ALL IN ONE COMPILER(Semantic Analyser)

A MINI PROJECT REPORT

Submitted by

VASU GOEL [RA2011033010094]
CHANDAN MADHAV GOGOI [RA2011033010134]

Under the guidance of

Dr. J Jeyasudha

(Assistant Professor, Department of Computational Intelligence)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING
with specialization in Software Engineering

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

May-2023

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

BONAFIDE CERTIFICATE

Certified that this project report "**ALL IN ONE COMPILER(Semantic Analyzer**" is the bonafide work of "**VASU GOEL [RA20110033010094], CHANDAN MADHAV GOGOI [RA2011033010134]**," of III Year/VI Sem B.Tech(CSE) who carried out the mini project work under my supervision for the course **18CSC304J- Compiler Design** in SRM Institute of Science and Technology during the academic year2023(Even Semester).

[illegible]
11/5/24

[illegible]

SIGNATURE
Faculty In-Charge
Dr. J Jeyasudha
Assistant Professor
Department of Computational Intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and Head ,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

ABSTRACT

Syntax analysis, also known as parsing, is a critical component in the compilation process of programming languages. It is the process of analyzing the syntax structure of the source code to ensure that it conforms to the grammar rules of the language. This is a crucial step in the compilation process because syntax errors can prevent a program from being compiled or executed correctly.

The primary function of a parser during syntax analysis is to create a parse tree, which represents the structure of the code. The parse tree is a hierarchical representation of the code and can be used to generate intermediate code or machine code. The parse tree is also used to identify syntax errors in the source code. If the parser detects any syntax errors, it will report them to the programmer, who can then correct them before attempting to compile the code.

Syntax analysis is a crucial step in the development of robust and efficient software systems. Proper syntax analysis ensures that the code is correct and conforms to the language's grammar rules. This leads to more efficient code and reduces the likelihood of bugs and errors in the final software product.

Overall, syntax analysis plays a vital role in the software development process and is essential for producing reliable and efficient software systems.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Syntax analysis, also known as parsing, is a critical component in the development and compilation of programming languages. It is a process that involves analyzing the structure of source code to ensure that it conforms to the grammar rules of the language. Syntax analysis plays a crucial role in the compilation process because it ensures that the code is correct and can be compiled and executed properly.

During the syntax analysis process, a parser is used to analyze the syntax structure of the code and create a parse tree, which represents the structure of the code. The parse tree is a hierarchical representation of the code and can be used to generate intermediate code or machine code. It is also used to identify syntax errors in the source code. If the parser detects any syntax errors, it will report them to the programmer, who can then correct them before attempting to compile the code..

One Syntax analysis is a vital step in the development of robust and efficient software systems. Proper syntax analysis ensures that the code is correct and conforms to the language's grammar rules. This leads to more efficient code and reduces the likelihood of bugs and errors in the final software product. In addition, syntax analysis plays a crucial role in ensuring that the code is maintainable and extensible.

1.2 PROBLEM STATEMENT

The problem statement of this project is to develop a robust and efficient syntactic analyzer for a given programming language. The syntactic analyzer should be capable of performing the following tasks:

1. **T Complexity of Programming Languages:** One of the major challenges in syntax analysis is the complexity of programming languages. As new features and functionalities are added to programming languages, new syntax rules are introduced, making the syntax analysis process more complicated.

2. **Computationally Expensive**: Syntax analysis can be computationally expensive, particularly for large and complex programs. This can lead to slower compilation times, which can be frustrating for programmers who need to make frequent changes to their code. Therefore, efficient algorithms and parsing techniques must be developed to minimize the time required for syntax analysis.

3. **Error Reporting**: Syntax errors can be difficult to identify and report, particularly for novice programmers who may not be familiar with the language's syntax rules. Parsers must provide clear and informative error messages to help programmers identify and correct syntax errors. Additionally, the syntax analysis process must be flexible enough to handle different programming styles and code formats..

4. **Compatibility with Various Tools and Environments**:
   Syntax analysis is a critical component in the development of robust and efficient software systems. However, it must be compatible with various tools and environments used in software development, such as compilers and integrated development environments (IDEs). Therefore, syntax analysis must be standardized and adaptable to various development environments to ensure that it can be seamlessly integrated into the software development process.

5. **Error Reporting:** The syntactic analyzer should provide informative error messages that clearly identify the nature and location of syntactic errors in the source code.

1.3 OBJECTIVES

The objective of the syntactic analyzer in compiler design is to perform a thorough analysis of the source code and enforce semantic rules and constraints of the programming language. Its primary goal is to ensure the correctness and integrity of the program by detecting and reporting syntactic errors that cannot be captured during the earlier lexical and syntactic analysis phases. The syntactic analyzer aims to achieve the following objectives:

1. **Validating Source Code**:
   The primary objective of syntax analysis is to validate the source code and ensure that it conforms to the grammar rules of the programming language. The parser checks the syntax of the code and reports any syntax errors to the programmer.

2. **Creating Abstract Syntax Tree (AST):**
   Another objective of syntax analysis is to create an Abstract Syntax Tree (AST) of the program. The AST is a data structure that represents the program's syntax in a hierarchical format, making it easier to analyze and manipulate the program's structure.

3. **Enabling Code Optimization:**
   Syntax analysis also enables code optimization. By analyzing the syntax of the code, the parser can identify redundant code and optimize it to improve the program's efficiency.

4. **Providing Useful Error Messages:**
   Syntax analysis aims to provide useful error messages to programmers to help them identify and correct syntax errors in their code. The error messages should be clear and informative, providing details about the specific syntax error and how to fix it.

5. **Supporting Different Programming Styles:**
   Another objective of syntax analysis is to support different programming styles. Programmers may have different preferences when it comes to writing code, and syntax analysis should be flexible enough to accommodate different programming styles while still ensuring that the code adheres

6. **Ensuring Syntax Compliance:**

   The primary objective of syntax analysis is to ensure that the source code complies with the syntax rules of the programming language. The parser checks the syntax of the code and reports any syntax errors to the programmer.

7. **Providing Accurate Error Messages:**
   Syntax analysis should provide accurate and informative error messages to programmers. The error messages should clearly identify the specific syntax error and provide guidance on how to correct it.

By achieving these objectives, the syntactic analyzer contributes to the overall quality, reliability, and correctness of the compiled program. It acts as a crucial component in the compiler pipeline, bridging the gap between the lexical analyzer and the subsequent code generation phase.

1.4 Need for Syntax Analyser

Syntax analysis is a critical process in software development that ensures the accuracy, maintainability, and efficiency of code. By checking the syntax of the code, syntax analysis helps identify and correct syntax errors that could lead to bugs, crashes, or vulnerabilities. Syntax analysis is especially important when developing large, complex software systems, where errors can be difficult to locate and fix. One of the main benefits of syntax analysis is that it facilitates code maintenance. By identifying syntax errors, developers can easily locate and fix issues in their code

Following things are done in Syntax Analysis:

1. **Ensuring Code Accuracy**: Syntax analysis checks the syntax of the code and ensures that it follows the language's rules and standards. This process helps ensure the code's accuracy, leading to fewer errors and more reliable software.

2. **Facilitating Code Maintenance:**
   Syntax analysis can identify syntax errors, making it easier for developers to maintain and update their code. Developers can easily locate and fix syntax errors, which can help reduce development time and improve code quality.

3. **Enabling Code Optimization:**
   Syntax analysis can identify redundant code, which can be removed to improve the program's efficiency and performance. This optimization process can result in faster execution times and improved user experience.

4. **Improving Code Readability**: Syntax analysis can help improve the code's readability by identifying and correcting syntax errors. Code that is easy to read and understand is more maintainable and less prone to errors.

5. **Enhancing Software Security:** Syntax analysis can identify vulnerabilities in the code that could be exploited by attackers. By detecting and correcting syntax errors, developers can help improve the security of their software and reduce the risk of cyber-attacks.

6. **Supporting Interoperability:** Syntax analysis helps ensure that software written in different languages can communicate with each other. By following language standards and rules, developers can ensure that their software is compatible with other systems and applications.

Finally, syntax analysis supports interoperability by ensuring that software written in different languages can communicate with each other. By following language standards and rules, developers can ensure that their software is compatible with other systems and applications, allowing for seamless integration and operation. In conclusion, syntax analysis is a necessary process in software development that helps ensure code accuracy, maintainability, and efficiency. By identifying and correcting syntax errors, developers can create software that is secure, reliable, and easy to maintain.

1.5 REQUIREMENTS SPECIFICATION

Hardware Requirements:

Processor: A modern multi-core processor (e.g., Intel Core i5 or higher) to handle the compilation process efficiently.
Memory (RAM): A minimum of 8 GB is recommended
Storage: Adequate storage space for the source code, compiler tools, libraries, and any additional resources.(A minimum of 128 GB is recommended)

Operating System: Windows / linux distributions / macOS

Development Environment:
Integrated Development Environment (IDE): Visual Studio Code, Eclipse, or JetBrains IntelliJ IDEA
Version Control: Git to manage source code, track changes, and collaborate with other developers if applicable

Programming Languages and Tools:

Compiler Design Language: C++Back-end Framework: Flask Front-end Framework: Boostrap

Documentation and Reporting:

Document Preparation Software: Used word processing software like Microsoft Word for creating the compiler design report.

CHAPTER 2

2.1 What does a compiler need to know during syntax analysis?

During syntax analysis, a compiler needs to know several details about the source code to check whether it follows the language's rules and standards. Here are some of the things that a compiler needs to know during syntax analysis:

The language's grammar rules: A compiler needs to know the grammar rules of the language in which the source code is written. These rules define the syntax of the language and specify how the language's elements can be combined to form valid statements.

The context-free grammar of the language: Context-free grammar (CFG) is a formal language model used to describe the syntax of a programming language. A compiler needs to know the CFG of the language to check whether the syntax of the source code is valid. CFG specifies the language's syntax in terms of production rules that define how elements can be combined.

The lexical structure of the language: The lexical structure of the language defines how the language's elements, such as keywords, identifiers, literals, and operators, are formed. A compiler needs to know the lexical structure of the language to tokenize the source code and group the tokens into meaningful constructs.

The rules for statement formation: The rules for statement formation specify how language constructs can be combined to form valid statements. A compiler needs to know these rules to ensure that statements are syntactically correct

The language-specific rules: Each programming language has its own specific rules and features that a compiler needs to be aware of. For example, some languages may allow nested functions, while others may not. A compiler needs to know these language-specific rules to correctly parse and interpret the source code.

In summary, a compiler needs to know the language's grammar rules, the context-free grammar of the language, the lexical structure of the language, the hierarchy of expressions, the rules for statement formation, and the language-specific rules during syntax analysis to ensure that the source code is syntactically correct.

Limitations of CFGs.

Context free grammars deal with syntactic categories rather than specific words. The declare before use rule requires knowledge which cannot be encoded in a CFG and thus CFGs cannot match an instance of a variable name with another. Hence the we introduce the attribute grammar framework.

Syntax directed definition

This is a context free grammar with rules and attributes. It specifies values of attributes by associating semantic rules with grammar productions.

Syntax Directed Translation

This is a compiler implementation method whereby the source language translation is completely driven by the parser.
The parsing process and parse tree are used to direct semantic analysis and translation of the source program.
Here we augment conventional grammar with information to control semantic analysis and translation. This grammar is referred to as attribute grammar.
The two main methods for SDT are Attribute grammars and syntax directed translation scheme

Attributes

An attribute is a property whose value gets assigned to a grammar symbol. Attribute computation functions, also known as semantic functions are functions associated with productions of a grammar and are used to compute the values of an attribute. Predicate functions are functions that state some syntax and the static semantic rules of a particular grammar.

Types of Attributes

Type -These associate data objects with the allowed set of values.
Location - May be changed by the memory management routine of the operating system.
Value -These are the result of an assignment operation.

Name-These can be changed when a sub-program is called and returns.
Component - Data objects comprised of other data objects. This binding is represented by a pointer and is subsequently changed.

Synthesized Attributes.

These attributes get values from the attribute values of their child nodes.
They are defined by a semantic rule associated with the production at a node such that the production has the non-terminal as its head.

An example

S → PQR
S is said to be a synthesized attribute if it takes values from its child node (P,Q,R).

Inherited Attributes.

These attributes take values from their parent and/or siblings.
They are defined by a semantic rule associated with the production at the parent such that the production has the non-terminal in its body.
They are useful when the structure of the parse tree does not match the abstract syntax tree of the source program.
They cannot be evaluated by a pre-order traversal of the parse tree since they depend on both left and right siblings.

An example;
S → ABC

A can get its values from S, B and C.
B can get its values from S, A and C
C can get its values from A, B and S

Expansion

This is when a non-terminal is expanded to terminals as per the provided grammar.

Reduction

This is when a terminal is reduced to its corresponding non-terminal as per the grammar rules.
Note that syntax trees are parsed top, down and left to right

Attribute grammar

This is a special case of context free grammar where additional information is appended to one or more non-terminals in-order to provide context-sensitive information.
We can also define it as SDDs without side-effects.
It is the medium to provide semantics to a context free grammar and it helps with the specification of syntax and semantics of a programming language.
When viewed as a parse tree, it can pass information among nodes of a tree.

An Example

Given the CFG below;
E → E + T { E.value = E.value + T.value }
The right side contains semantic rules that specify how the grammar should be interpreted.
The non-terminal values of E and T are added and their result copied to the non-terminal E.

An Example

Consider the grammar for signed binary numbers
number → signlist
sign → + | −
list → listbit | bit
bit → 0 | 1

We want to build an attribute grammar that annotates Number with the value it represents.
First we associate attributes with grammar symbols

The attribute grammar

Defining an Attribute Grammar

Attribute grammar will consist of the following features;

- Each symbol X will have a set of attributes A(X)
- A(X) can be;
    - Extrinsic attributes obtained outside the grammar, notable the symbol table
    - Synthesized attributes passed up the parse tree
    - Inherited attributes passed down the parse tree.
- Each production of the grammar will have a set of semantic functions and predicate functions(may be an empty set
- Based on the way an attribute gets its value, attributes can be divided into two categories; these are, Synthesized or inherited attributes.

Abstract Syntax Trees(ASTs)

These are a reduced form of a parse tree.
They don't check for string membership in the language of the grammar.
They represent relationships between language constructs and avoid derivations.

Properties of abstract syntax trees.

- Good for optimizations.
- Easier evaluation.
- Easier traversals.
- Pretty printing(unparsing) is possible by in-order traversal.
- Postorder traversal of the tree is possible given a postfix notation.

Implementing Syntax Actions during Recursive Descent parsing.

During this parsing there exist a separate function for each non-terminal in the grammar. The procedures will check the lookahead token against the terminals it expects to find. Recursive descent recursively calls procedures to parse non-terminals it expects to find. At certain points during parsing appropriate semantic actions that are to be performed are implemented.

Roles of this phase.

- Collection of type information and type compatibility checking.
- Type checking.
- Storage of type information collected to a symbol table or an abstract syntax tree.
- In case of a mismatch, type correction is implemented or a semantic error is generated.
- Checking if source language permits operands or not.

CHAPTER 3

SYSTEM ARCHITECTURE AND DESIGN

3.1 FRONT-END DESIGN:-

For the Front-End Framework we have use Bootstrap, overview of Bootstrap:-

1. **Bootstrap Framework:** Bootstrap is also the name of a popular front-end development framework used to build responsive and mobile-first websites. The Bootstrap framework consists of various components and tools that aid in web development, including:

2. **HTML/CSS Components:** Bootstrap provides a collection of pre-designed HTML and CSS components, such as buttons, forms, navigation bars, and grids. These components can be easily integrated into web pages to ensure consistency and responsiveness.

3. **JavaScript Plugins:** Bootstrap offers a set of JavaScript plugins that add functionality and interactivity to web pages. These plugins include features like carousels, modals, tooltips, and dropdown menus.

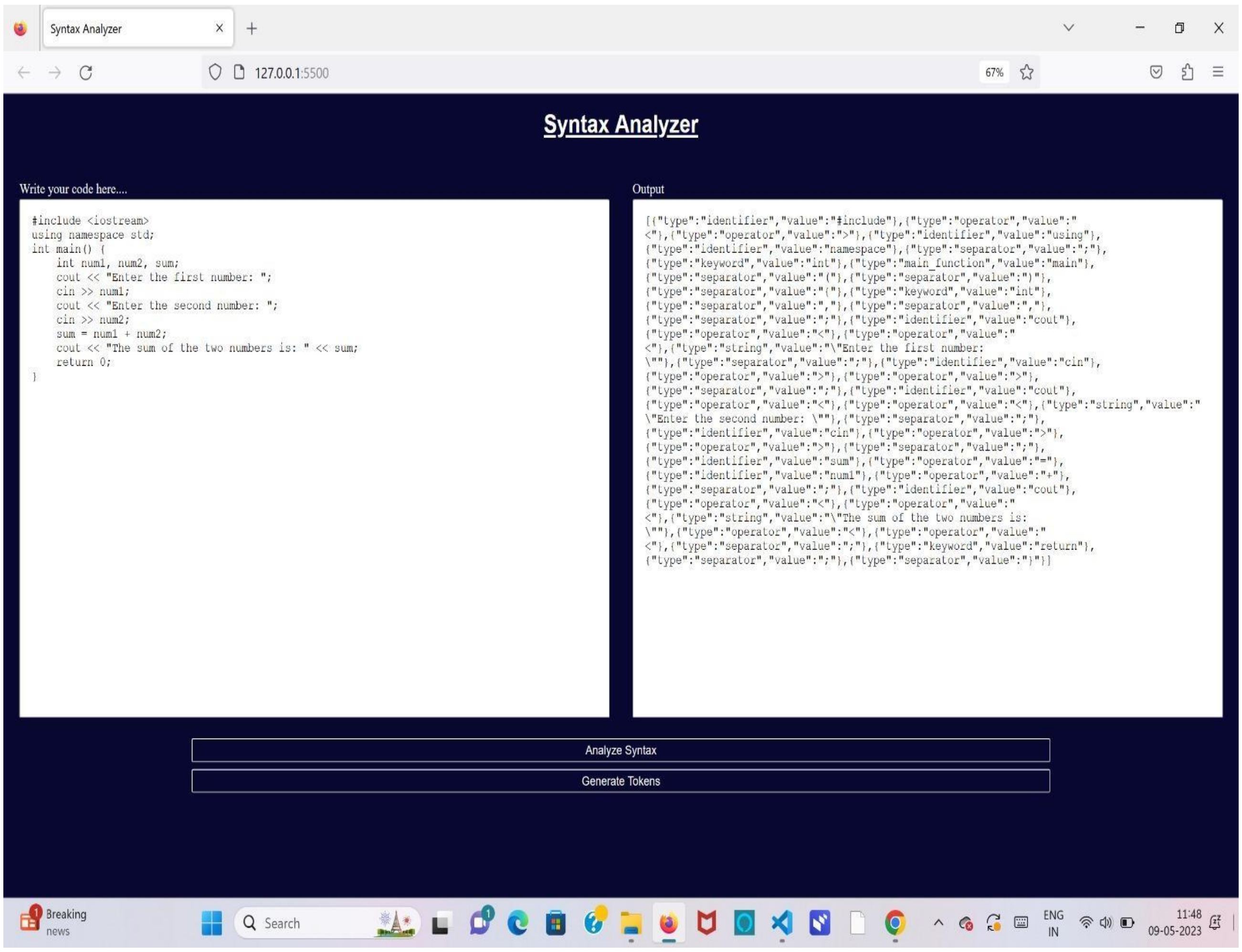
4. **Responsive Grid System:** Bootstrap includes a responsive grid system that enables developers to create flexible and responsive layouts for web pages. The grid system helps in achieving a consistent look and feel across different devices and screen sizes.

5. **Bootstrapping a Compiler or Interpreter:** In the context of compiler or interpreter design, bootstrapping refers to the process of implementing a compiler or interpreter for a programming language using the same language itself. The bootstrapping process typically involves the following stages:

6. **Initial Compiler/Interpreter:** A basic version of the compiler or interpreter is written in a different language (often a lower-level language or an existing language). It is used to compile or interpret the subsequent versions of the compiler or interpreter.

7. **Self-Compilation:** The initial compiler or interpreter is used to compile or interpret an updated version of itself written in the target language. Iterative Refinement: The process is repeated, using each new version to compile or interpret a more advanced version until the final compiler or interpreter is achieved.

Overall, the components of bootstrap vary depending on the specific context, such as the system or software application being bootstrapped. It can involve components like the bootloader, operating system kernel, application initialization, HTML/CSS components, JavaScript plugins, and self-compilation in the case of compiler or interpreter design.



```
#include <iostream>
using namespace std;
int main() {
    int num1, num2, sum;
    cout << "Enter the first number: ";
    cin >> num1;
    cout << "Enter the second number: ";
    cin >> num2;
    sum = num1 + num2;
    cout << "The sum of the two numbers is: " << sum;
    return 0;
}
```

```
[{"type":"identifier","value":"#include"},{"type":"operator","value":"
<"},{"type":"operator","value":">"},{"type":"identifier","value":"using"},
{"type":"identifier","value":"namespace"},{"type":"separator","value":";"},
{"type":"keyword","value":"int"},{"type":"main_function","value":"main"},
{"type":"separator","value":"("},{"type":"separator","value":")"},
{"type":"separator","value":"{"},{"type":"keyword","value":"int"},
{"type":"separator","value":","},{"type":"separator","value":","},
{"type":"separator","value":";"},{"type":"identifier","value":"cout"},
{"type":"operator","value":"<"},{"type":"operator","value":"
<"},{"type":"string","value":"\"Enter the first number:
\""},{"type":"separator","value":";"},{"type":"identifier","value":"cin"},
{"type":"operator","value":">"},{"type":"operator","value":">"},
{"type":"separator","value":";"},{"type":"identifier","value":"cout"},
{"type":"operator","value":"<"},{"type":"operator","value":"<"},{"type":"string","value":"
\"Enter the second number: \""},{"type":"separator","value":";"},
{"type":"identifier","value":"cin"},{"type":"operator","value":">"},
{"type":"operator","value":">"},{"type":"separator","value":";"},
{"type":"identifier","value":"sum"},{"type":"operator","value":"="},
{"type":"identifier","value":"num1"},{"type":"operator","value":"+"},
{"type":"separator","value":";"},{"type":"identifier","value":"cout"},
{"type":"operator","value":"<"},{"type":"operator","value":"
<"},{"type":"string","value":"\"The sum of the two numbers is:
\""},{"type":"operator","value":"<"},{"type":"operator","value":"
<"},{"type":"separator","value":";"},{"type":"keyword","value":"return"},
{"type":"separator","value":";"},{"type":"separator","value":"}"}]
```

Figure 3.1

FRONT-END ARCHITECTURE DESIGN:-

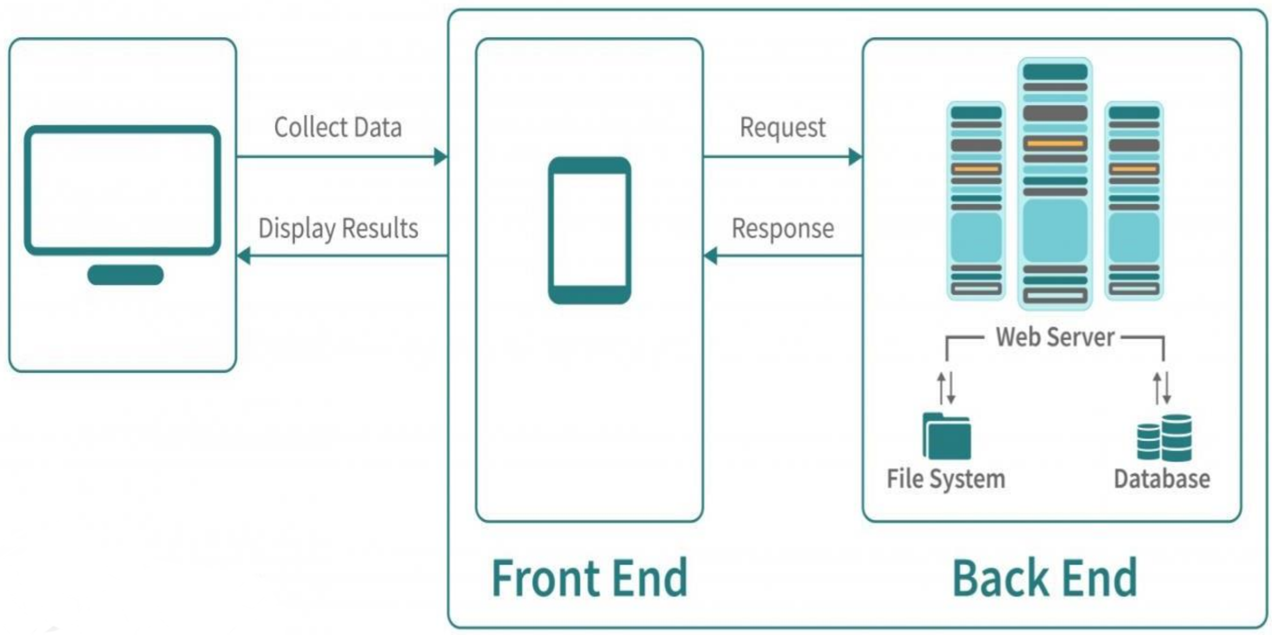


Figure 3.2

3.2 BACK-END DESIGN:-

Flask is a popular back-end web framework for Python. It provides a lightweight and flexible approach to building web applications. When using Flask as a back-end framework, it typically consists of the following key components:

1. **Routing:** Flask allows you to define URL routes and associate them with specific functions or methods. These routes determine how the application responds to different URLs or HTTP methods (GET, POST, etc.). By using decorators or URL patterns, you can map routes to corresponding view functions or class methods.
2. **Views:** Views in Flask are Python functions or class methods that handle requests and generate responses. They receive data from requests, process it, and return appropriate responses. Views can render templates, return JSON data, redirect to other URLs, or perform other actions based on the application's requirements.

3. **Templates:** Flask integrates with template engines (such as Jinja2) to separate the presentation logic from the application logic. Templates allow you to dynamically generate HTML pages by

incorporating data and logic. Flask provides support for template inheritance, variable substitution, control structures, and other template features.

4. **Forms:** Flask provides utilities for handling HTML forms, including form validation, data retrieval, and rendering. It allows you to define form classes, specify validation rules, and generate HTML form elements. Flask also supports form submission handling, including processing form data and handling validation errors.

5. **Middleware:** Flask allows the use of middleware, which are components that intercept and process requests and responses before they reach the view functions. Middleware can perform tasks such as authentication, logging, error handling, or modifying the request/response objects.

6. **Extensions:** Flask has a rich ecosystem of extensions that provide additional functionality and integrate with various services. These extensions cover areas such as database integration, authentication, session management, caching, API development, and more. Extensions can be easily integrated into Flask applications to enhance their capabilities.

7. **Configuration:** Flask allows you to configure various aspects of the application, such as database connections, debugging options, logging settings, and more. Configuration can be done through environment variables, configuration files, or programmatically in the application code.

These components of Flask form the foundation for developing back-end web applications. They provide the necessary tools and abstractions to handle routing, views, templates, database interactions, form handling, middleware, and configuration. Flask's simplicity and flexibility make it a popular choice for building web applications of varying sizes and complexities.

BACK END ARCHITECTURE DESIGN:-

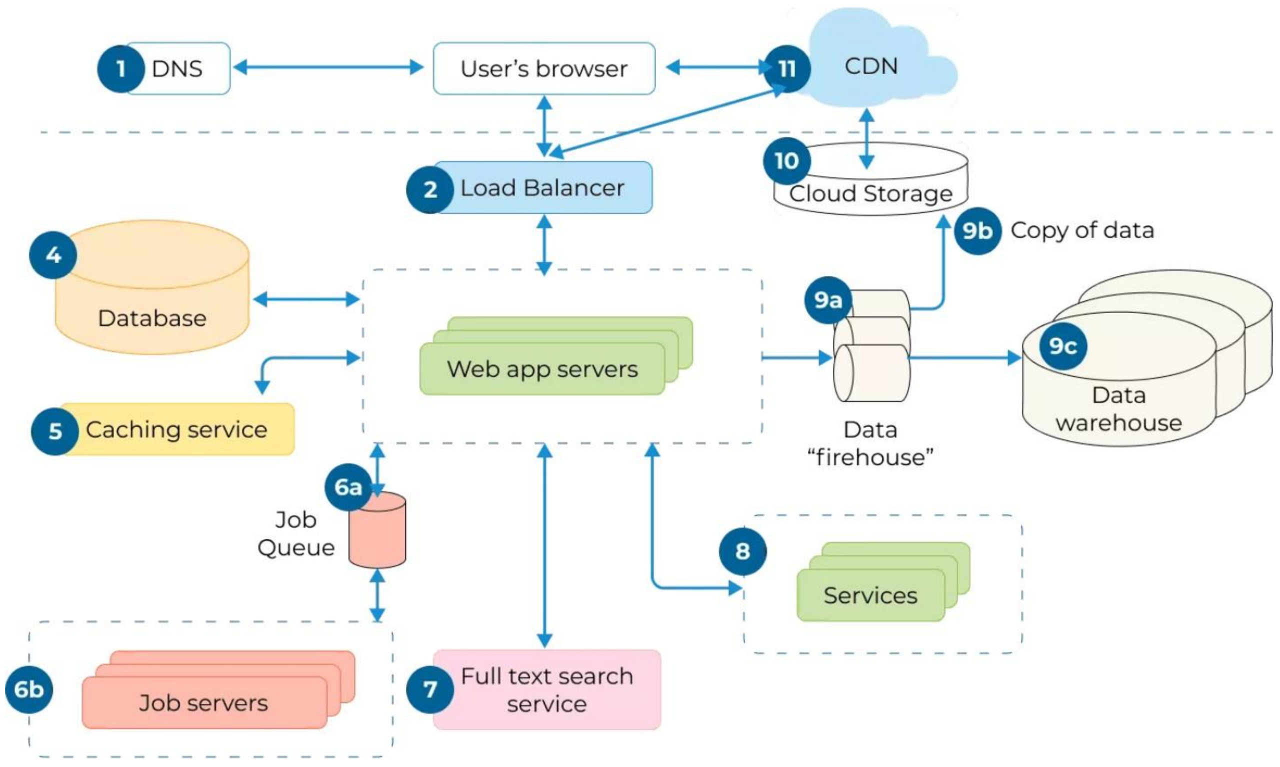


Figure 3.3

SYNTAX ANALYZER ARCHITECTURE DESIGN:-

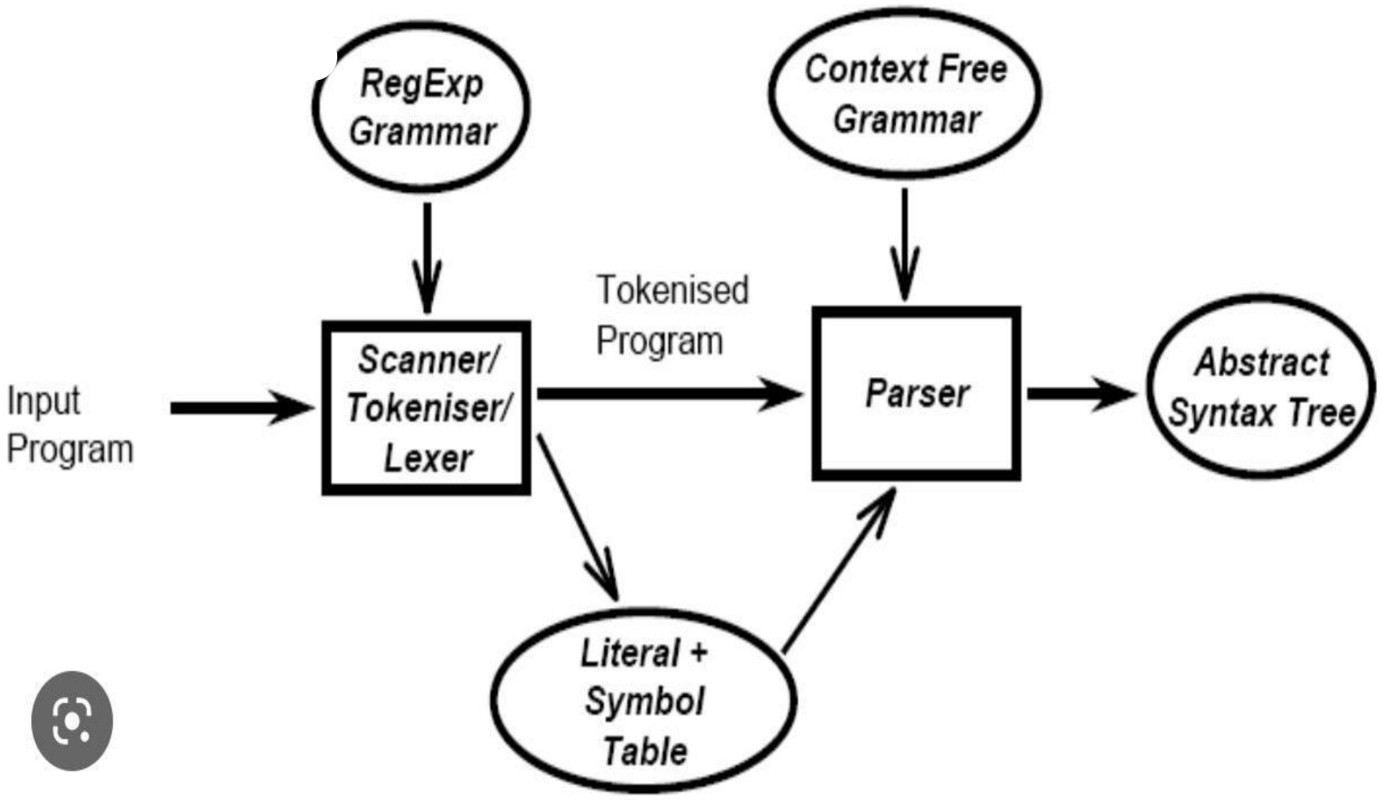


Figure 3.4

- Syntax Analyzer: This is the main component that encompasses the entire syntax analysis process
- Lexer: The lexer component is responsible for tokenizing the input source code. It takes the source code as input and breaks it down into individual lexical units or tokens. The lexer identifies keywords, identifiers, operators, literals, and other language-specific constructs, and provides these tokens to the parser.
- Parser: The parser component performs the syntax analysis of the source code based on a grammar or set of rules. It receives the tokens from the lexer and constructs a parse tree or an abstract syntax tree (AST) representing the syntactic structure of the source code.
- Symbol Table: The symbol table component is responsible for managing symbols encountered during the syntax analysis phase. It stores information about variables, functions, and other identifiers defined in the source code.

CHAPTER 4

The requirement to run the script -

To run a semantic analyzer code built using Flask and JavaScript, you will need the following requirements:

1. **A compatible version of Python:** Flask is a web framework for Python, so you will need to have Python installed on your computer. The specific version of Python required may vary depending on the version of Flask and other dependencies used in the code.

2. **Flask and its dependencies:** Flask is a third-party library for Python, and it has several dependencies that must be installed to use it. These dependencies may include Werkzeug, Jinja2, and others. You can use a package manager like pip to install these dependencies.

3. **A web server:** Flask is a web framework, and it requires a web server to run. You can use the built-in development server that comes with Flask, or you can use a production-ready web server like Apache or Nginx.

4. **A browser:** The JavaScript code used in the semantic analyzer will be executed in the user's browser, so you will need a modern web browser like Chrome, Firefox, or Safari to view and interact with the analyzer.

5. **Code editor or IDE:** To edit the Flask and JavaScript code, you will need a code editor or integrated development environment (IDE) that supports Python and JavaScript.

6. **Syntax analysis libraries:** Depending on the specific requirements of your semantic analyzer, you may need to install additional libraries or tools for semantic analysis, such as Natural Language Processing (NLP) libraries or machine learning frameworks.

Overall, running a semantic analyzer built using Flask and JavaScript requires a working environment with the appropriate dependencies, a web server, and a compatible browser. With these requirements met, you can run and interact with the semantic analyzer to analyze and understand the meaning and context of programming code.

CHAPTER 5

CODING AND TESTING

CODING:-

1. **index.js**

```
var keywords = ["auto", "break",
"case", "char", "const", "continue",
"default",
   "double", "else", "enum",
"extern", "float", "goto", "int",
"long", "register",
   "return", "short", "signed",
"sizeof", "static", "struct",
"typedef", "union",
   "unsigned", "void", "volatile"];
var operators = ["+", "-", "*", "/",
"=", "<", ">", "<=", ">=", "==",
"!=",
   "&&", "||", "!", "++", "--", "+=",
"-=", "*=", "/="];
var separators = ["(", ")", "{", "}",
"[", "]", ";", ","];
var func = ["main"];
var inBuildFunctions = ["for",
"do", "if", "printf", "scanf",
"switch", "while"];
```

```
var BracketStack = [];
var mainAvailable = false;

function analyzeCode(code) {
   var tokens = [];
   var currentToken = "";
   var insideString = false;
   var insideComment = false;

   for (var i = 0; i <
code.length; i++) {
      var c = code.charAt(i);
      var nextC =
code.charAt(i + 1);

      if (insideString) {
         if (c == '"') {
            insideString = false;
            tokens.push({ type:
"string", value: currentToken
+ c });
            currentToken = "";
         } else {
            currentToken += c;
         }
      } else if
(insideComment) {
         if (c == "\n") {
            insideComment =
false;
            tokens.push({ type:
"comment", value:
currentToken });
            currentToken =    "";
         }
else if (c == '/') {
         if (nextC == '/') {
            insideComment =
true;
            currentToken += c +
nextC;
            i++;
         } else {
            tokens.push({ type:
"operator", value: c });
            currentToken = "";
         }
      }
```

```
            else if (operators.indexOf(c) >= 0) {
        tokens.push({ type: "operator", value: c });
        currentToken = "";
    } else if (separators.indexOf(c) >= 0) {
        tokens.push({ type: "separator", value: c });
        currentToken = "";
    } else if(func.indexOf(currentToken + c) >= 0){
        mainAvailable = true;
        tokens.push({ type: "main_function", value: currentToken + c });
        currentToken = "";
    } else if (/\s/.test(c)) {
        if (currentToken != "") {
            if (keywords.indexOf(currentToken) >= 0) {
                tokens.push({ type: "keyword", value: currentToken });
            }else if(inBuildFunctions.indexOf(currentToken) >= 0){
                tokens.push({ type: "inBuildFunctions", value: currentToken});
            }else {
                tokens.push({ type: "identifier", value: currentToken });
            }
            currentToken = "";
        }
    }
    else {
        currentToken += c;
    }
} if (currentToken != "") {
    if (keywords.indexOf(currentToken) >= 0) {
        tokens.push({ type: "keyword", value: currentToken });
    } else {
        tokens.push({ type: "identifier", value: currentToken });}
}return tokens;
    }
    function checkBracketStack(value) {
     if ((value == ')' && BracketStack[BracketStack.length - 1] == '(')
    || (value == '}' && BracketStack[BracketStack.length - 1] == '{')
       || (value == ']' && BracketStack[BracketStack.length - 1] == '[')) {
                  BracketStack.pop();
    }
else if ( value == '(' || value == '{' || value == '[' || (value == ')' || value == '}' || value == ']') &&
                              BracketStack.length == 0){
                 BracketStack.push(value);
                    }
            }
          function syntaxAnalyze(tokens) {
                        for (var i = 0; i < tokens.length; i++) {
                               var token = tokens[i];
                                console.log(JSON.stringify(token.value));
                        switch (token.type) {
                                    case 'separator':
                                     checkBracketStack(token.value);
                         break;
                                      case 'keywords':
                         break;
                         default:
                         break;
              }
          }
       function Convert_button() {
                 var code = document.getElementById("code").value;
                          if(code.length>0){
                                document.getElementById("output").value = Count_Lines(code) +
                                      " Lines of code checked\n\n";
                                 document.getElementById("output").value +=
                                    /* JSON.stringify(analyzeCode(code)); */

                            }
                       else
                            document.getElementById("output").value = "Code block is empty!"

                           }

             }
```

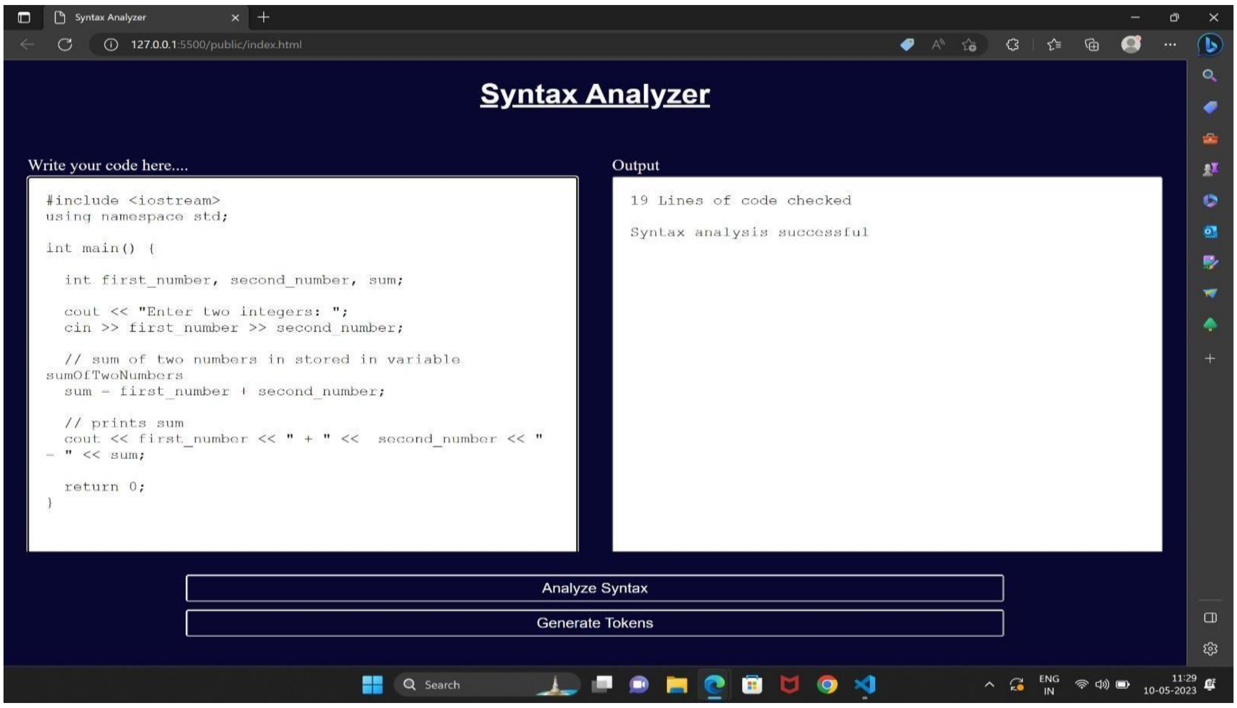
2. index.html :-

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Syntax Analyzer</title>
 <script src="/functions.js"></script>
 <link rel="stylesheet" href="style.css">
 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
          awesome/4.7.0/css/font-awesome.min.css">
 </head>
 <body>
                  <h1> Syntax Analyzer </h1>
         <div class="column">

                 <div class="row">
                            <h2>Write your code here... </h2>
                            <textarea id="code"></textarea>
                   </div>

                 <div class="row">
                              <h2>Output</h2>
                              <textarea id="output"></textarea>
                 </div>

        </div>

        <div id="Convert_button">
        <button id="convertButton" onclick="Convert_button()">
            Analyze Syntax
         </button>

          <button class="generateTokens" onclick="GenerateTokens()">
           Generate Tokens
          </button>
     </div>

 </body>
</html>
```

TESTING

Testing the analyzer -

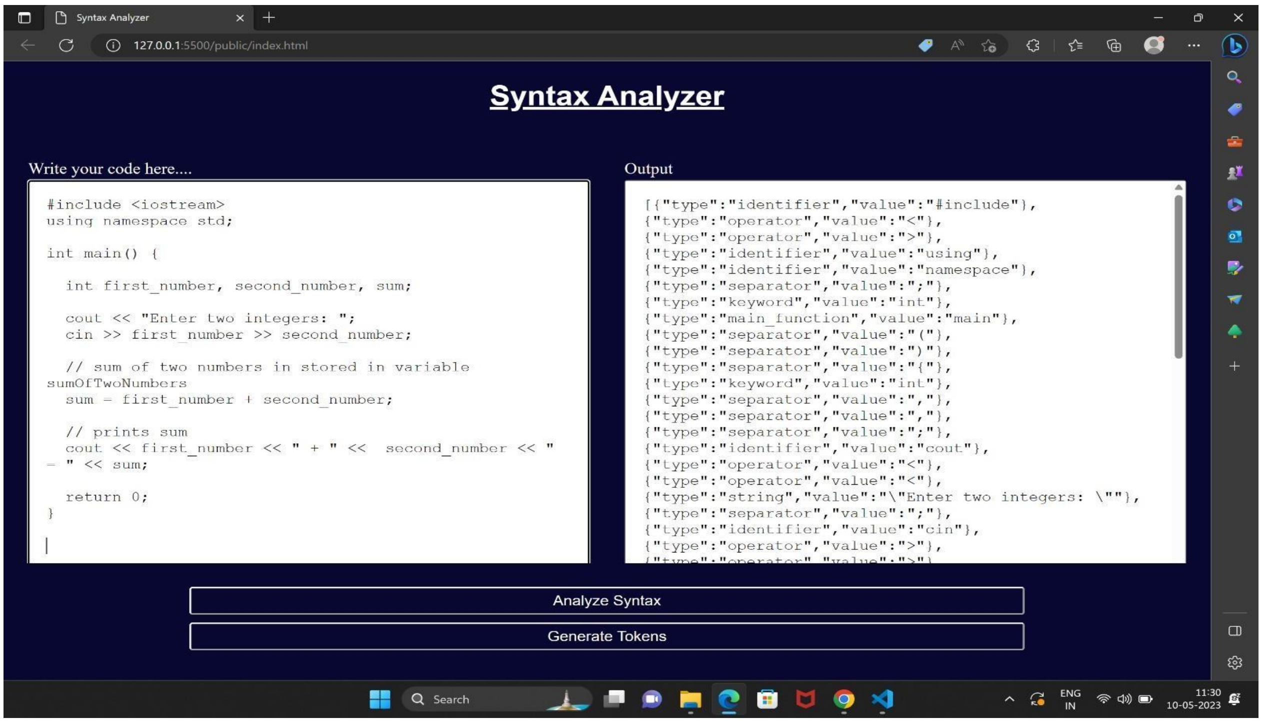
Figure 5.1



```
#include <iostream>
using namespace std;

int main() {

  int first_number, second_number, sum;

  cout << "Enter two integers: ";
  cin >> first_number >> second_number;

  // sum of two numbers in stored in variable
sumOfTwoNumbers
  sum = first_number + second_number;

  // prints sum
  cout << first_number << " + " <<  second_number << "
= " << sum;

  return 0;
}
```

```
19 Lines of code checked

Syntax analysis successful
```

Testing with some different inputs -



```
#include <iostream>
using namespace std;

int main() {

  int first_number, second_number, sum;

  cout << "Enter two integers: ";
  cin >> first_number >> second_number;

  // sum of two numbers in stored in variable
sumOfTwoNumbers
  sum = first_number + second_number;

  // prints sum
  cout << first_number << " + " <<  second_number << "
= " << sum;

  return 0;
}
```

```
[{"type":"identifier","value":"#include"},
{"type":"operator","value":"<"},
{"type":"operator","value":">"},
{"type":"identifier","value":"using"},
{"type":"identifier","value":"namespace"},
{"type":"separator","value":";"},
{"type":"keyword","value":"int"},
{"type":"main_function","value":"main"},
{"type":"separator","value":"("},
{"type":"separator","value":")"},
{"type":"separator","value":"{"},
{"type":"keyword","value":"int"},
{"type":"separator","value":","},
{"type":"separator","value":","},
{"type":"separator","value":";"},
{"type":"identifier","value":"cout"},
{"type":"operator","value":"<"},
{"type":"operator","value":"<"},
{"type":"string","value":"\"Enter two integers: \""},
{"type":"separator","value":";"},
{"type":"identifier","value":"cin"},
{"type":"operator","value":">"},
```

Figure 5.2

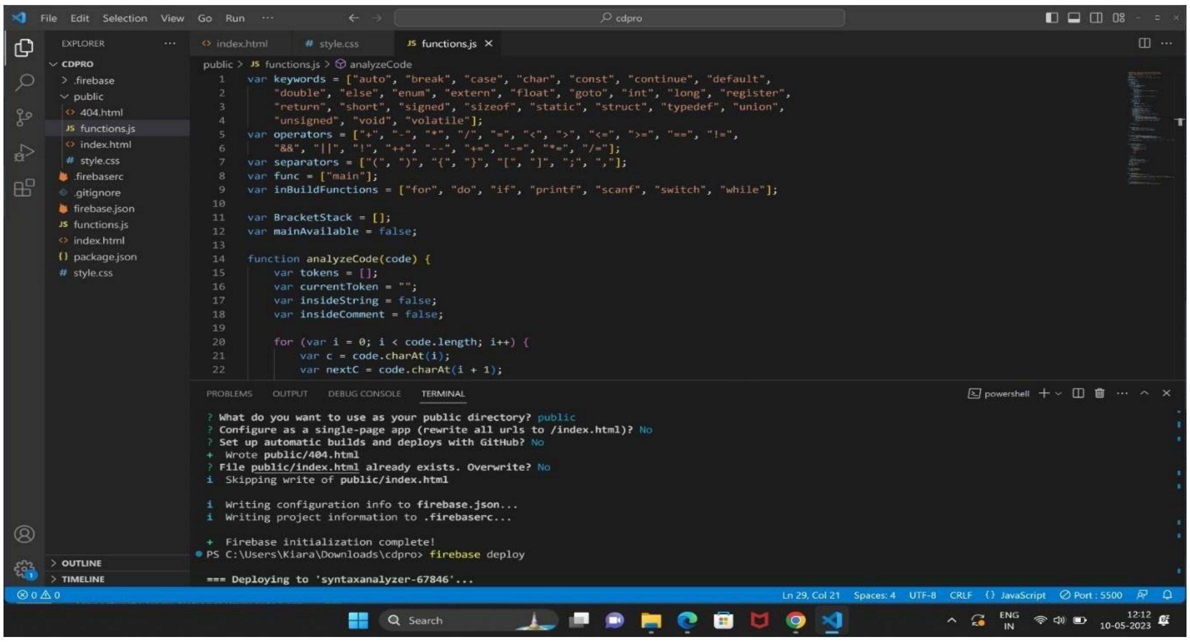
CHAPTER 6

OUTPUT AND RESULTS

5.1 <u>OUTPUT:</u>-

<u>Terminal –</u>

Figure 6.1

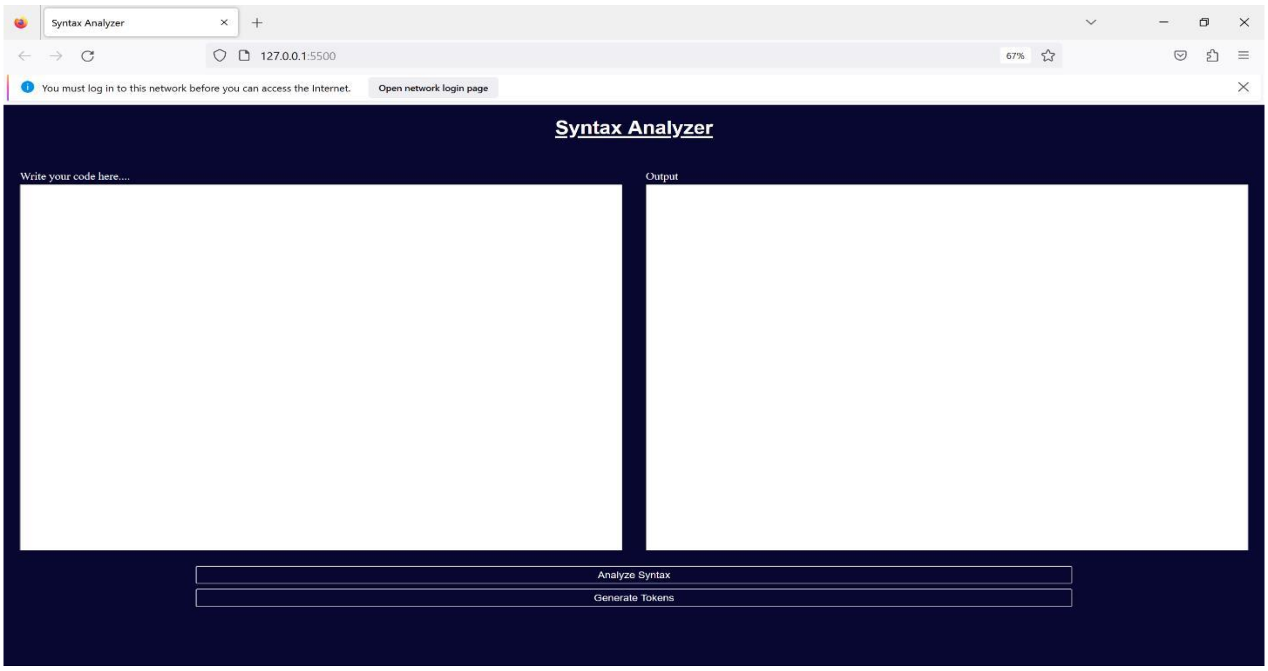


```
public > functions.js > analyzeCode
var keywords = ["auto", "break", "case", "char", "const", "continue", "default",
    "double", "else", "enum", "extern", "float", "goto", "int", "long", "register",
    "return", "short", "signed", "sizeof", "static", "struct", "typedef", "union",
    "unsigned", "void", "volatile"];
var operators = ["+", "-", "*", "/", "=", "<", ">", "<=", ">=", "==", "!=",
    "&&", "||", "!", "++", "--", "+=", "-=", "*=", "/="];
var separators = ["(", ")", "{", "}", "[", "]", ";", ","];
var func = ["main"];
var inBuildFunctions = ["for", "do", "if", "printf", "scanf", "switch", "while"];

var BracketStack = [];
var mainAvailable = false;

function analyzeCode(code) {
    var tokens = [];
    var currentToken = "";
    var insideString = false;
    var insideComment = false;

    for (var i = 0; i < code.length; i++) {
        var c = code.charAt(i);
        var nextC = code.charAt(i + 1);
```

```
? What do you want to use as your public directory? public
? Configure as a single-page app (rewrite all urls to /index.html)? No
? Set up automatic builds and deploys with GitHub? No
+  Wrote public/404.html
? File public/index.html already exists. Overwrite? No
i  Skipping write of public/index.html

i  Writing configuration info to firebase.json...
i  Writing project information to .firebaserc...

+  Firebase initialization complete!
PS C:\Users\Kiara\Downloads\cdpro> firebase deploy

=== Deploying to 'syntaxanalyzer-67846'...
```

<u>Home Page -</u>

Figure 5.2



You must log in to this network before you can access the Internet.

5.2 RESULT

The results of a semantic analyzer can vary depending on the specific implementation and use case. However, some common results of a semantic analyzer could include:

1. **Parse tree:** The syntax analyzer generates a parse tree that represents the syntactic structure of the input program. The parse tree is a hierarchical structure that shows the relationship between the tokens in the input program and their corresponding syntax rules.
2. **Syntax error:** If the syntax analyzer finds an error in the input program, it reports a syntax error and stops the parsing process. The syntax error message usually includes information about the location of the error in the input program and a description of the error.

3. **Symbol Table:** The syntax analyzer may also create a symbol table that stores information about the variables, functions, and other symbols defined in the input program. The symbol table is used by subsequent phases of the compiler or interpreter to resolve references to symbols in the input program.

4. **Intermediate code :** In some cases, the syntax analyzer may generate intermediate code that represents the input program in a form that is easier to process by subsequent phases of the compiler or interpreter. The intermediate code may be in a lower-level language, such as assembly language, or in a higher-level language, such as bytecode

5. **Optimization information:** The syntax analyzer may also provide optimization information that can be used by subsequent phases of the compiler or interpreter to optimize the performance of the input program. This information may include details about the structure of the input program, such as loop nests and data dependencies.

CHAPTER 7

7.1 CONCLUSION:

The syntax analyzer, also known as the parser, is an important component of the compiler or interpreter that checks whether the sequence of tokens in the input program corresponds to a valid sentence in the programming language's grammar. The main goal of the syntax analyzer is to build a parse tree, which represents the syntactic structure of the input program.

During the parsing process, the syntax analyzer performs various tasks such as tokenization, error detection and recovery, and building the parse tree. The tokenization process involves breaking down the input program into a sequence of tokens based on the language's lexical rules. The error detection and recovery mechanism helps the syntax analyzer to detect and handle errors in the input program, allowing the parser to continue parsing the input program even if errors are found.

Once the input program has been tokenized and errors have been handled, the syntax analyzer builds a parse tree based on the language's grammar rules. The parse tree represents the syntactic structure of the input program in a hierarchical manner, with the root node representing the start symbol of the language's grammar and the leaves representing the input tokens.

The syntax analyzer's primary objective is to ensure that the input program conforms to the language's syntax rules, and if it does, it generates the parse tree as output. If the input program does not conform to the syntax rules, the syntax analyzer reports an error and stops the parsing process.

REFERENCES

1. https://dickgrune.com/Books/PTAPG_1st_Edition/BookBody.pdf /

2. https://www.cs.princeton.edu/~appel/modern/c/

3. https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.ht m
4. https://www.geeksforgeeks.org/introduction-to-syntax-analysis/