****

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Spring, Year:2021), B.Sc. in CSE (Day/Eve)**

**Course Title:** **Compiler Lab**

**Course Code: CSE-306                 Section:DB**

**Lab Project Name:** Detection of tokens in a C program by Lexical Analysis

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Md.Nur A Neouse | 193002093 |
| **2.** | Jakirul Islam | 193002101 |
| **3.** | Nazifa Alam Nowrin | 193002103 |

**Submission Date : 27.12.2021**

**Course Teacher’s Name : Md. Atikuzzaman**

**[For Teachers use only: Don’t Write Anything inside this box]**

|  |
| --- |
| **Lab Project Status**  **Marks: …………………………………                              Signature: .....................**  **Comments: ..............................................                             Date: ..............................** |

**Chapter 1**

Introduction

* 1. **Introduction**

Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences . In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.

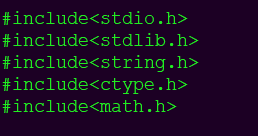
Programs that perform Lexical Analysis in compiler design are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

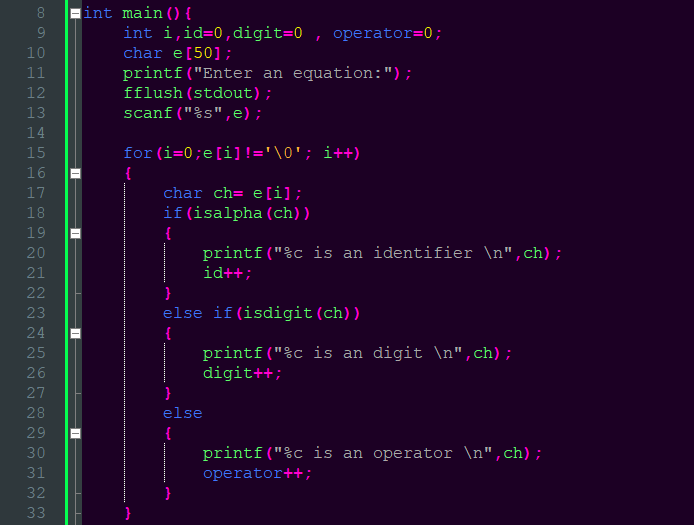
* 1. **Objectives :**

1. To program in the C programming language
2. To understand the lexical analysis phase of program compilation

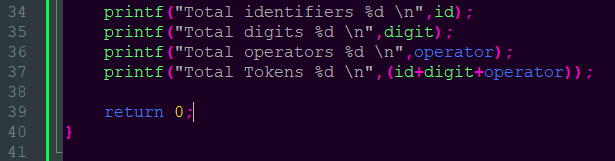
**Chapter 2**

Implementation of the project

**2.1. include all libraries :**

**2.2. Seperation of different tokens into different catagories :**

**2.3. displaying different types of tokens :**



**Chapter 3**

Performance Evaluation

**3.1. simulation procedure :**

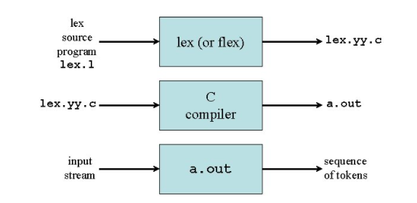
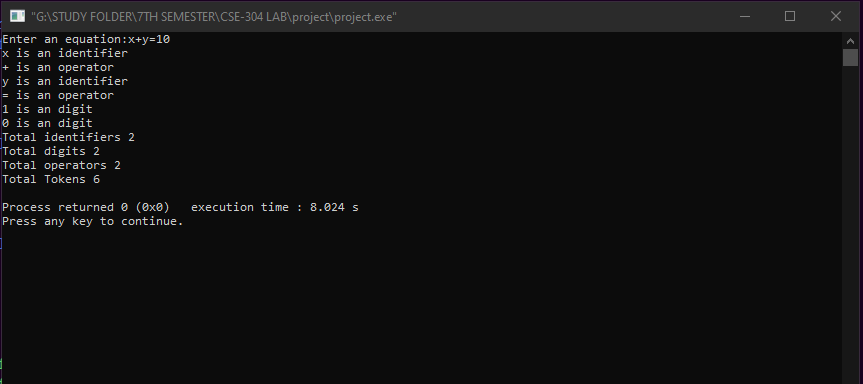


fig : simulation process of a scanner

**3.2. Results and Discussions :**

3.2.1. output :



**3.2.2. Analysis and outcome :**

Now we have the code that detects token. Now the user should enter an expression. Thus we get the desired output

**Chapter 4**

Conclusion

**4.1. Introduction**

Program result checking has been established as well suited method to construct formally correct compiler but it has never proved for real life compilers. Such a proof is necessary to establish result checking as the method of choice to implement compilers correctly. We show that the lexical analysis of compiler can be formally specified. Thereby we demonstrate that formal specification and verification techniques are able to handle real life compilers.

**4.2. Practical implications :**

1. speed up the reading of the source program.
2. looks one or more characters beyond the next lexeme.
3. there are many situations where we need to look at least one additional character ahead.

**4.3. Scope of future work :**

Helps to identify token into the symbol table

Removes white spaces and comments from the source program

Correlates error messages with the source program

Helps you to expands the macros if it is found in the source program Read input characters from the source program