Concepts of Programming Languages, CSCI 305, Fall 2020
Lexical Analyzer for Small Rust

Sept. 18 – Regular expression for each token and for comments, 5%
Sept. 25 – NFA, 5% An NFA with a single start state and multiple accept states labeled with a token name or the action that the scanner should take. (Start on this submission early. It can become complex.)
Oct. 2 - DFA, 5%
Oct. 23 - Lexical analyzer in C#, ~~along with human readable table,~~ 45%
Nov. 6 - Lexical analyzer in Python, ~~along with human readable table (probably the same table,~~ 40%

Small Rust Programs

“Rust is a multi-paradigm programming language focused on performance and safety, especially safe concurrency. Rust is syntactically similar to C++, and provides memory safety without using garbage collection. (…) Rust has gained increasing use in industry and is now Microsoft's language of choice for secure and safety-critical software components. Rust has been named the "most loved programming language" in the Stack Overflow Developer Survey every year since 2016.”
Wikipedia, https://en.wikipedia.org/wiki/Rust_(programming_language)

“The concrete syntax of Rust is similar to C and C++, with blocks of code delimited by curly brackets, and control flow keywords such as if, else, while, and for. Not all C or C++ keywords are implemented, however, and some Rust functions (such as the use of the keyword match for pattern matching) will be less familiar to those versed in these languages. Despite the superficial resemblance to C and C++, the syntax of Rust in a deeper sense is closer to that of the ML family of languages and the Haskell language. Nearly every part of a function body is an expression, even control flow operators. For example, the ordinary if expression also takes the place of C's ternary conditional. As in Lisp, a function need not end with a return expression: in this case if the semicolon is omitted, the last expression in the function creates the return value.”
Wikipedia, https://en.wikipedia.org/wiki/Rust_(programming_language)

Rust is very safe. The language was designed so the compiler can catch most everything. It doesn’t use garbage collection, but ownership of variables is required, so memory can be reclaimed. When a variable is passed to a routine, ownership of that variable is lost to the calling routine, instead it belongs to the called routine. Rust allows the programmer to put complex objects onto the stack.

Small Rust programs contain tokens, whitespace, and comments. Tokens include identifiers, Booleans, character literals, string literals, number literals, open and close parentheses, open and close curly braces, quote, double quote, colon and semicolon. Additionally, keyword tokens, which can be recognized as identifiers and looked up in a table, are to be returned as tokens. Whitespace consists of spaces (0x20), tabs (0x09) and newline (0x0d, 0x0a) characters. A token may be surrounded by any number of whitespace characters. Identifiers, Booleans, character literals, string literals and number literals are delimited by whitespace or opening and closing parentheses (opening before another token and closing after another token).

Tokens:

Name	**Description**	**Symbol**
(	Open parenthesis	(
)	Closing parenthesis	)
{	Open curly brace	{
}	Close curly brace	}
:	Colon	:
;	Semi-colon	;
binNumber	Binary number	See below
boolean	Boolean	true, false
character	Single character	See below
decimalNumber	Base 10 number	See below
floatNumber	Decimal point number	See below
hexNumber	Hexadecimal number	See below
identifier	Identifier	See below
octalNumber	Octal number	See below
string	String	See below

Keyword Tokens:

bool	let	while
char	loop	value
else	main	
false	return	
fn	string	
if	struct	
float	true	
for	type	
int	where	

Comments

Comments in Small Rust can be “non-doc” or “doc” comments.

Non-doc comments follow the general C++ style of line (//) and block (/* ... */) comment forms. Rust allows any level of nesting non-doc comments. Small Rust only allows one level of nesting non-doc comments, and the nested comments can only occur within block style comments.

Doc comments also have a line style, beginning with exactly *three* slashes (///), and block doc comments (/** ... */). Line comments beginning with //! and block comments /*! ... */ are doc comments that apply to the parent of the comment, rather than the item that follows. //! comments are usually used to document modules that occupy a source file.

Isolated carriage returns (0x0D), i.e. not followed by a line feed (0x0A), are not allowed in any comments. All comments are interpreted as a form of whitespace.

In Small Rust the characters a-z, A-Z, 0-9, _, ., ~~/,~~ ~~*~~, ~~!~~, `, “, +, -, (,), {, }, :, ;, space (0x20), horizontal tab (0x09), carriage return (0x0D), and line feed (0x0A), are allowed in comments (/, *, ! have been removed).

Whitespace

Whitespace is any non-empty string containing only characters:

- space, ` `, 0x20
- horizontal tab, `\t`, 0x09
- carriage return, `\r`, 0x0D
- new line or line feed, `\n`, 0x0A

Rust is a "free-form" language, meaning that all forms of whitespace serve only to separate tokens in the grammar, and have no semantic significance.

A Rust program has identical meaning if each whitespace element is replaced with any other legal whitespace element, such as a single space character.

Identifiers

Either

- The first character is a letter.
- The remaining characters are alphanumeric or _.

Or

- The first character is _.
- The identifier is more than one character. _ alone is not an identifier.
- The remaining characters are alphanumeric or _.

Booleans

Boolean literals are the keywords, true and false.

Character Literals

Rust allows Unicode and ASCII characters, assuming Unicode characters by default. Small Rust only allows the ASCII characters, a-z, A-Z, 0-9, _, /, *, !, space (0x20), horizontal tab (\t), carriage return (\r) and line feed (\n). In Small Rust characters are surrounded by a single quote, ` (0x27), so the character h would be written `h`.

Allowed escape characters:

- space, ` `, 0x20
- horizontal tab, `\t`, 0x09
- carriage return, `\r`, 0x0D
- newline or line feed, `\n`, 0x0A

String Literals

In Rust and Small Rust strings are surrounded by double quotes (“). While Rust allows line-breaks in string literals, Small Rust only allows the characters a-z, A-Z, 0-9, _, ., /, *, !, `, “, +, -, (,), {, }, :, ;, space (0x20), horizontal tab (\t), carriage return (\r) and line feed (\n) in string literals (double quotes, “, have been removed). Empty strings are allowed.

Small Rust uses the escape character \ (0x5C), and allows the following escape sequences to appear within strings:

\n	Newline
\r	Carriage return
\t	Horizontal tab
\\	Backslash
\`	Single quote
\”	Double quote

Number Literals

Small Rust allows decimal, binary, octal and hexadecimal integers and floating point numbers. Numbers may begin with an optional + or – sign. Non-decimal integers indicate the radix via a ‘0b’, ‘0o’ and ‘0x’ for binary, octal and hexadecimal integers, respectively. Leading zeros after the radix are not allowed in integers or floating point number, unless the integer is 0 or the floating point number is less than one. At least one digit is required preceding a decimal point.

Underscores may be used to make numbers more readable. Underscores must not begin or end a number, and an underscore cannot appear on its own. Underscores do not change the value of a number. Underscores must be maintained when giving the lexeme of a number literal.

Examples:

Number literals	**Legal**	**Illegal**	**Digits Allowed**
Decimal integer	98_222	005	0-9
Hex integer	0xff, 0x0	0x0.a	0-9 and letters a-f
Octal integer	0o77	0o9	0-7
Binary integer	0b1111_0000	0b0_00	0 and 1
Floating point	123.0007700	12.E+27	0-9

Legal Characters

a-z, A-Z, 0-9, _, ., /, *, !, `, “, +, -, (,), {, }, :, ;, space (0x20), horizontal tab (\t), carriage return (\r) and line feed (\n)

Sample Small Rust Program

```
fn main() {
   let x = true;
   let y: bool = false; // with the Boolean type annotation

   // Use of Booleans in conditional expressions
   if x {
      println( "x is true");
   }
}
```

Assignment

Create a table-driven lexical analyzer for Small Rust which uses the table to identify and remove comments, and to identify tokens. Once an identifier is found, a keyword lookup table is allowed to identify keyword tokens. To create the table, begin by defining regular expressions for each token, using BNF. From that create an NFA that recognizes all tokens, a DFA, a minimal DFA, and the table.

The interface to your C# program should:

- Describe the purpose of the lexical analyzer and how to use it.
- Request a path to your scanning table and the token table, and provide a file picker so the user can select the files. Also, provide default paths so the user isn’t forced to choose the files every time.

- Allow the user to repeatedly choose a test file containing a Small Rust program. Tell the user if the scanning table and/or token tables cannot be found, and allow them to choose those files.
- If the user is in the middle of scanning a Small Rust program, and requests to scan another Small Rust program, warn the user that scanning the current program is not complete. Allow the user to choose what they want to do.
- When a Small Rust program is being scanned, allow the user (eventually this will be the parser) to repeatedly request a token, until the Small Rust program has been scanned or the user decides to quit scanning.
- After each request, your program should display the token identified and the lexeme associated with the token, and allow the user to request the next token.
- When the analyzer encounters an error in the Small Rust program it is scanning, display an error message that it helpful to the owner of the Small Rust program. Display of what character(s) caused the error, and continue parsing the program.
- Recognize the end of the program and inform the user that the end of file has been reached.

For full credit:

- Your program should be well commented. It should use descriptive variable, class and method names. Comments should describe each class and the methods within the classes. The inputs and outputs of a method should be described.
- Your program should be well designed.
- Your tables should not be hard coded.
- Paths to your tables are not hard coded.

You cannot use the Unix tool lex for this assignment.

Please submit to Moodle or me:

1. Regular expressions for the tokens
2. NFA with a single start state and multiple accept states.
3. DFA for the token table
4. Scanning table in a human readable format

A GitLab repository will be created for the programs.