**Fun with lexical analysis and Rust**

[[](https://medium.com/@blotato?source=post_page-----5707e7010cee--------------------------------)](https://medium.com/@blotato?source=post_page-----5707e7010cee--------------------------------)

[[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----5707e7010cee--------------------------------)](https://levelup.gitconnected.com/?source=post_page-----5707e7010cee--------------------------------)

[Blotato.com](https://medium.com/@blotato?source=post_page-----5707e7010cee--------------------------------)

·

Follow

Published in

[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----5707e7010cee--------------------------------)

·

7 min read

·

Nov 25

7

In this post, we’ll use the Rust programming language to implement a lexical tokenizer for a simple Cron expression.



**Introduction**

Let’s talk about lexical analysis. I was originally going to name this post “Implementing Tokenizer” but the tides have turned, the times are changing, and for the sake of not enduring a squall of comments such as “bruh where’s my LLama BPE tokenizer you promised me”, we will stick to lexical analysis for now.

This article is meant for beginners in lexical analysis of Rust, so please be mindful of the audience before you start your “um my hand-rolled table lookup is 10x better than this crap” and “these lifetimes are making me end my own”.

However, constructive comments and better ways to do things are always appreciated and will most likely be moved to the bottom of this article.

Quite a long disclaimer intro. I hope you chuckled at least once and made it here. Without further ado, let’s begin.

*Lexical tokenization is conversion of a text into (semantically or syntactically) meaningful lexical tokens belonging to categories defined by a “lexer” program*

In plain terms, a bunch of characters go in, and a bunch of tokens things come out. Note that the tokens just have to be meaningful in your context, and there is no one way to skin the cat tokenize a string. For example, a string "hello world" could be tokenized as:

# One single token  
<HELLO\_WORLD>  
  
# Two tokens separated by a whitespace  
<HELLO> <WHITESPACE> <WORLD>  
  
# One token surrounded by non whitespaces  
# Treating non-whitespace characters  
<NON\_WHITESPACE> <SPACE> <NON\_WHITESPACE>

This essentially makes a tokenizer a function of the form *t*: (*S*∗,*T*) → *T*∗, which takes a list of characters from an alphabet S and an a token alphabet and produces a list of tokens from an alphabet T. Or at least I think so.

**Defining tokens**

For the purposes of this article, we’ll implement a tokenizer for a simple Cron expression. We’ll stick to expressions that are comprised of 5 fields exactly and only allow digits 0-9, \*, /, ,, - as tokens, with the following rules for each field



We can fire up a new Rust project by running cargo new cronlexer and get down to business.

I like to start by first defining our tokens in the code. The fact that each token is different from one another makes them a perfect candidate for a Rust enum. Here’s one example of structuring our tokens:

#[derive(Debug, PartialEq, Clone)]  
pub enum Token {  
 Whitespace,  
 Number(u8),  
 Dash,  
 Slash,  
 Star,  
 Comma,  
}

The names of the enum variants are self-explanatory and directly map to the tokens in question. However, you may have noticed we decided to implement a digit token as a Number(u8) , forcing the tokenizer to aggregate all the digits into a number before emitting a token. However, we could have made each digit a separate token like this:

#[derive(Debug, PartialEq, Clone)]  
pub enum Token {  
 Whitespace,  
 Zero,  
 One,  
 Two,  
 Three,  
 Four,  
 Five,  
 Six,  
 Seven,  
 Eight,  
 Nine,  
 Dash,  
 Slash,  
 Star,  
 Comma,  
}

Or even going in the opposite direction and have separate tokens for days, weeks, and year, like so:

#[derive(Debug, PartialEq, Clone)]  
pub enum Token {  
 Whitespace,  
 Minutes(u8),  
 Hours(u8),  
 DayOfMonth(u8),  
 Month(u8),  
 DayOfWeek(u8),  
 Dash,  
 Slash,  
 Star,  
 Comma,  
}

Depending on how we structure it, parsing down the road might be easier or harder. We don’t care about parsing yet, so we will adopt a set of tokens that make sense in our context and treat any number as syntactically correct and absorb it as part of our token. So, we settle on:

#[derive(Debug, PartialEq, Clone)]  
pub enum Token {  
 Whitespace,  
 Number(u8),  
 Dash,  
 Slash,  
 Star,  
 Comma,  
}

**Designing tokenizer interface**

When designing a library interface, I like to think about the end user. How are they going to use our library? What are the inputs and outputs of the exposed public functions and structs and enums? Then, as I discover better ways of using the library, we can improve the ergonomics and developer experience.

For starters, we might want to use our library as follows:

let source = "\*/5 1,2,3 \* \* \*";  
let tokenizer = Tokenizer::new(source.as\_bytes());  
for token in tokenizer {  
 println!("token: {:?}", token);  
}

Since we want to iterate over tokens, by convention we can implement an IntoIterator trait for our Tokenizer, and return the actual iterator there. Typically in Rust, every time you want to return something that implements a set of traits, you wrap it in a struct and have that struct implement them. This can be seen in a lot of places – functions like map, peek, iter, split\_whitespaces, all return different helper structs that then implement the necessary traits.

But then we quickly realize that the tokenization might return an error. For example, tokenizing "schedule every day" in our Cron expression grammar should simply fail.

let source = "schedule every day";  
let tokenizer = Tokenizer::new(source.as\_bytes());  
// Can't really iterate over a failed tokenization can we?  
for token in tokenizer {  
 println!("token: {:?}", token);  
}

Instead, we’ll have a new method tokenize that will return the result of our tokenization. Now in the "real world", there are other things we might need our tokenizer to do – peek at the next value, backtrack, in some optimized fast way. For now, we keep it simple – we just want to tokenize it and iterate over the tokens. Our library usage becomes more like:

let source = "\*/5 1,2,3 \* \* \*";  
let tokenizer = Tokenizer::new(source.as\_bytes());  
// notice the '#', or match on the error  
let tokens = tokenizer.tokenize()?;  
for token in tokenizer {  
 println!("token: {:?}", token);  
}

**Implementation**

Let’s first fill out basic structs. We need the main Tokenizer struct as well as a TokenizerError

// capture some helpful message for the library user  
#[derive(Debug)]  
pub struct TokenizerError {  
 pub message: String,  
}  
  
// vector of tokens  
pub type TokenizerResult = Result<Vec<Token>, TokenizerError>;  
pub struct Tokenizer<'a> {  
 source: &'a [u8],  
 pos: usize,  
}  
impl<'a> Tokenizer<'a> {  
 pub fn new(source: &'a [u8]) -> Self {  
 Self { source, pos: 0 }  
 }  
 pub fn tokenize(&mut self) -> TokenizerResult {  
 todo!()  
 }  
}

Now let’s implement our tokenizer. It will simply scan the tokens from left to right, greedily consuming them and putting them into a resulting vector of tokens.

impl<'a> Tokenizer<'a> {  
 // create a new tokenizer  
 pub fn new(source: &'a [u8]) -> Self {  
 Self { source, pos: 0 }  
 }  
  
 pub fn tokenize(&mut self) -> TokenizerResult {  
 let mut tokens = Vec::new();  
  
 while self.pos < self.source.len() {  
 // match on every single token, and get both the token  
 // and the number of bytes consumed  
 let (token, consumed) = match self.source[self.pos] {  
 // consume the whitespaces greedily  
 b' ' | b'\t' => {  
 let count = self.peek\_while(is\_whitespace);  
 (Token::Whitespace, count)  
 }  
  
 b'-' => (Token::Dash, 1),  
 b',' => (Token::Comma, 1),  
 b'/' => (Token::Slash, 1),  
 b'\*' => (Token::Star, 1),  
 b'0' | b'1' | b'2' | b'3' | b'4' | b'5' | b'6' | b'7' | b'8' | b'9' => {  
 // consume the digits greedily  
 let count = self.peek\_while(is\_digit);  
  
 (  
 Token::Number(to\_u8(&self.source[self.pos..(self.pos + count)])),  
 count,  
 )  
 }  
 ch => {  
 // we found a character that is not in our alphabet  
 // return an error  
 return Err(TokenizerError {  
 message: format!("Could not parse token: {ch}",),  
 })  
 }  
 };  
 // advance internal state  
 self.pos += consumed;  
 // append the token  
 tokens.push(token);  
 }  
  
 Ok(tokens)  
 }  
  
 // peek while the predicate is true  
 // without modifying the state  
 // return the number of elements peeked  
 fn peek\_while<P>(&self, pred: P) -> usize  
 where  
 P: Fn(u8) -> bool,  
 {  
 let mut consumed = 0;  
  
 while (self.pos + consumed) < self.source.len() && pred(self.source[self.pos + consumed]) {  
 consumed += 1;  
 }  
  
 consumed  
 }  
}  
  
// check if a byte is a digit  
fn is\_digit(v: u8) -> bool {  
 v >= b'0' && v <= b'9'  
}  
  
// check if a byte is a whitespace  
fn is\_whitespace(v: u8) -> bool {  
 v == b' ' || v == b'\t'  
}  
  
// quick and dirty way to extract a digit from a byte string  
// e.g. b'12' -> 12  
fn to\_u8(vs: &[u8]) -> u8 {  
 let mut result: u8 = 0;  
 const RADIX: u8 = 10;  
 let len = vs.len();  
 for i in 0..len {  
 result = result + (RADIX.pow((len - i - 1) as u32) \* (vs[i] - b'0'));  
 }  
 result  
}

Some interesting notes and observations here:

* tokenize takes mut self because it modifies internal state and is not idempotent (calling tokenize twice will result in an error). This is a great way to signify to the client that calling tokenize multiple times is not allowed and enforce it at compile time.
* Our Tokenizer takes source by reference. A great way to read this in simple cases like this one is to think about how long a reference to a Tokenizer should be valid for? Well, the source reference must outlive (or live as long as) the Tokenizer otherwise it won't have a valid place to read the data from.
* We maintain current index pos in the internal state, advancing it greedily as we parse the tokens. Token parsing is simply a giant match statement (which could be factored out in a separate function) that will return the token itself and the amount of bytes consumed.
* Our Cron grammar can be unambiguously tokenized. This makes the whole process a fairly straightforward exercise in incrementing the index carefully and matching on the tokens until we run out of bytes.

**Testing**

Let’s write some tests:

#[cfg(test)]  
mod tests {  
 use super::\*;  
  
 #[test]  
 fn basic\_tokenization() {  
 let source = "\*/12";  
 let tokenizer = Tokenizer::new(source.as\_bytes());  
 let tokens = tokenizer.tokenize();  
 assert!(tokens.is\_ok());  
 assert\_eq!(  
 tokens.unwrap(),  
 vec![Token::Star, Token::Slash, Token::Number(12)]  
 );  
 }  
  
 #[test]  
 fn failed\_tokenization() {  
 let source = "why cant't I just say 'every hour'";  
 let tokenizer = Tokenizer::new(source.as\_bytes());  
 let tokens = tokenizer.tokenize();  
 assert!(tokens.is\_err());  
 }  
}

100% coverage achieved. Just the way I write it in prod.



**Blotato.com** — turn existing scripts into production-grade apps in minutes, without migrating code! Blotato gives you instant scalable infrastructure, execution logs, access controls, secrets management, and one-click oauth2 integrations.