igit, hex_d igit$;

// ...

named!(integer$_l iteral2, recognize!(do_parse!(sign >> bin_d igit >> ()))))$;

named!(integer$_l iteral8, recognize!(do_parse!(sign >> oct_d igit >> ()))))$;

named!(integer$_l iteral10, recognize!(do_parse!(sign >> digit >> ()))))$;

named!(integer$_l iteral16, recognize!(do_parse!(sign >> hex_d igit >> ()))))$;

And based on that, some new parsers that return =i64=s…

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(integer2<i64>, map$_r es!(map_r es!(integer_l iteral2, std :: str :: from_u tf8), |s|i64 :: from_s tr_r adix(s, 2)))$;

named!(integer8<i64>, map$_res!(map_res!(integer_literal8, std :: str :: from_utf8), |s|i64 :: from_str_radix(s, 8)));$

named!(integer10<i64>, map$_res!(map_res!(integer_literal10, std :: str :: from_utf8), |s|i64 :: from_str_radix(s, 10)));$

named!(integer16<i64>, map$_res!(map_res!(integer_literal16, std :: str :: from_utf8), |s|i64 :: from_str_radix(s, 16)));$

Finally, we need to combine all the parsers above into one parser that can handle all kinds of integers, choosing one of the subparsers depending on the numbers **radix specifier**.

nom provides an elegant way to do this, `preceded!` takes two parsers, tries to apply the first one and then returns the result of second one.

Remember that `#d` is optional, so we have to use `opt!` there.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(integer<i64>, alt!(preceded!(tag!("b"), integer2) | preceded!(tag!("o"), integer8) | preceded!(opt!(tag!("d")), integer10) | preceded!(tag!("x"), integer16)));

Now fire up the REPL to check if everything works as expected:

```
>> 123
Parsed Done([], 123)
>> +123
Parsed Done([], 123)
>> #x+FF
Parsed Done([], 255)
>> #x+Ff
Parsed Done([], 255)
>> #b101010
Parsed Done([], 42)
>> #oFF
Parsed Error(Alt)
```

There is a lot of code duplication going on above but I don't want to get into macros just now, so let's call it a day.

6 Booleans

In R5RS there are two boolean literals, `#t` for `true` and `#f` for `false`.

With our newly aquired nom skills, this should be easy:

language=rust,label= ,caption= ,captionpos=b,numbers=none // We can't name this parser bool because that is a registered keyword in rust named!(boolean<bool>, alt!(tag!("t") => $|_{|}true|tag!("f") => |_{|}false)$);

7 Chars

Chars are not that complex either, there are just three cases to handle:

- `#\space` as alias for `' '`
- `#\newline` as alias for `'\n'`
- `#\<any char>`

All of these cases begin with `#\`, so `preceded!(tag!("#\\"), ...)` would be a good start.

language=rust,label= ,caption= ,captionpos=b,numbers=none // Top of the file, // all the digits are needed for the number parsers from earlier parts use nom::digit, oct$_{d}igit, hex_{d}igit, anychar$;

// ...

named!(character<char>, preceded!(tag!("
"), alt$_{c}omplete!(tag!("space") => |_{|}'')|tag!("newline" => |_{|}''|anychar)))$;

For the first two cases, we just match on the names with `tag!` and use the `=>` syntax to return the right `char`. The third case is surprisingly easy as well because `nom::anychar` does exactly what we want: to match any character and return a `char`.

Again we need to use `alt_complete!` instead of `alt!` and put `anychar` at the end of the chain, otherwise `#\space` would get parsed as `#\s` or `#\s` as an `Incomplete #\space`.

8 Combining Types

At the end we want to have a parser that can handle all kinds of scheme values and returns some wrapper type.

For now, there are just four cases to handle:

1. Keywords, from Part 1
2. Numbers, from Part 2 & 3
3. Booleans

4. Characters

language=rust,label= ,caption= ,captionpos=b,numbers=none [derive(Debug)] enum Token Keyword(SyntacticKeyword), Number(i64), Boolean(bool), Character(char),

In addition to that new wrapper type, we need a new parser that combines the parsers for all types and wraps the results in the corresponding `Token` type.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(token<Token>, alt!(syntactic$_k eyword => |kw|Token :: Keyword(kw)|integer => |i|Token :: Number(i)|boolean => |b|Token :: Boolean(b)|character => |c|Token :: Character(c)));

language=rust,label= ,caption= ,captionpos=b,numbers=none fn parse(line: str) let res = token(line.as$_b ytes()); println!("Parsed:?", res);

9 Testing

An assertion might look like this:

language=rust,label= ,caption= ,captionpos=b,numbers=none assert$_e q!(boolean("t".as_b ytes()), no IResult :: Done(b""[..], true)); assert_e q!(boolean("f".as_b ytes()), nom :: IResult :: Done(b""[..], false));

There is a lot of boilerplate code because the input has to be a `&[u8]`, not `&str` and we expect our input to be parsed fully, so the first part of `Done` is an empty `&[u8]` (which we get by `&b""[..]`).

A nice fix is to write a macro that takes the parts we care about (parser, input string, output value) and fills in the rest:

language=rust,label= ,caption= ,captionpos=b,numbers=none macro$_r ules!assert_p arsed_f ully(pars input : expr,result:expr) => assert$_e q!(parser(input.as$_b ytes()), nom :: IResult :: Done(b""[..],result));

Now we can write tests in a much cleaner way:

language=rust,label= ,caption= ,captionpos=b,numbers=none [test] // This marks functions as unit tests, they can be run with 'cargo test' fn test$_b ool()assert_p arsed_f ully!(boolean, "t", true); assert_p arsed_f ully!(boolean, "f", false);

[test] fn test$_c haracter()assert_p arsed_f ully!(character, "space",' '); assert_p arsed_f ully!(character, "n

[test] fn test$_i nteger()assert_p arsed_f ully!(integer, "1", 1); assert_p arsed_f ully!(integer, "d + 1", 1); ass

In order to use `assert_eq!` on `Token=s, we need to define a way to test if two of them are equal, formalized in the =PartialEq` trait.

We won't use `Eq` here, because in the future there might be some tokens (e.g. =NaN=) where the equivalence relation is not reflexive (`v !` v= for some token `v`).

Just like the `Display` trait, we can make rust derive `PartialEq` automatically by adding it in the `#[derive(...)]` above `Token`, `SyntacticKeyword` and `ExpressionKeyword`.

language=rust,label= ,caption= ,captionpos=b,numbers=none [derive(Debug, PartialEq)] enum Token // ..

Now `assert_parsed_fully!` works for tokens, too.

language=rust,label= ,caption= ,captionpos=b,numbers=none [test] fn test$_t oken() assert_p arsed_f ully!(token, "1", Token :: Number(1)); assert_p arsed_f ully!(token, "else", Tok$

I'll leave coming up with more test cases as an exercise for the reader. If you find a case that does not work as expected, feel free to open up an issue.

10 Strings

The R5RS spec for strings is pretty simple, but in addition to that, support for `\n`, `\r` and `\t` would be nice.

```
<string> → " <string element>* "
<string element> → <any character other than " or \ > | \" | \\
```

nom seems to have two options to handle escaped strings:

- `escaped!`
- `escaped_transform!`

Let's use the later one, because the example code already does 80% of what we want.

language=rust,label= ,caption= ,captionpos=b,numbers=none fn to$_s(i : Vec < u8 >) - > StringString :: from_u tf8_l ossy(i).into_o wned()$

named!(string$_c ontent < String >, map!(escaped_t ransform!(take_u ntil_e ither!(\text{"}\Xi), '', alt!(tag!("") =$

The only changes are to use `take_until_either!("\"\\")` to matche any characters until either a \ or a " appears instead of `alpha` and add support for `\r` and `\t`.

Based on this parser for stuff inside the ", next we need a way to make sure there are `"=s around our strings. =delimited!` is similar to the earlier `preceded!` and does just that, it takes three parsers

- opening delimiter
- body
- closing delimiter

and returns only the result for the body.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(string<String>, delimited!(tag!("\""), string$_c ontent, tag!("\"")));

Now we only need to add a string type to the `Token` enum, the `string` parser to the `token` parser and everything should work fine.

language=rust,label= ,caption= ,captionpos=b,numbers=none [derive(Debug, PartialEq)] enum Token Keyword(SyntacticKeyword), Number(i64), Boolean(bool), Character(char), String(String),

named!(token<Token>, alt!(syntactic$_k eyword => |kw|Token :: Keyword(kw)|integer => |i|Token :: Number(i)|boolean => |b|Token :: Boolean(b)|character => |c|Token :: Character(c)|stri$ $|s|Token :: String(s)));

```
>> "seems to work"
Parsed Done([], String("seems to work"))
>> "test123 \n\t\r\"\"\\"
Parsed Done([], String("test123 \n\t\r\"\"\\"))
>>
```

11 Identifiers

We're making good progress, booleans, numbers (in a simplified form), characters and strings already work, the only missing part is a parser for identifiers.

You might ask "But what about the `keyword` parser from Part 1"?

Turns out, we don't even need it for the `token` parser, but it will come in handy once we start to parse expressions.

```
<token> →
  <identifier> |
  <boolean> |
  <number> |
  <character> |
  <string> |
  ( | ) | #( | ' | ` | , | ,@ | .
```

11.1 Peculiar Identifiers

Let's start with something simple, "peculiar identifiers":

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(peculiar$_i dentifier, alt!(tag!$ ")|tag!(" − ")|tag!("...")));

named!(identifier<String>, map!(peculiar$_{i}dentifier, |s|String :: from_{u}tf8_{l}ossy(s).into_{o}wned()$));

A combination of `alt!` and `tag!` matches each of the peculiar identifiers and we can use the same method as in the `to_s` function from earlier to convert `&[u8]` to `String`.

Next we need add a `Identifier` type to the `Token` enum and parser. Note that I removed the `Keyword` type, too.

language=rust,label= ,caption= ,captionpos=b,numbers=none [derive(Debug, PartialEq)] enum Token Number(i64), Boolean(bool), Character(char), String(String), Identifier(String),

named!(token<Token>, alt$_{c}omplete!(integer => |i|Token :: Number(i)|boolean => |b|Token :: Boolean(b)|character => |c|Token :: Character(c)|string => |s|Token :: String(s)|identifier => |s|Token :: Identifier(s)$));

Again it's important to use `alt_complete!` instead of `alt!` to avoid conflicts between the number `+1` and the identifier `+`.

11.2 "Common" Identifiers

First we need some helper classes to match the different groups of characters. We can't use `nom::digit` or `nom::alpha` here because they match multiple characters while we only want to match a single one.

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(letter<char>, one$_{o}f!("abcdefghijklmnopqrstuvwxyz")); named!(single_{d}igit < char >, one_{o}f!("0123456789")); nam$ $char >, one_{o}f!($"!named!(special$_{s}ubsequent < char >, one_{o}f!(" + -.@"))$;

I'm sure there is a more elegant way to do this but `one_of!` with a string of all characters is good enough for now. The result of these parsers is `char`, not `&[u8]` so we need to explicitely annotate their type.

Like in Part 3 we can use a combination of `recognize!` and `do_parse!` to match "common" identifiers:

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(common$_{i}dentifier, recognize!(do_{p}arse!(initial >> many0!(subsequent) >>$ ()))));

Finally change `identifier` to support both types:

language=rust,label= ,caption= ,captionpos=b,numbers=none named!(identifier<String>, map!(alt!(peculiar$_{i}dentifier|common_{i}dentifier), |s|String :: from_{u}tf8_{l}ossy(s).into_{o}wned()$));

And add the remaining tokens:

language=rust,label= ,caption= ,captionpos=b,numbers=none [derive(Debug, PartialEq)] enum Token Number(i64), Boolean(bool), Character(char), String(String), Identifier(String), LBracket, RBracket, HashBracket, Quote, Quasiquote, Unquote, UnquoteSplicing, Dot named!(token<Token>, alt$_complete!(integer => |i| Token::Number(i) | boolean =>$ $|b| Token::Boolean(b) | character => |c| Token::Character(c) | string =>$ $|s| Token::String(s) | identifier => |s| Token::Identifier(s) | tag!("(") =>$ $|_| Token::LBracket | tag!(")") => |_| Token::RBracket | tag!("(") => |_| Token::HashBracket | tag!("'")$ $|_| Token::Quote | tag!("`") => |_| Token::Quasiquote | tag!(",@") => |_| Token::UnquoteSplicing | tag!("$ $|_| Token::Unquote | tag!(".") => |_| Token::Dot$));

A quick test shows that everything works as expected and there don't seem to be any strange conflicts between identifiers and numbers:

```
>> test
Parsed Done([], Identifier("test"))
>> +1
Parsed Done([], Number(1))
>> +
Parsed Done([], Identifier("+"))
>> ...
Parsed Done([], Identifier("..."))
>> $foo123
Parsed Done([], Identifier("$foo123"))
>> .
Parsed Done([], Dot)
>> (
Parsed Done([], LBracket)
>> #(
Parsed Done([], HashBracket)
>>
```

This was easier than I expected, but I'm sure things will get more exiting once we start parsing expressions.

Full source code: l3kn/r5rs-parser.