SPL-1 Project Report, 2019

**Project Name** :Auto-Commenting

Course: Software Project Lab I

Course No: SE 305

Submitted by  
Md. Nadim AhmedBSSE Roll No. : 1028BSSE Session: 2017-2018

Supervised by  
 Dr. Kazi Muheymin-Us SakibDesignation: Professorof Information Institute TechnologyUniversity of Dhaka

Submission Date29-05-2018

**Table of Contents**

**1.Introduction …................................................................................4**

**1.1.Background study .........................................................................5-7**

**1.2.Challenges .................................................................................... 7-8**

**2.Project Overview ................................................................................ 8**

**3.User Manual .................................................................................10-15**

**4.Conclusion ........................................................................................ 16**

**5.References ......................................................................... ................16**

**Appendix............................................................................................... 16**

**1.Introduction**

# Code comments are an integral part of a code. They improve program comprehension .The lack of code comments is a common problem to understand a unknown code. Therefore, it is beneficial to generate code comments automatically. In this project, it is proposed a general approach to generate code comments automatically by analyzing codes.

# The project implementation is started with collecting the first year students’ code of BSSE 10th batch .Then, the project will be implemented with two main tasks:-

# 1. Commenting on functional blocks of an uncommented source code.

# 2. Commenting after finding similarities between uncommented and commented source code.

# 

# 1.1. Background study

# Lexical analysis

Lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning). A program that performs lexical analysis may be termed a *lexer*, *tokenizer*, or *scanner*, though *scanner* is also a term for the first stage of a lexer. A lexer is generally combined with a parser which together analyze the syntax of programming . A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Common token names are

* identifier: names the programmer chooses;
* keyword: names already in the programming language;
* separator (also known as punctuators): punctuation characters and paired-delimiters;
* operator: symbols that operate on arguments and produce results;
* literal: numeric, logical, textual, reference literals;
* comment: line, block.

**Syntax**

In programming, syntax refers to the rules that specify the correct combined sequence of symbols that can be used to form a correctly structured program using a given programming language. Programmers communicate with computers through the correctly structured syntax, semantics and grammar of a programming language.

**Semantics**

Semantics is a linguistic concept separate from the concept of syntax, which is also often related to attributes of computer programming languages. The idea of semantics is that the linguistic representations or symbols support logical outcomes, as a set of words and phrases signify ideas to both humans and machines.

**Abstract Syntax Tree**

An abstract syntax tree (AST), or just syntax tree, is a [tree](https://en.wikipedia.org/wiki/Directed_tree) representation of the [abstract syntactic](https://en.wikipedia.org/wiki/Abstract_syntax) structure of [source code](https://en.wikipedia.org/wiki/Source_code) written in a [programming language](https://en.wikipedia.org/wiki/Programming_language). Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-related details. For instance, grouping [parentheses](https://en.wikipedia.org/wiki/Bracket#Parentheses) are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.



Figure 1: Abstract Syntax Tree

**1.2 Challenges**

Implementing a new software solution carries with it a number of challenges. The process can be overwhelming, confusing and lenthy. Implementing this project there are lot of challenges that I have faced. Some of them are:-

▪Handling large code for the first time  
▪Learning and understanding algorithm  
▪Reading Abstract Syntax Tree  
▪Reading Syntactically similarity  
▪Implementing Abstract Syntax Tree  
▪Matching similarity of two code

**2. Project Overview**

I have divided my whole project into four different parts. They are:-  
•Commenting on functional blocks  
•Implementation of syntactically similarity  
•Implementation of semantically similarity

•Implementation of Abstract Syntax Tree

3. User Manual

3.1 **Commenting on functional blocks**

Input Function:-

void partition(int \*arr,int low,int high)

{

int pivot,temp,i,j;

pivot=arr[high];

for(i=low-1,j=low;j<high;j++)

{

if(arr[j]<pivot)

{

i+=1;

temp=arr[j];

arr[j]=arr[i];

arr[i]=temp;

}

}

temp=arr[high];

arr[high]=arr[i+1];

arr[i+1]=temp;

return i+1;

}

**Output Function:-**

void partition(int \*arr,int low,int high)//The name of the function is \*\*\*partition\*\*\*.The return type of the function is \*\*\*void\*\*\*.The parameters of the function respectively are (\*arr,low,high) and the data type of the parameters respectively are (int,int,int).

{

int pivot,temp,i,j;

pivot=arr[high];

for(j=low-1,j=100;j++)//This is a for loop which starts with 1 and ends with till the value .The for loop is incremental. The variable name is \*\*\*i \*\*\* and the data type of the variable is \*\*\*int\*\*\*.

{

if(arr[j]<pivot)

{

i+=1;

temp=arr[j];

arr[j]=arr[i];

arr[i]=temp;

}

}

temp=arr[high];

arr[high]=arr[i+1];

arr[i+1]=temp;

return i+1;

}

**3.2 Syntactically Similarity**

This similarity is implemented by using string of two source codes. Then the both strings are compared with edit distance algorithm.

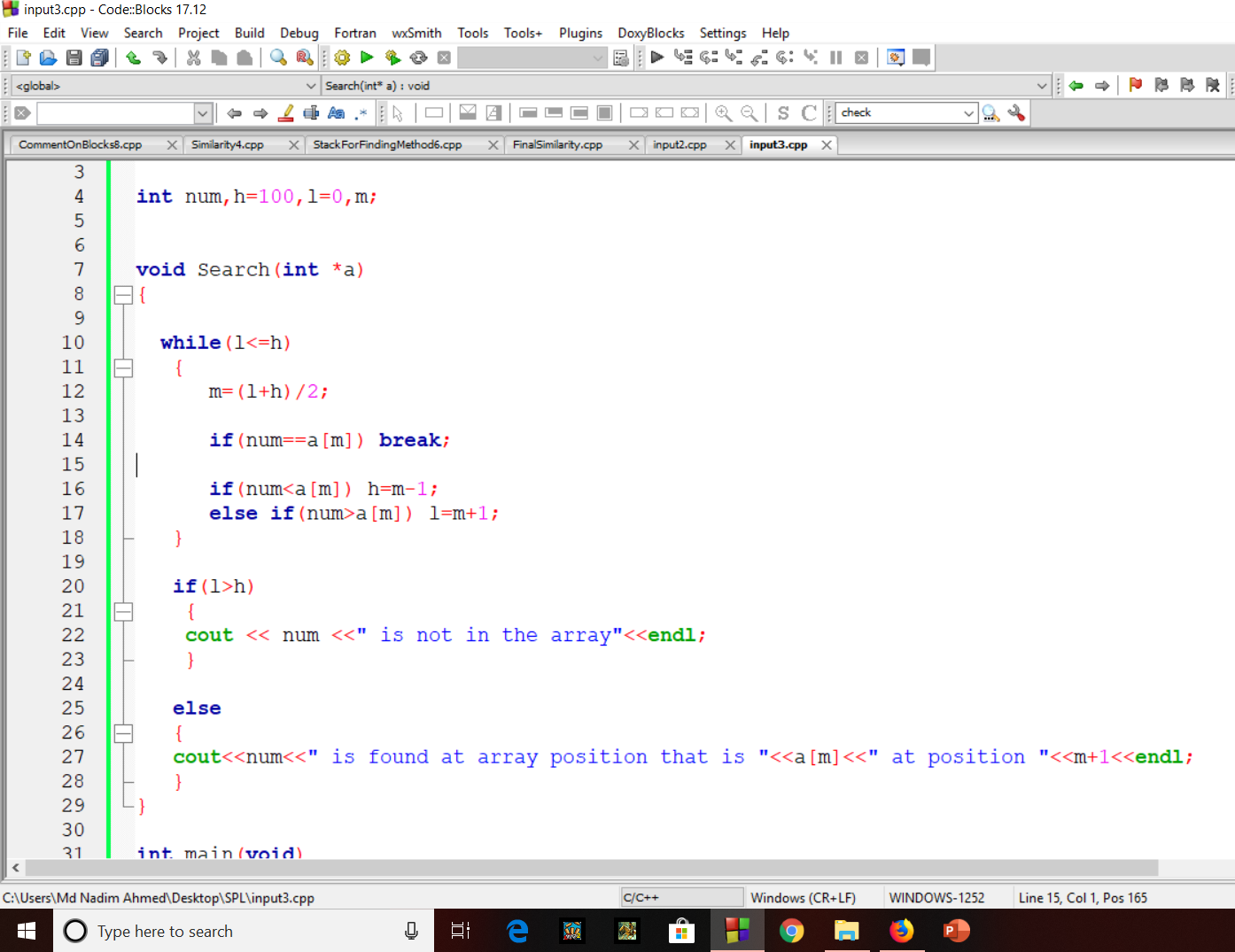


Figure 2:User manual: Input 1

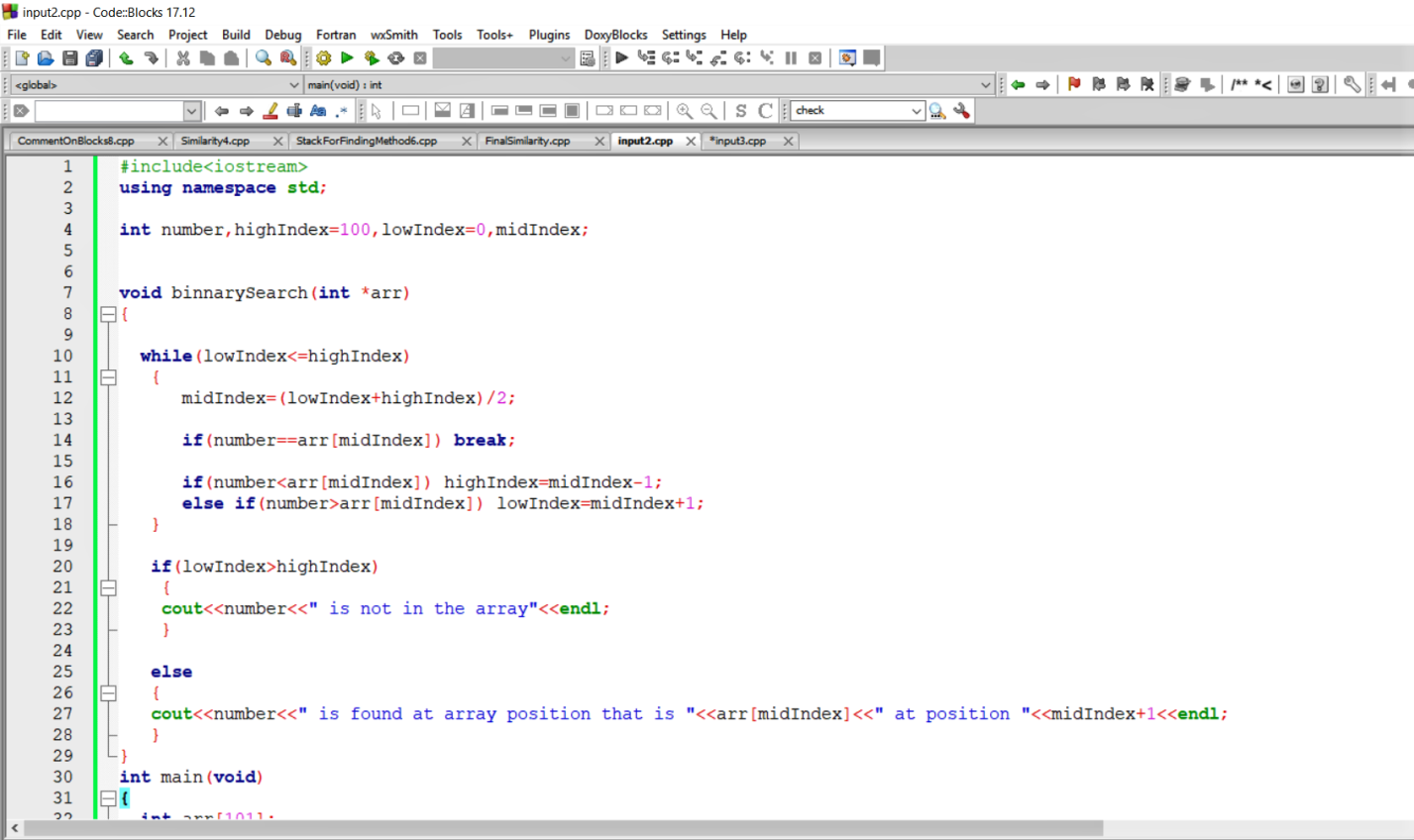


Figure 3:User manual: Input 2

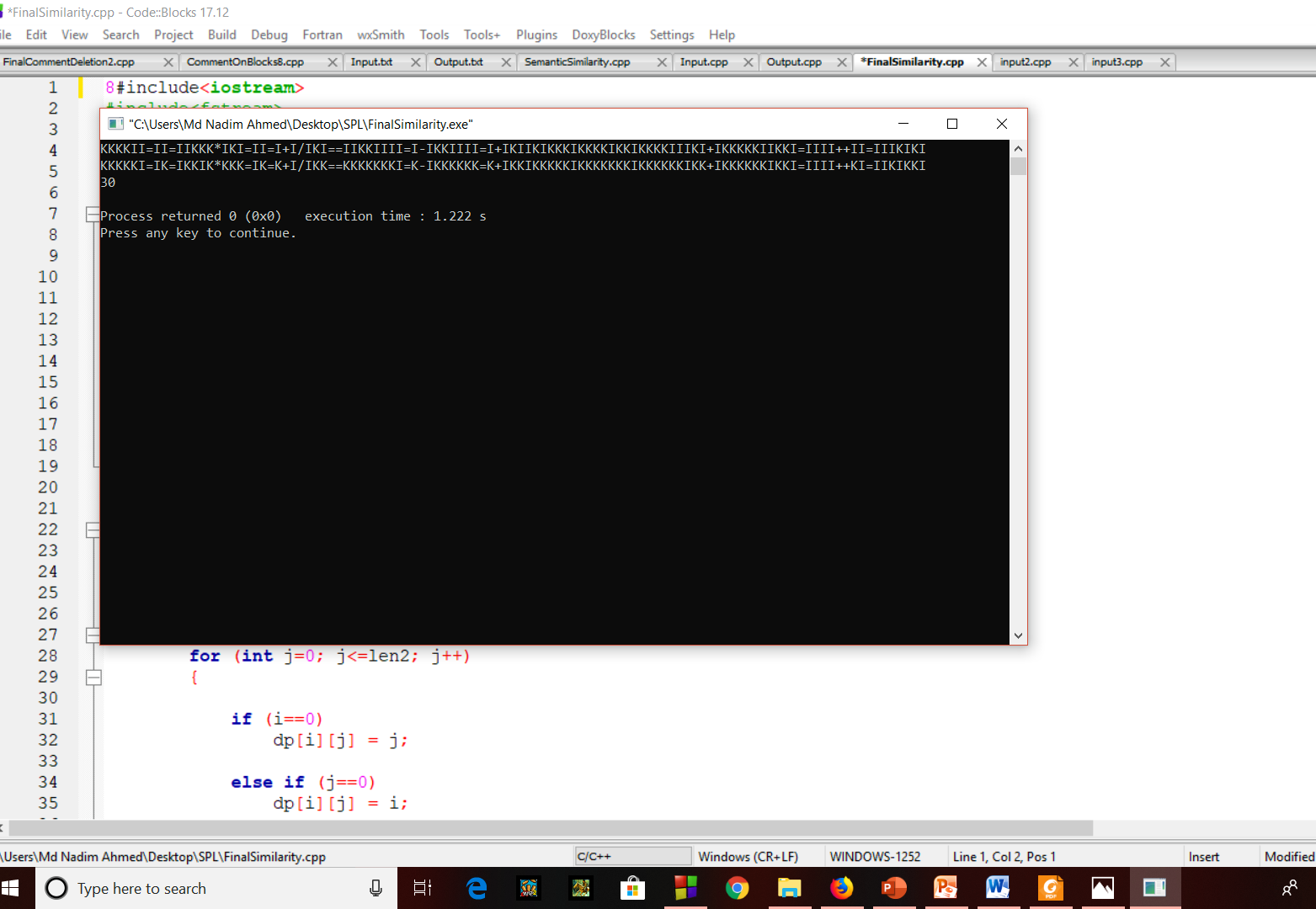


Figure 4:Output of Syntactically Similarity

**3.3 Sementically Similarity**

This similarity is implemented by splitting the functions of two codes. Then the functions are concatenated with a main function format. By using a function named “system”, the codes are run from the terminal and got output.If the output of both codes are same for same input.Then, we can tell that the codes are syntactically similar.

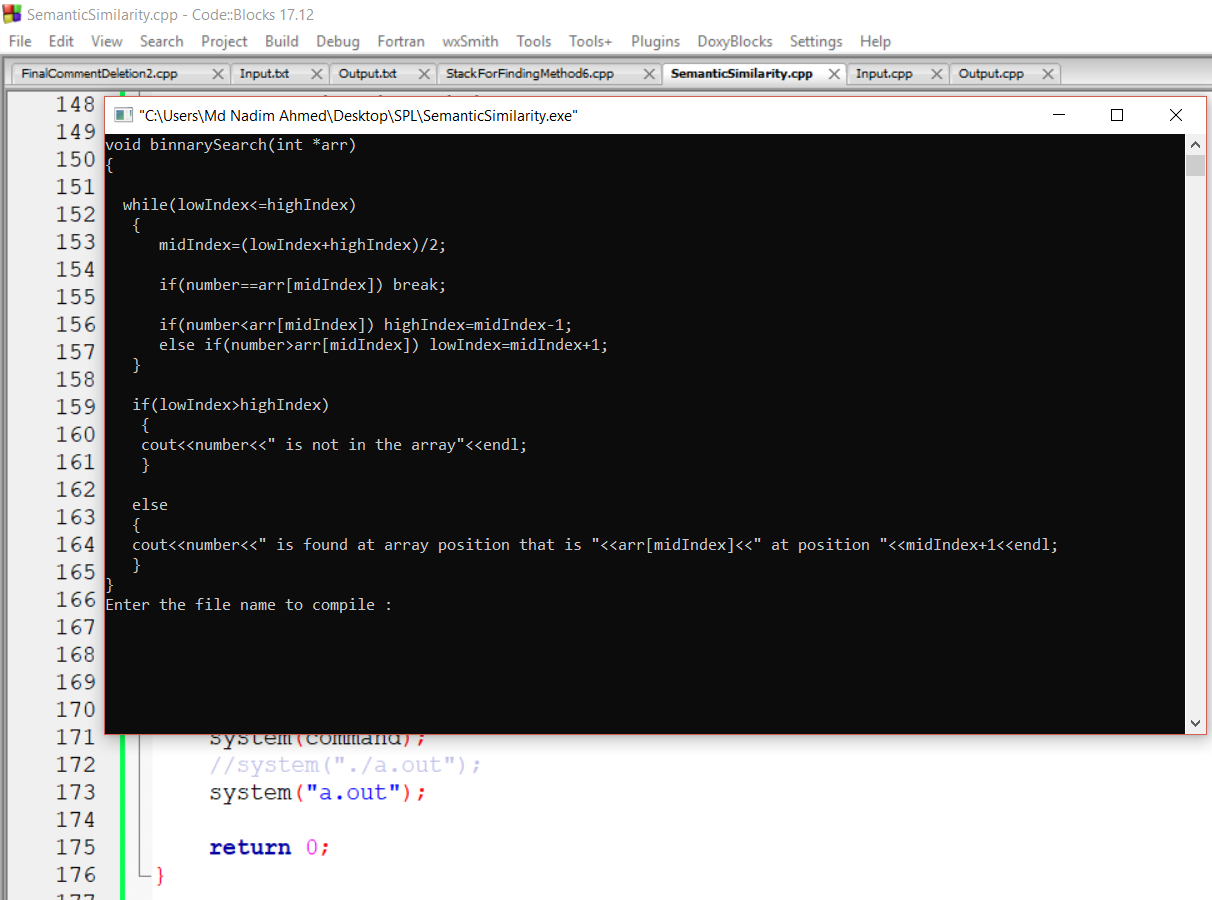


Figure 5:User manual: Output 1

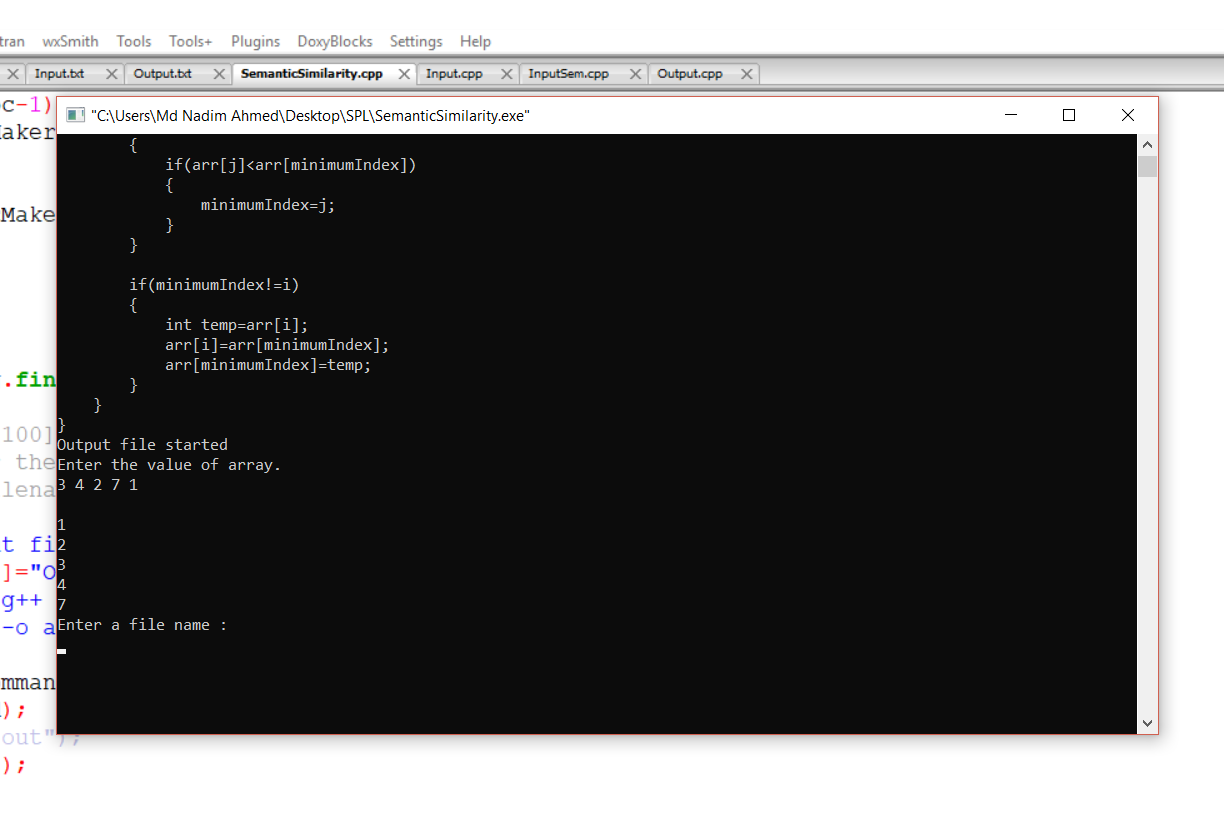


Figure 6:User manual: Output 1

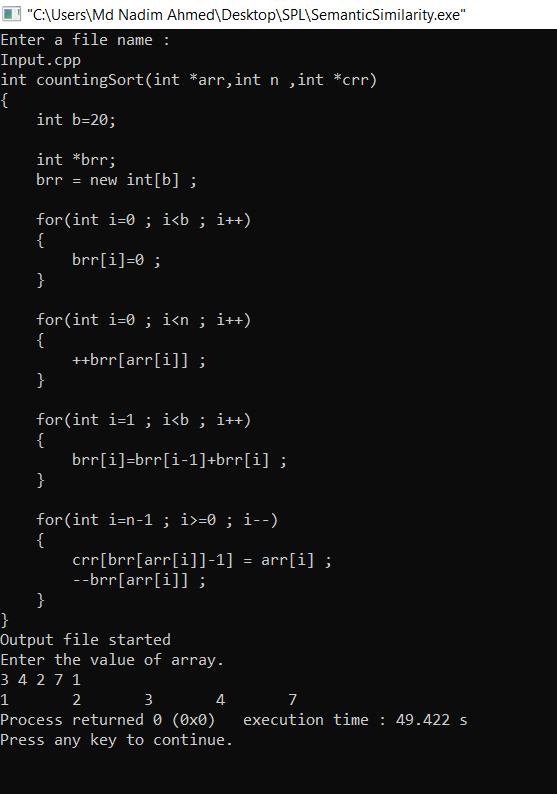


Figure 7:User manual: Output 2

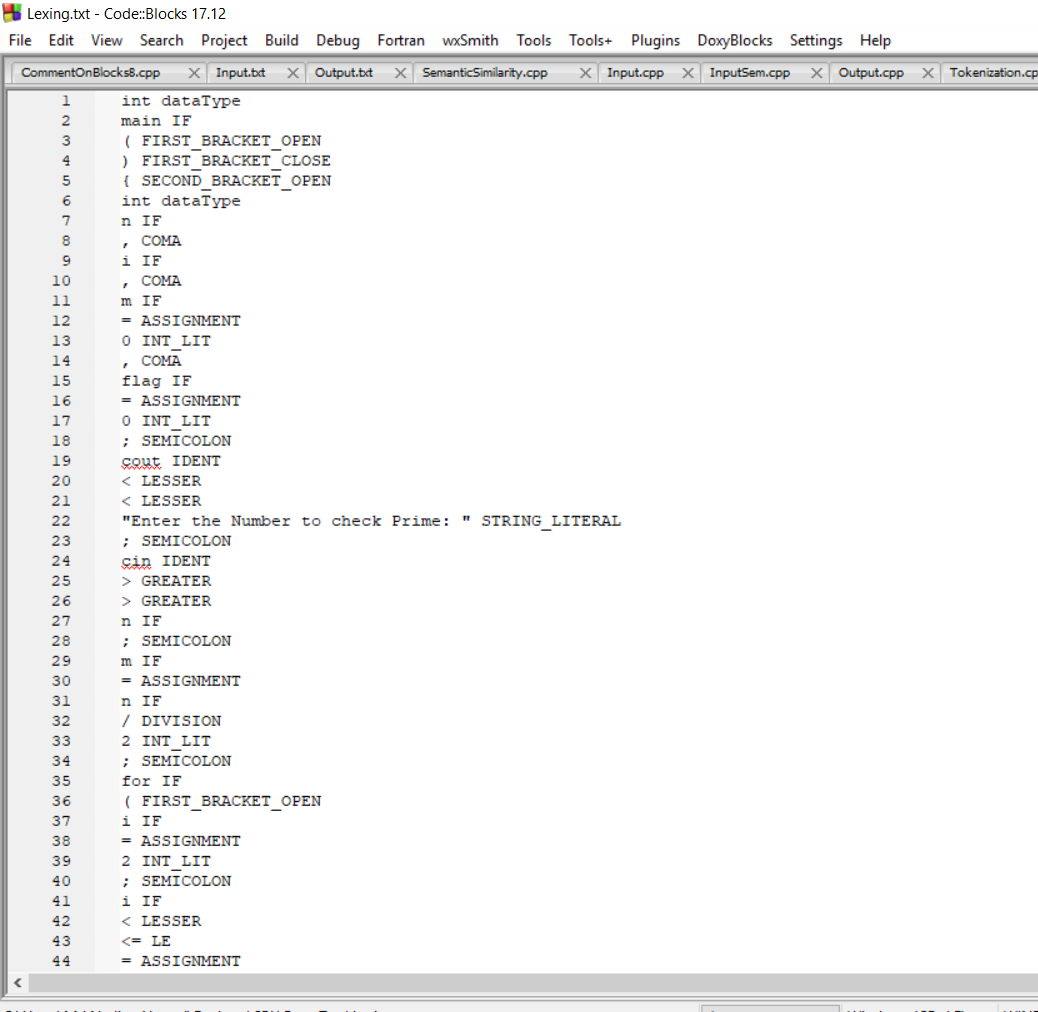


Figure 8:Lexical Analysis

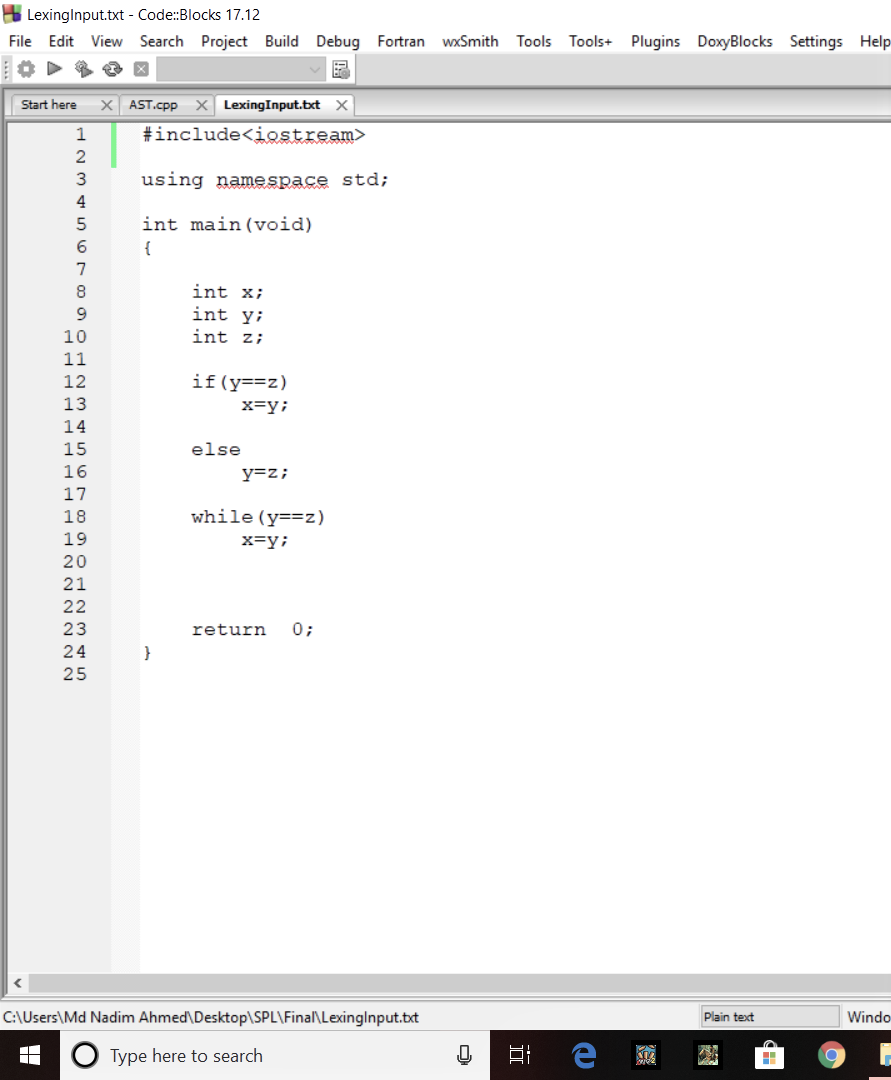
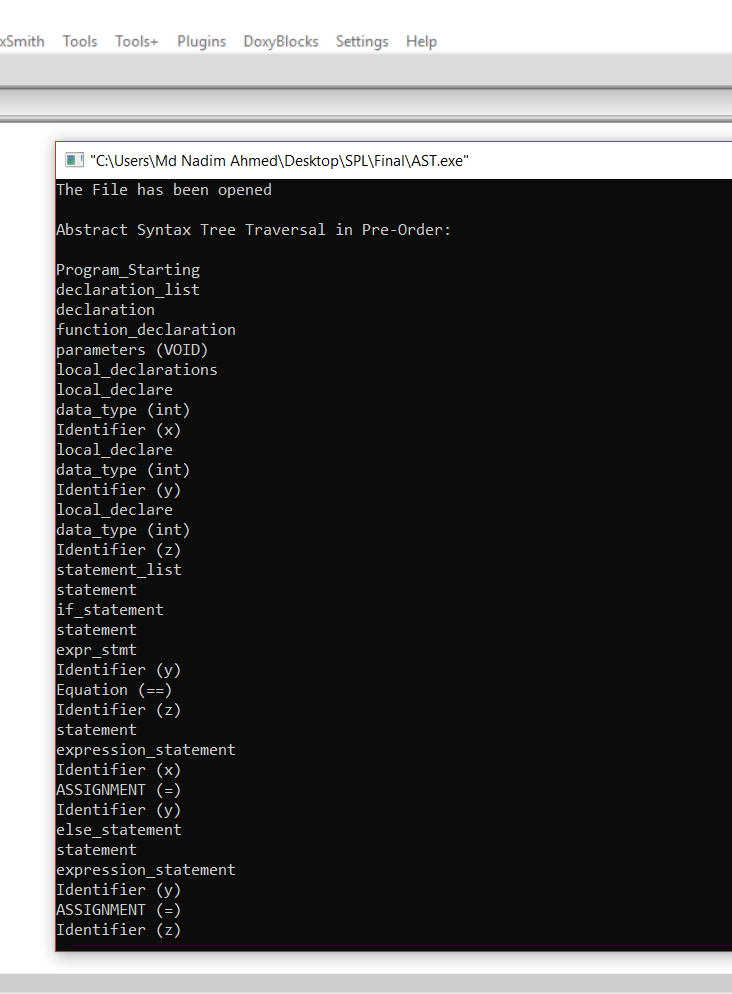


Figure 9:Input File



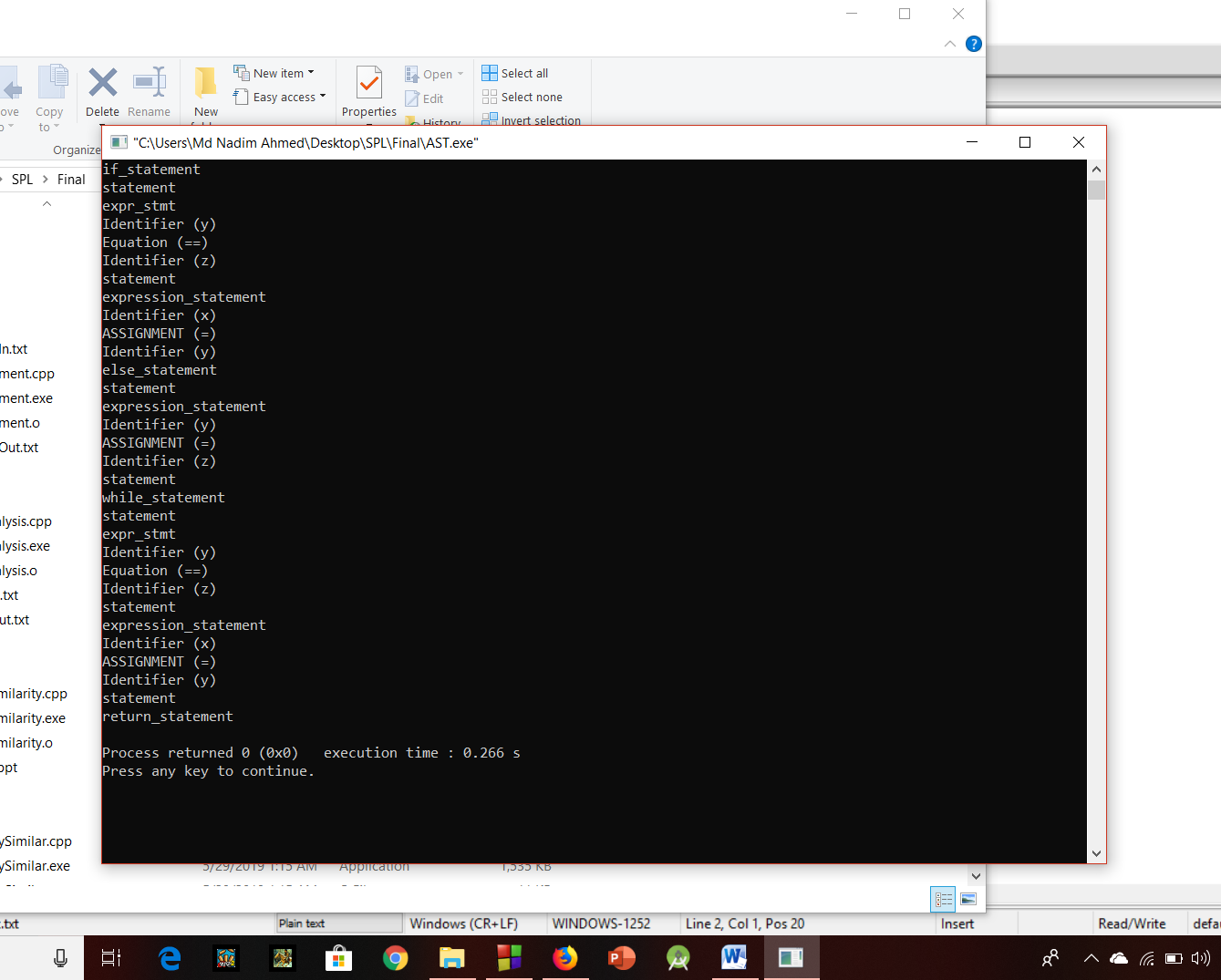


Figure 9:Abstract Syntax Tree Traversal.

**4.Conclusion**

Implementing this project helps me to improve my coding skill and I have learned to handle large code for the first time. I hope it will help me to deal with difficulties in future. This project was quiet challenging and I gained a lot of experience from it. I want to thank my supervisor for guiding me a lot during this project.

5. References

<https://ece.uwaterloo.ca/~lintan/publications/clocom-saner15.pdf>

<https://en.wikipedia.org/wiki/Lexical_analysis>

<https://www.techopedia.com/definition/3959/syntax>

<https://en.wikipedia.org/wiki/Abstract_syntax_tree>

**Appendix**In this project, I have implemented auto-commenting in a source code for first year student’s code. In future, I want to implement this project for a large code.