COMP 442: Compiler Design

Assignment 1: Lexical Analyzer

Done by:

Abdulla Alhaj Zin, 40013496

Date:

28/01/2020

To:

Dr. Joey Paquet

Section 1: Lexical Specifications

The lexical specifications used were the same as that provided by the instructor.

There wasn't any need to change any of the specifications. However, the approach followed in identifying invalid tokens is that 1abc is considered an invalid id rather than classifying it as 1 – integer and abc – id .

For reference:

Atomic lexical elements of the language

```
      id  ::=  letter alphanum*
alphanum  ::=  letter | digit | _
 integer  ::=  nonzero digit* | 0
   float  ::=  integer fraction [e[+|–] integer]
fraction  ::=  .digit* nonzero | .0
  letter  ::=  a..z |A..Z
   digit  ::=  0..9
 nonzero  ::=  1..9
```

Operators, punctuation and reserved words

```
==    +    (    ;    if        do        read
<>    -    )    ,    then      end       write
<     *    {    .    else      public    return
>     /    }    :    while     private   main
<=    =    [    ::   class     or        inherits
>=         ]         integer   and       local
                     float     not
```

Comments

- Block comments start with **/*** and end with ***/** and may span over multiple lines.
- Inline comments start with **//** and end with the end of the line they appear in.

Section 2: Finite State Automaton

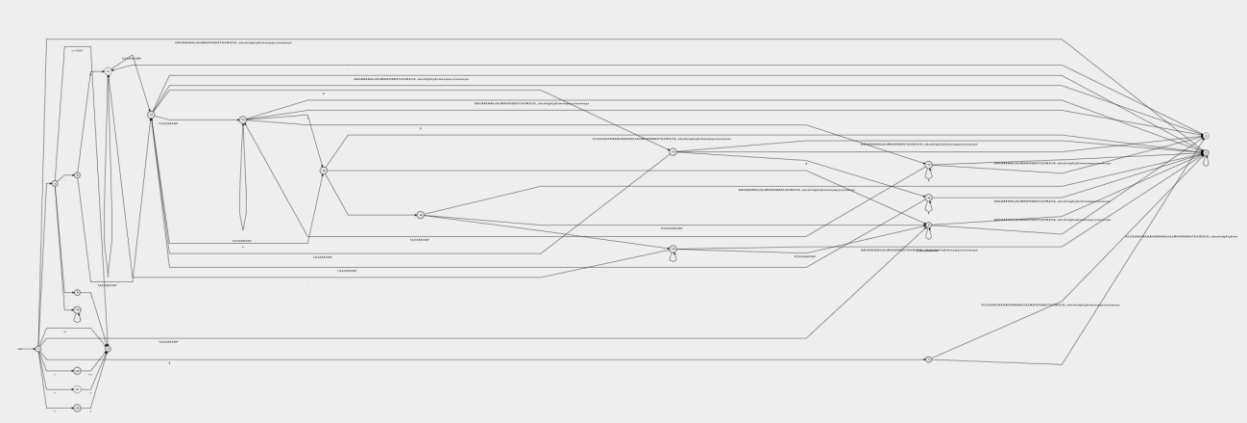
The DFA in this section is generated from the following regex:

(if|do|read|then|end|write|else|public|private|main|class|or|inherits|integer
|float|and|not|local)|==|<>|<|>|<=|>=|\+|-
|*|/|\=|\(|\)|\{|\}|\[|\]|\;|\:|\,|\.|\::|((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6
|7|8|9)*)*|0|((((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)*|0)\.(0|1|2|3|4
|5|6|7|8|9)*(1|2|3|4|5|6|7|8|9)(e(\+|-
)((1|2|3|4|5|6|7|8|9)*(0|1|2|3|4|5|6|7|8|9)*|0)|є))*(((a|b|c|d|e|f|g|h|i|j|
k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
S|T|U|V|W|X|Y|Z)*)*((0|1|2|3|4|5|6|7|8|9)*|(_)*)*)*

Which represents the language accepted by the specifications, alphabet, and grammar provided in Section 1.

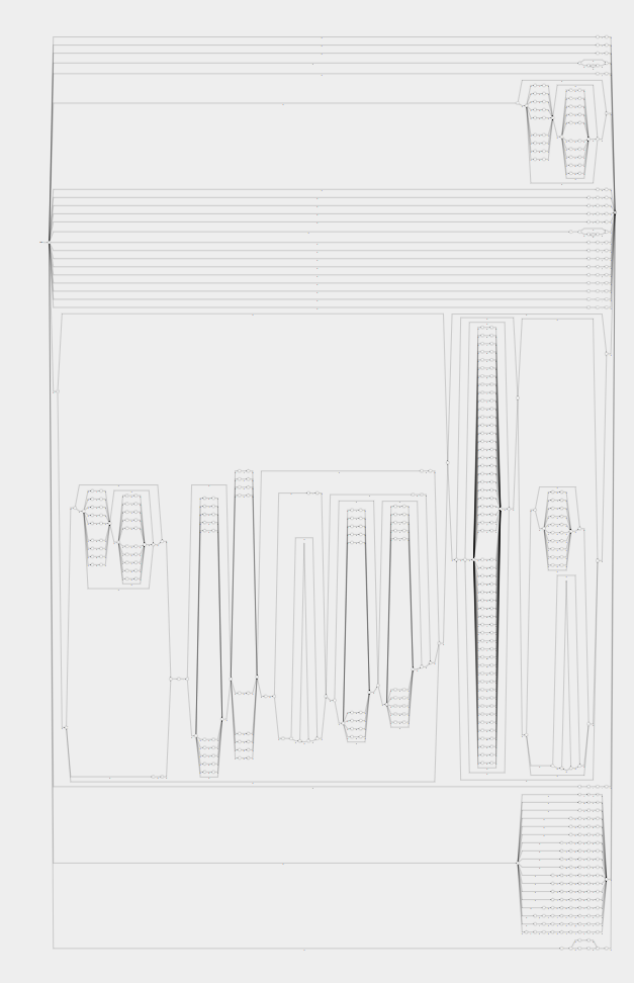
- The DFA can be found here: link
- The min-DFA can be found here: link

The min-DFA from the link:



- The NFA can be found here: link

  Below is a picture of the NFA from the link:



Section 3: Design

The design for the lexical analyzer implemented was a hand-written one. The implementation was written in C#.

The lexical analyzer can identify the following:

All reserved keywords, operands, and punctuations:

==	+	(	;	if	do	read
<>	-	)	,	then	end	write
<	*	{	.	else	public	return
>	/	}	:	while	private	main
<=	=	[	::	class	or	inherits
>=		]		integer	and	local
				float	not	

In addition, the lexer can tokenize three types:

1. Id of the form `^([a-z]+|[A-Z]+)([a-z]*[A-Z]*[0-9]*(_)*)*$`
2. Integer of the form `^[1-9]+[0-9]*$ or 0`
3. Float of the forms:
    a. `^([1-9]+[0-9]*|0)(\\.)0$`
    b. `^([1-9]+[0-9]*|0)(\\.)[0-9]*[1-9]+$`
    c. `^([1-9]+[0-9]*|0)(\\.)[0-9]*[1-9]+(e)[+|-]([1-9]+[0-9]*|0)$`

Anything that doesn't match the above regex is an invalid token of that type.

Both inline comments and block comments are supported by the TruCompiler lexical analyzer. It accepts the regular cases of inline, single line, and multiple line block comments. Moreover, it also supports edge and extreme cases such as:

a. Line that contains code and inline comments, `if//this is a comment`
b. Line that contains a comment in the middle of code,
```
if/* comment*/then do while
```
c. Multiple line block comments in the middle of recognizable code,
```
if/* comm
en
multiple line block*/then do while
```

In order to validate the above cases and other similar edge cases, 115 unit tests and test cases were implemented in C# as well. Please refer to TruCompilerTests project to see all the cases handled by the Lexical Analyzer.

The Lexical Analyzer can be broken down to:

1. Tokens: Class that handles creating the proper the token with the proper information such as the lexeme, value, location.
2. CommentsAnalyzer: Class that contains all the functions that analyze the different kinds of comments.
3. DynamicLexValidator: Class that implements the interface IValidator which includes a generic validate method that would execute the needed validation method based on the given type: Integer, float, or identifier. Those functions contain the main regex for the three types.
4. LexicalAnalyzer: the main class responsible for tokenizing a line or a list of lines and manages the actions of the above-mentioned classes.
5. Driver: Contains the compile method that would execute the lexical analyzer and then extract the tokens to two files, .outlextokens and .outlexerrors, that contain the tokens and the errors, respectively.
6. Program: main program that parses the passed arguments from the user and then runs the driver.

Section 4: Use of Tools

No external libraries were used outside of the built-in .Net core C# libraries. No special classes were used to handle any of the tokenization or the analysis; it was rather all done as a hand-written lexical analyzer.

Tools used to facilitate the implementation:

http://regexstorm.net/tester - A tool to test .net regex

https://cyberzhg.github.io/toolbox/ - A tool used to convert regex to DFA

https://github.com/abdullahzen/TruCompiler - Github was used for source control