

FACULTY OF INFORMATION TECHNOLOGY

Bachelor of Science in Informatics and Computer Science

ICS4208 – Compiler Construction

**(Lab 1 & 2)**

**98433** – **Kabecha James Boro**

ICS A

Lab 1

Link to Lab 1 - <https://github.com/JB254/98433_CCProjects>

Lab 2: Scanner Output

**Describe the output of a scanner. [5 points]**

The ***scanne****r*, is usually based on a [finite-state machine](https://en.wikipedia.org/wiki/Finite-state_machine) (FSM). It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are termed [lexemes](https://en.wikipedia.org/wiki/Lexical_analysis#Lexeme)). For example, an ***integer*** lexeme may contain any sequence of [numerical digit](https://en.wikipedia.org/wiki/Numerical_digit) characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is termed the [*maximal munch*](https://en.wikipedia.org/wiki/Maximal_munch), or *longest match*, rule). In some languages, the lexeme creation rules are more complex and may involve [backtracking](https://en.wikipedia.org/wiki/Backtracking) over previously read characters. For example, in C, one 'L' character is not enough to distinguish between an identifier that begins with 'L' and a wide-character string literal.

Scanning involves finding substrings of characters that constitute units called textual elements. These are the words, punctuation, single-and multi-character operators, comments, sequences of spaces, and perhaps line boundary characters. In its simplest form a scanner finds these substrings and classifies each as to which sort of textual element it is.

The scanner converts the high-level language input program into a sequence of tokens. The output is a sequence of tokens which is sent to the parser for syntax analysis.

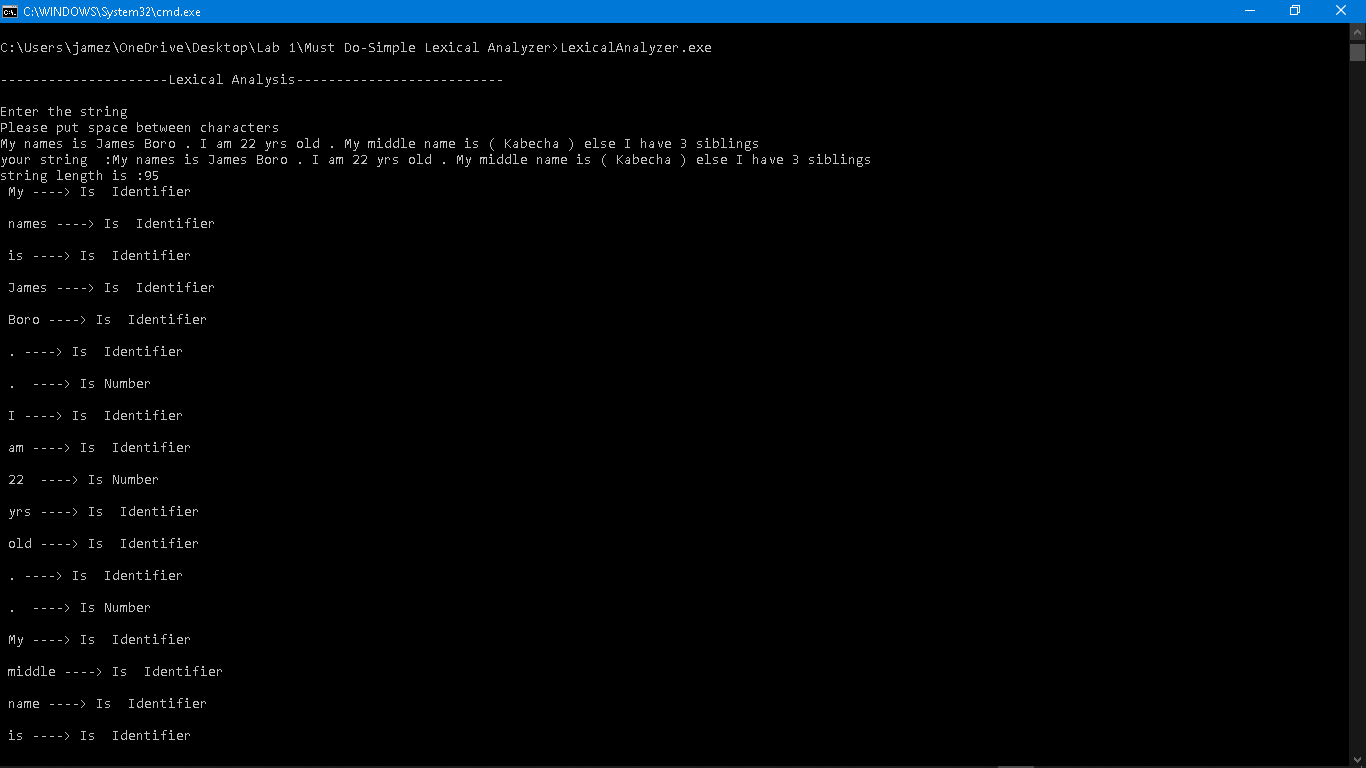
In Flex:

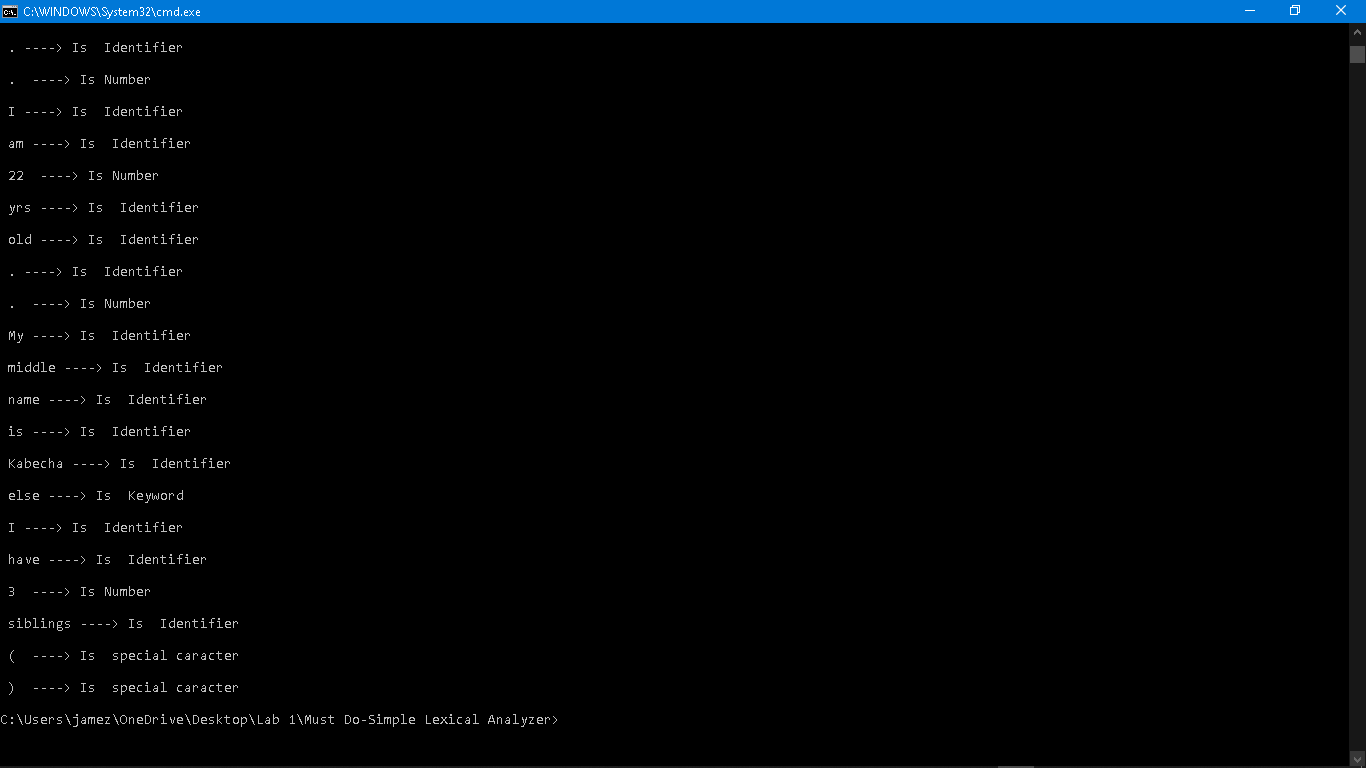
1. An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.
2. The C compiler compiles lex.yy.c file into an executable file called a.exe
3. The output file a.exe takes a stream of input characters and produces a stream of tokens.

The Lexical analyzer:

1. Through tokenization, it divides the program into valid tokens.
2. Removes comments and white space characters.
3. It also provides help in generating error messages by providing row numbers and column numbers.

## Output





**From the string, all the valid tokens are:**

‘My’ , ‘name’ , ‘is’ , ‘James’ , ‘Boro’ , ‘.’ , ‘I’ , ‘am’ , ‘22’ , ‘yrs’ , ‘old’ , ‘.’ , ‘My’ , ‘middle’ , ‘name’ , ‘is’ , ‘(’ , ‘Kabecha’ , ‘)’ , ‘else’ , ‘I’ , ‘have’ , ‘3’ , ‘siblings’

Total tokens: 26

**The string is then tokenized where:**

My is an **identifier.**

Name is an **identifier.**

Is is an **identifier.**

James is an **identifier.**

Boro is an **identifier.**

**.** is an **identifier.**

I is an **identifier.**

am is an **identifier.**

22 is a **number.**

Yrs is an **identifier.**

Old is an **identifier.**

My is an **identifier.**

Middle is an **identifier.**

Name is an **identifier.**

( is a **special character.**

Kabecha is an **identifier.**

) is a **special character.**

else is a **keyword.**

I is an **identifier.**

Have is an **identifier.**

3 is a **number.**

Siblings is an **identifier.**

= is an **operator.**

The tokenized output is then displayed.