UNION COLLEGE

Assignment #1: Building a Lexer

Course: *CSC-375 Compiler Design* – Professor: *Aaron Cass*
Due date: *DUE DATE*

Objectives

- Familiarize yourself with the Rust programming language
- Complete the first part of code compilation, breaking a program string into lexical tokens
- Pass all tests given in *lexer_tests.rs*
- Work in groups of 2-3

An explanation of lexing: From a high level, it might look like this step isn't even necessary. Why would we need to process our code at all before trying to compile it? We already wrote it and all the syntax highlighting checks out, isn't that enough?

Behind the scenes, your text editor is doing a lot of heavy lifting. As humans we can easily read over our code that has comments and whitespace and structure, but to a computer this means nothing. From a computer's perspective, all it sees is one giant long list of characters next to each other with no natural way to see what it all "means".

The point of a lexer is to take this giant string and try to make it a little more readable by a computer program, for example transforming something like "true || false" into the list of tokens [TRUE, OR, FALSE]. Doing this step will make our jobs much easier going forward and will allow us to have a more easily readable list of tokens instead of a raw list of characters to try and work with.

Lexer Structure

Listing 1: mod.rs

```rust
/// Core of the lexing process
pub mod lexer_core;

/// Acceptable tokens
pub mod token;
```

mod.rs:

This file defines the modules related to the lexer and makes them available for other modules to see. You can see this in the folder for the lexer inside the *frontend/* folder. All modules each have documentation in this file to show what the module is used for. The lexer is relatively simple, so all it has is a module containing the tokens it can use and the lexer itself.

You don't have to do anything to this file.

token.rs:

This file contains an **enum** which houses all of the tokens we have available to break the string into. The final output will be a **vector** (sort of like a list) of these tokens. For example, if when scanning through the program string you encounter a '(', you should probably add Token::LPAREN to your output vector. Other tokens will be more complex to detect.

Again, you don't have to do anything here. All of these tokens are given.

lexer_core.rs:

This is where the magic happens. This file is where the structure for taking a string input and scanning through it to create a list of tokens exists.

All of the functions here have been intentionally left blank, and none of the tests will pass. That's kind of a problem since we need our lexer to function, so this is where you come in.

Follow these guidelines to build the lexer structure in lexer_core.rs

1. Do not change any of the function signatures, i.e. leave all parameters, parameter types, and return types as you found them.
2. Follow all instructions given in the comments where they exist, and take any potential hints into consideration when making your design.
3. Remember you are scanning through a string one character at a time, but at times (such as with variable and function identifiers) you may need to collect more than one character for a token, this is okay.
4. Remember to check if a character is what you think it is before pushing a token to the output vector. If you see '<' that might be part of "<=", never assume!
5. Start early. This may seem simple, but for many of you this is an entirely new language experience, and some concepts may take time to get used to.

Cheat sheet for creating tokens

it might not be obvious how to create tokens before pushing them to the output vector, so here are some tips on how to do this.

1. Tokens are part of the **Token enum** as showed before, so when creating one use "Token::" to signal you want to make an item from the Token enum, followed by the token's name. For example, if I want a token for '*' the token I want would be **Token::STAR**.
2. For tokens that have no data (such as a left parenthesis for example) creating tokens is simple and I already showed how. If I detect a left parenthesis, then I would push **Token::LPAREN** to the output vector.
3. Some tokens have many characters at once, for example the integer "42". Once you have all characters collected in a vector of characters (for example we'll call it **vec**) you can create a token with data in it by specifying the name of the token like above and then, in parentheses, the data you want to attach. In our example with 42 I would create **Token::INT(vec)**.