ECS 140A – Programming Languages
Project 2: Rust

Due: October 31, Sunday, 2021, 11:59pm PT

This project specification is subject to change at any time for clarification.

1 Overview

For this project you will be writing a Rust program to parse language X. Language X is similar to the C language, but is simpler and slightly different. For this program you will be reading in an X language file and converting it to a syntax highlighted XHTML version. You will be creating several *structs* (similar as C++ classes) that will solve the various portions of the problem.

2 X Programming Language

The following EBNF describes the X language. The non-terminals on the right are identified by italics. Literal values are specified in bold. The operators not in bold or italics describe the special options of the EBNF: {} for repetition, [ ] for optional, () for grouping, and | for or.

The X programming language allows for nested function declarations. Variables and functions must be declared before they are referenced or called. Identifiers have the same scoping rules as C. An identifier may only be declared once per block, but may be declared more than once per file.

Token Rules:

```
Identifier := ( _ | Alpha ) { ( _ | Digit | Alpha )  }
IntConstant := [ - ] Digit { Digit }
FloatConstant := [ - ] Digit { Digit } [ . Digit { Digit } ]
Digit := 0 - 9
Alpha := A - Z | a - z
```

EBNF Rules:

```
Program := {Declaration} MainDeclaration {FunctionDefinition}
Declaration := DeclarationType (VariableDeclaration | FunctionDeclaration)
MainDeclaration := void main ( ) Block
FunctionDefinition := DeclarationType ParameterBlock Block
DeclarationType := DataType Identifier
VariableDeclaration := [= Constant] ;
FunctionDeclaration := ParameterBlock ;
Block := { {Declaration} {Statement} {FunctionDefinition} }
ParameterBlock := ( [Parameter {, Parameter}] )
DataType := IntegerType | FloatType
Constant := IntConstant | FloatConstant
Statement := Assignment | WhileLoop | IfStatement | ReturnStatement |
   (Expression ;)
Parameter := DataType Identifier
IntegerType := [unsigned] ( char | short | int | long )
FloatType := float | double
Assignment := Identifier = {Identifier =} Expression ;
WhileLoop := while ( Expression ) Block
IfStatement := if ( Expression ) Block
ReturnStatement := return Expression ;
Expression := SimpleExpression [ RelationOperator SimpleExpression ]
SimpleExpression := Term { AddOperator Term }
Term := Factor { MultOperator Factor }
Factor := ( ( Expression ) ) | Constant | (Identifier [ ( [ Expression {,
   Expression}] ) ] )
RelationOperator := ( == ) | < | > | ( <= ) | ( >= ) | ( != )
AddOperator := + | -
MultOperator := * | /
```

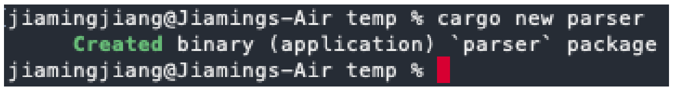
3 Output File

The output text file will output the X language file with fonts and colors specified in *format.csv*. The token classes are IntConstants, FloatConstants, Keywords, Operators, Variables, and Functions. They are listed in "token.rs". The keywords of language X are: **unsigned**, **char**, **short**, **int**, **long**, **float**, **double**, **while**, **if**, **return**, **void**, and **main**. The operators of language X are: **(**, **,**, **)**, **{**, **}**, **=**, **==**, **<**, **>**, **<=**, **>=**, **!=**, **+**, **-**, *****, **/**, and **;**. Each nested block should be indented another level.

4 Instruction

4.1 Getting Started

1. Download and install Rust: `https://www.rust-lang.org/learn/get-started`
2. In the directory where you want to put your program,
   (a) Create a folder ("cargo") named "parser" for your program by running:
   `cargo new parser`
   If the cargo is created successfully, something like this will be displayed:



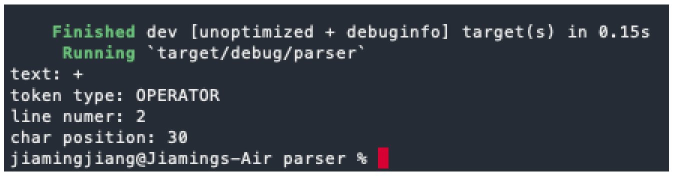
```
jiamingjiang@Jiamings-Air temp % cargo new parser
     Created binary (application) `parser` package
jiamingjiang@Jiamings-Air temp %
```

(b) Inside the "parser" folder, a file named "Cargo.toml" and a folder named "src" are created.

(c) Inside the "src" folder, there is a file named "main.rs". This is similar to the driver program in C++. The structs you created should be in other files and imported into *main.rs*.

(d) Down the starting code from Canvas: *main.rs, token.rs, character_stream.rs* into the "src" folder.

(e) Run your program insider the "parser" directory or "src" using
`cargo run`.
If everything is working correctly, it should show the information of the token created in the mainf unction: Some warning messages will be generated. You may ignore them. They are



```
    Finished dev [unoptimized + debuginfo] target(s) in 0.15s
     Running `target/debug/parser`
text: +
token type: OPERATOR
line numer: 2
char position: 30
jiamingjiang@Jiamings-Air parser %
```

complaining about the unused items defined in the `TokenType`.

4.2 Writing the Scanner (Lexical Analyzer)

Write a struct called `Scanner` that will open the .x file as specified on the command line and tokenize the text of the files into operators, intConstants, floatConstants, keywords, and identifiers.

Note:

- Due to the fact that there is a unary negation "-" operation in X, the scanning has one slight complication. If a "-" sign is followed by digits, but preceded by an ID or constant, it is considered the subtract operator, and not part of the following constant.
- Don't worry about differentiating between variables and function names at this stage. Your parser will deal with this issue later.
- The Scanner struct should have a method named `get_next_token()` or something similar that when called, will return the next token as read from the .x file.
- You need to finish the `CharStream` struct (in character_stream.rs) to be able to read the input file character by character. Unlike C++, Rust does not have a convenient way of reading a file character by character using `FileInputStream`. For simplicity, you may read the whole file at once and store it in a string or a vector of strings. Then you may read the stored string or vector of strings character by character. But keep in mind that this is not how modern scanner works because the input program may be too large to be stored as a second copy.

4.3 Writing the Parser (Syntax Analyzer)

Write a **recursive descent parser** struct that will parse the .x file by making repeated calls to the `GetNextToken()` method of the scanner struct in the previous step.

Note:

- This struct should implement one method per EBNF rule.

- It should also be able to validate that the .x file does conform to the grammar specified. If an error is found in the .x file, the line number of the error and the grammar rule where the error was found should be printed with an appropriate message. Then the parser may terminate. (Also keep in mind that this is not how modern parsers work. But for simplicity, you do not need to find as many syntactic errors as possible.)

4.4 Output XHTML File

Using the three classes written in steps 1 to 3, write a program that reads in a .x source file and outputs a .xhtml file.

Note:

- The colors and styles of tokens in the .xhtml file should match the specification in *format.csv*.
- The .xhtml file should be a valid .html file that can be loaded into any web browser and viewed. You will probably need to begin your .xhtml file with:
  `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1 -transitional.dtd">`
  You may also need to have attributes in your html element. So it should look like:
  `<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">`
- Examples of valid input file .x and the output .xhtml file can be found on Canvas.

4.5 Submission

You must submit your entire "parser" cargo and a README.txt file in a zipped folder.

You should avoid using existing source code as a primer that is currently available on the Internet. You are also not allowed to use the parser tools found on the Internet.

You must specify in your readme file any sources of code that you have viewed to help you complete this project. All class projects will be submitted to MOSS to determine if students have excessively collaborated. Excessive collaboration, or failure to list external code sources will result in the matter being referred to Student Judicial Affairs.