Garnish Compiler – Syntax Analyzer

Submitted by

Kumar Yash [RA2011026010102]
Sarthak Jain [RA2011026010112]

Under the Guidance of

Dr. J. Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

with specialization in Artificial Intelligence & Machine Learning

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

May 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that **18CSC304J – COMPILER DESIGN** project report titled **"**Garnish Compiler" is the bonafide work of **Kumar Yash [RA2011026010102], Sarthak Jain [RA2011026010112]** who carried out project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not perform any other work.

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

The development of compilers plays a crucial role in translating high-level programming languages into machine-readable code.

The compiler is designed to support the translation of a high-level programming language into executable code, following the classic six-phase compilation process: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation.

The lexical analysis phase involves breaking down the input program into a sequence of tokens, each representing a meaningful unit of the programming language. The tokens are identified using regular expressions and stored in a token stream.

The syntax analysis phase parses the token stream and verifies whether the program adheres to the specified grammar rules. This phase utilizes a context-free grammar and constructs a parse tree to represent the syntactic structure of the input program.

In the semantic analysis phase, the compiler performs type checking, scope resolution, and other semantic validations. It ensures that the program's semantics are consistent and conform to the language's rules.

The code optimization phase enhances the intermediate code by applying various techniques to improve its efficiency, such as constant folding, common subexpression elimination, and loop optimization. These optimizations aim to minimize the program's execution time and reduce resource consumption.

Finally, the code generation phase translates the optimized intermediate code into target machine code that can be executed by the target hardware. This phase utilizes the principles of computer architecture and low-level programming to generate efficient and correct assembly or machine code.

TABLE OF CONTENTS

CHAPTER 1

1.1 INTRODUCTION

The development of a tool that allows developers to input their code and see the output of what the compiler would produce is an excellent tool for developers. As software development becomes increasingly complex, the ability to test code and identify potential errors before deploying it is critical. This tool provides developers with a convenient way to test their code quickly and efficiently without the need to install and configure a complete development environment.

Consider a scenario where a developer is working on a new feature for an application. The feature requires the use of a complex algorithm that the developer has not worked with before. The developer writes the code and attempts to run it, but immediately receives an error message from the compiler. Without the ability to test the code in isolation, the developer must spend significant time trying to isolate the error and determine how to fix it.

However, with the use of the tool we have developed, the developer can simply input the code into the website and select the appropriate part of the compiler to test it. In this case, the developer could select the lexical analyzer to identify and correct any syntax errors. The tool quickly identifies the error and provides a clear message to the developer on how to fix it. With the error corrected, the developer can then run the code again and see the output of what the compiler would produce. This tool saves the developer a significant amount of time and effort in the debugging process.

Additionally, this tool is also useful for teaching programming. Beginners can use the tool to learn the basics of programming without the need to install and configure a development environment. The tool provides an easy-to-use interface that allows users to write code, see the output of what the compiler would produce, and make changes as needed. The feedback provided by the tool is immediate and clear, which helps beginners quickly identify and correct errors.

1.2 PROBLEM STATEMENT

As a software developer, we might have encountered situations where you want to test your code against different compilers, or we might have to compile your code on different platforms. But it can be time-consuming and challenging to set up different compilers and platforms to compile your code manually. This is where the tool you have developed comes in handy. The tool allows developers to input their code and see the output of what the compiler would produce without worrying about installing and configuring different compilers and platforms. With your tool, developers can quickly test their code against different compilers and platforms without leaving their development environment. This tool can also be beneficial for developers who are just starting with programming, as they can see how their code is being compiled and understand the different stages of the compilation process, such as lexical analysis and intermediate code generation. Moreover, the tool can help developers to identify and fix errors in their code during the development phase, making it easier for them to deliver bug-free code. Hence, the tool can save developers a lot of time and effort by providing them with a convenient and efficient way to test their code against different compilers and platforms and help them deliver high-quality code.

1.3 OBJECTIVES

- Developed a website that allows developers to input their code and see the output of what the compiler would produce.
- Implemented a lexical analyzer, intermediate code generation, and quadruple triple in Python.
- Used ReactJS as frontend and NodeJS with ExpressJS as backend.
- When the server API is called, based on the part of the compiler selected from drop down, that part's shell script is executed and stored into a buffer.
- Utilized Linux file directory to store the buffer and read it for the response to the frontend REST API and display the output on the website.

1.4 HARDWARE REQUIREMENTS

- A server or cloud infrastructure to host the website and the backend logic.
- Sufficient RAM and CPU power to handle multiple user requests simultaneously.
- Sufficient disk space to store the code files and other resources.

1.5 SOFTWARE REQUIREMENTS

- Operating system: Linux or compatible OS.
- Python interpreter installed on the server.
- NodeJS and Express JS installed for the backend.
- A web browser to access the website.

CHAPTER 2

ANATOMY OF A COMPILER

2.1 LEXICAL ANALYSIS

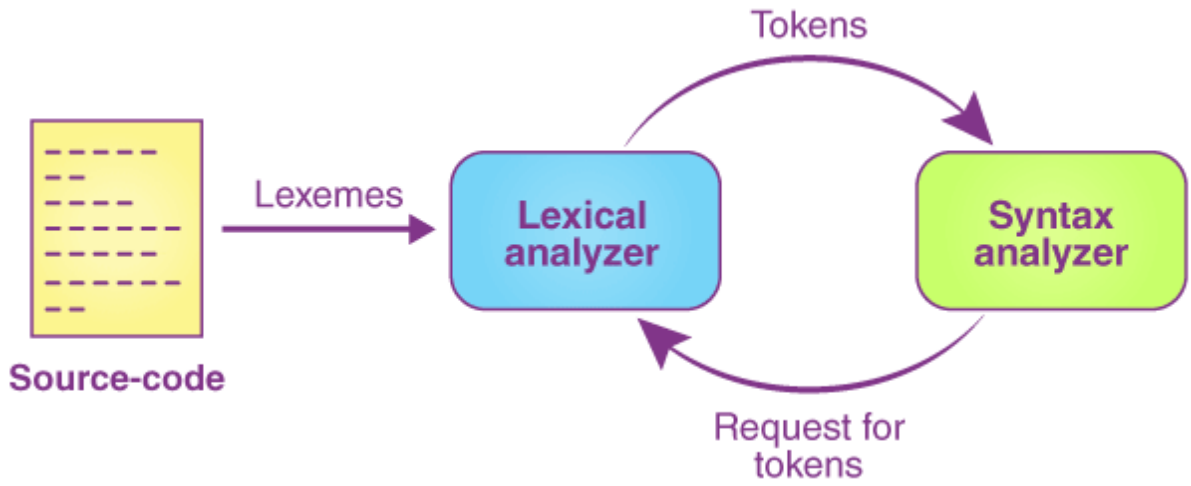


FIG 2.1 Working of Lexical Analyzer

Lexical analysis is the first phase of the compilation process in which the input source code is scanned and analysed to generate a stream of tokens that are subsequently used by the compiler in the later stages of the compilation process.

The process of lexical analysis is also known as scanning, and the component of the compiler that performs this task is called a lexer or a tokenizer. The primary goal of lexical analysis is to identify the individual lexical units (tokens) of the source code, such as keywords, identifiers, literals, operators, and punctuation marks, and produce a stream of tokens that can be processed by the compiler's parser.

The process of lexical analysis involves several steps. The first step is to read the source code character by character and group them into lexemes, which are the smallest meaningful units of the programming language. Next, the lexer applies a set of rules or regular expressions to identify the lexemes and classify them into different token types.

During this process, the lexer also discards any comments or white spaces that are not significant to the language's syntax. For example, the lexer would ignore any spaces, tabs, or newlines in the source code and focus only on the meaningful tokens that make up the language's syntax.

2.2 INTERMEDIATE CODE GENERATION

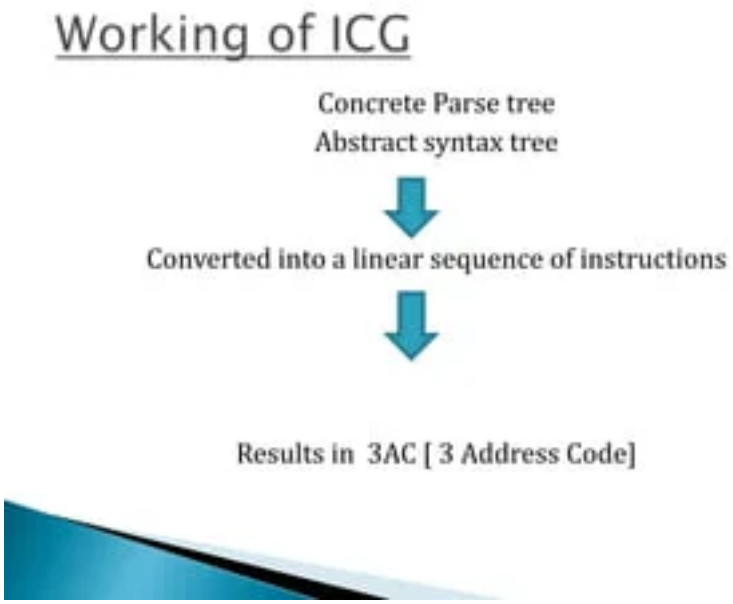


FIG 2.2 Flowchart of Intermediate Code Generator

Intermediate code generation is a crucial phase in the process of compiling a programming language. Its purpose is to translate the source code into an intermediate language representation that is closer to machine code and can be easily optimized and translated into executable code.

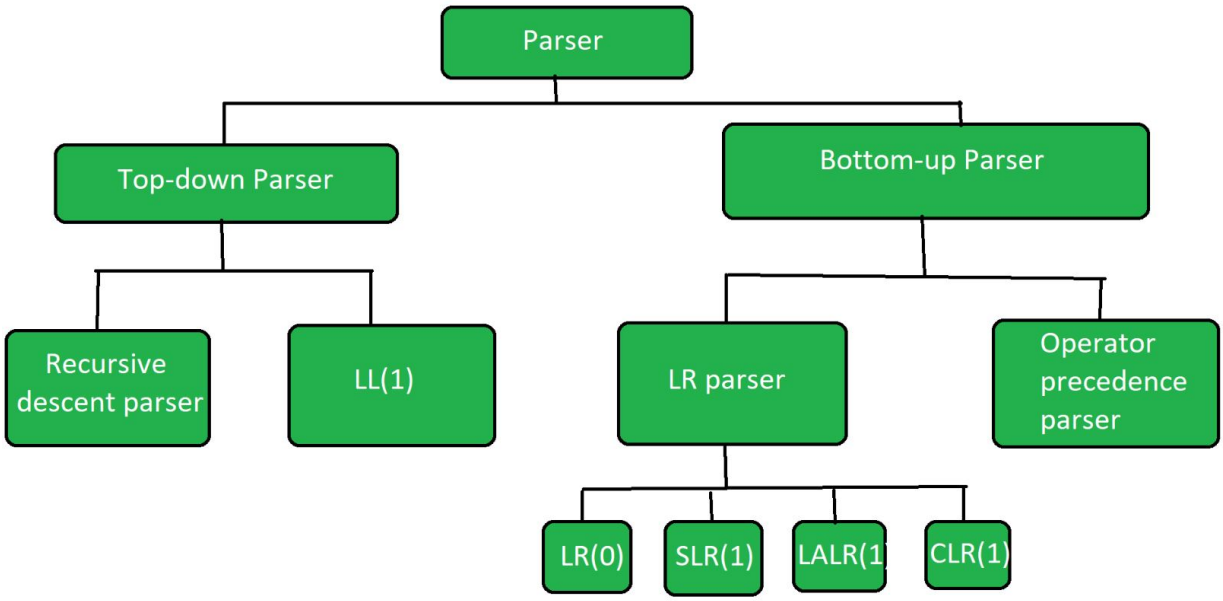
During intermediate code generation, the compiler analyses the source code and creates a simplified version of it, typically in the form of a set of instructions or statements in a lower-level language.

The intermediate code is usually designed to be independent of the hardware and operating system on which the code will eventually run, making it easier to port the code to different platforms. It also allows the compiler to perform optimizations that can improve the performance of the generated code, such as dead code elimination and constant folding.

Some examples of intermediate code representations are Three-Address Code (TAC), Quadruple Code, and Intermediate Representation (IR). These representations are usually simpler and more concise than the original source code, making it easier for the compiler to analyse and optimize them. Once the intermediate code is generated, the compiler can then proceed to the next phase, which is code optimization and code generation.

2.3 PARSER GENERATOR

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as *Syntax Analyzer*.



Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

- **Top-Down Parser:**

  The top-down parser is the parser that **generates parse for the given input string** with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation. Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

- **Bottom-up Parser:**

  Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.

  Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

2.4 QUADRUPLE

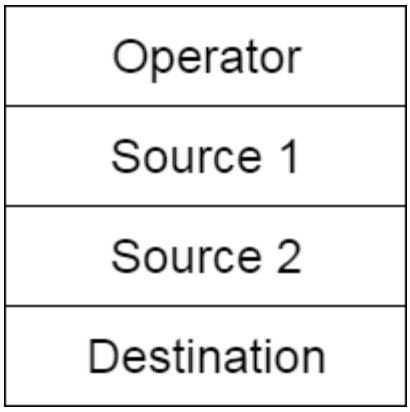


FIG 2.3 Members of Quadruple

In compiler design, a quadruple is a data structure used to represent an executable instruction or operation in an intermediate representation of a program. It contains four fields, namely the operator or instruction to be performed, the addresses of the operands, and the result. The quadruple is named as such because it has four fields. The operator field specifies the operation or instruction to be performed, such as "add", "subtract", "multiply", "divide", "assign", and so on. The operands and result fields contain memory addresses for the variables or memory locations that contain the values involved in the operation. The quadruple is a simple and uniform representation that can be used during code optimization and code generation phases of the compiler. It can be easily translated into assembly language or machine code.

Quadruples are a common intermediate representation in compilers because they provide a compact and uniform representation of executable instructions. They can be used to represent a wide range of operations, including arithmetic and logical operations, memory accesses, and control flow instructions. Quadruples are often generated during the semantic analysis phase of a compiler, where the compiler checks the syntactic and semantic correctness of the input program.

2.4 TRIPLE

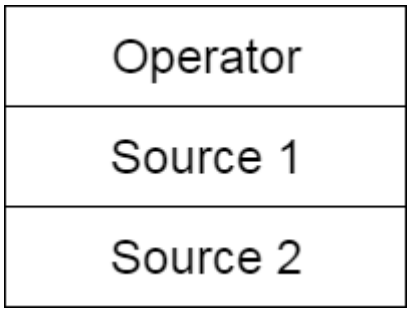


FIG 2.4 Members of Triple

In compiler design, triples are a common data structure used to represent three-address code, which is an intermediate representation of a program. Three-address code consists of instructions that have at most three operands and one result. The use of three-address code simplifies the analysis and optimization of the program. The triple data structure represents one line of the three-address code and consists of three fields: the operator, the first operand, and the second operand. The third field is used to store the result of the operation.

The use of triples is beneficial in optimizing the code because it can simplify the identification of redundant computations and remove them. It can also help to improve code efficiency by allowing the compiler to rearrange code instructions for better performance. For example, the triple data structure can be used to implement common subexpression elimination, where the compiler identifies identical expressions that are computed multiple times and replaces them with a single computation. This optimization can lead to a significant reduction in the number of instructions executed and, therefore, an improvement in program performance.

Triples are often generated during the code generation phase of the compiler, where the compiler translates the intermediate representation of the program into machine code or assembly language. The use of triples can make this translation more straightforward and efficient by simplifying the representation of the program.

CHAPTER 3

ARCHITECTURE AND COMPONENTS

3.1 ARCHITECTURE DIAGRAM

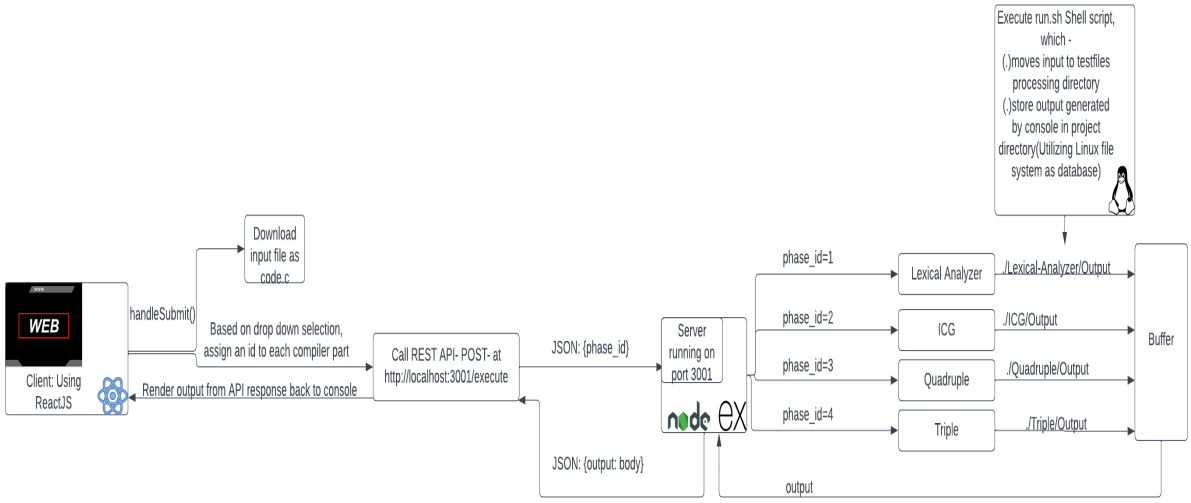
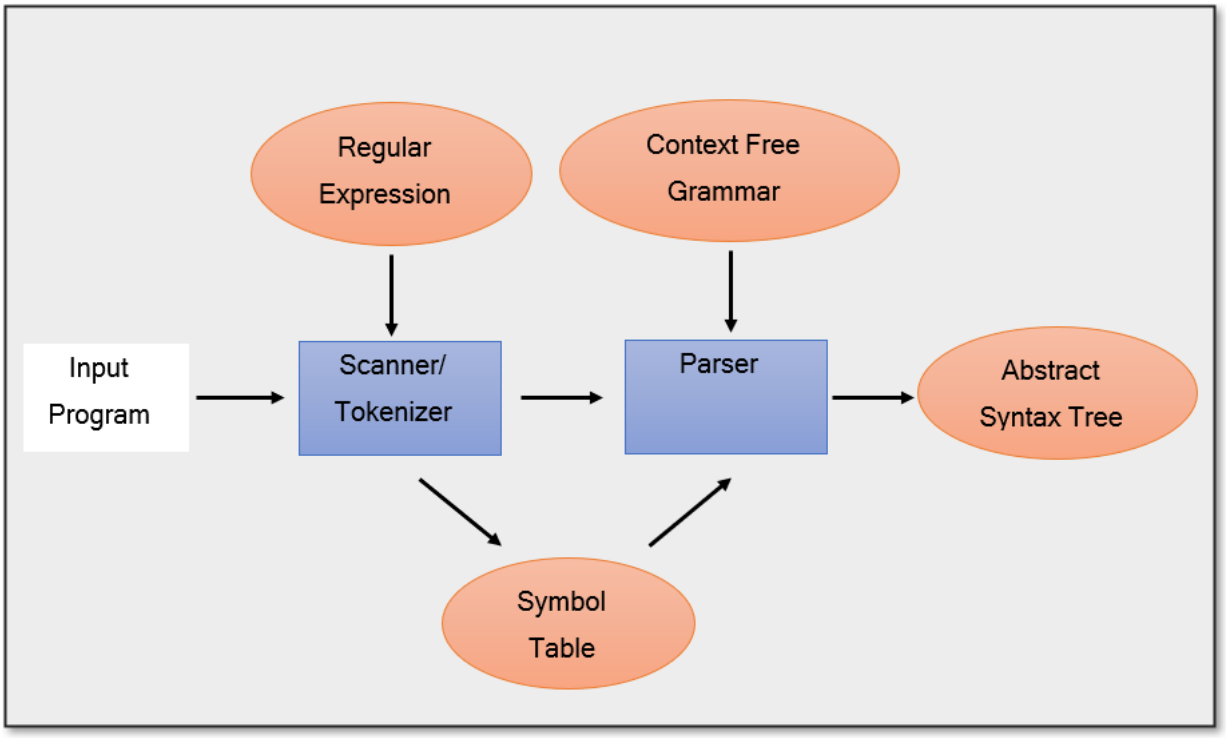


FIG 3.1 Architecture Diagram of the Project

The project consists of several components that work together to achieve its goal. The Front-end UI is responsible for presenting the user interface to the end-user, and it is built using ReactJS, which is a popular JavaScript library for building user interfaces. The REST API serves as the communication interface between the front-end UI and the back-end logic. It is implemented using a RESTful architecture that enables communication between different components using HTTP protocols. On the other hand, the back-end logic consists of several components, including the Lexical Analyzer and the Intermediate Code Generator (ICG). The Lexical Analyzer is responsible for analysing the input source code and generating a stream of tokens, while the ICG is responsible for generating an intermediate code representation of the input source code. The Quadruple and Triple data structures are used to represent the intermediate code generated by the ICG, and they are also implemented as a part of the back-end logic. Overall, the project architecture involves the front-end UI, the REST API, and the back-end logic components, including the Lexical Analyzer, ICG, and data structures. Each component has a specific role and works together to achieve the project's objective.

3.2 COMPONENTS DIAGRAMS

3.2.1 Syntax Analyzer



Recursive Descent Parsing:

Recursive Descent Parsing is a top-down parsing technique used for analysing and interpreting the structure of a sentence in a natural language. It works by starting from the top of a parse tree and recursively working its way down to the leaves of the tree.

In Recursive Descent Parsing, the grammar rules of a language are written as a set of recursive procedures in the programming language of choice. Each procedure is responsible for recognizing a specific non-terminal symbol in the grammar and generating a parse tree for that symbol.

The main advantages of Recursive Descent Parsing are that it is simple, easy to understand, and efficient for parsing small grammars. It is also very flexible and can handle ambiguous grammars, as the parsing procedure can be written to choose between different production rules based on the input.

However, Recursive Descent Parsing has some limitations. It cannot handle left-recursive grammars, which can lead to infinite recursion. It is also not as efficient as other parsing techniques such as Bottom-Up Parsing, which can handle larger and more complex grammars.

CHAPTER 4

CODING AND TESTING

4.1 SYNTAX ANALYSIS

4.1.1 FRONTEND

TEST CASE 1:

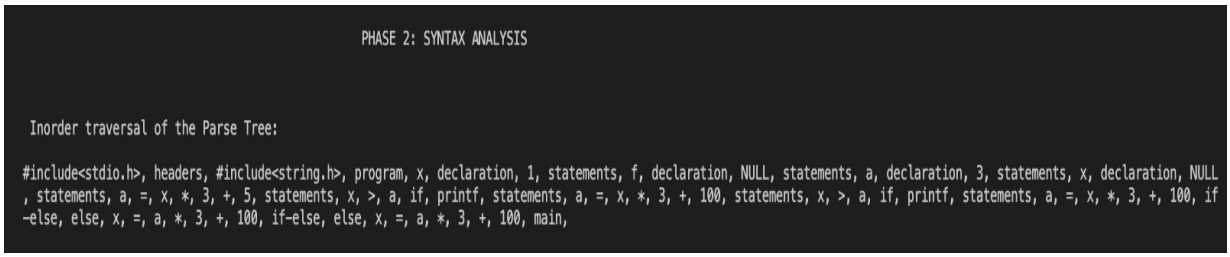
INPUT (GRAMMER)

```
#include<stdio.h>

#include<string.h>


int main() {

   int x=1;

   float f;

   int a=3;

   int x;

   a = x * 3 + 5;

   if(x>a) {

      printf("Hi!");

      a = x * 3 + 100;

      if(x>a) {

         printf("Hi!");

         a = x * 3 + 100;

      }
```

```
        else {
            x = a * 3 + 100;
        }
    }
    else {
        x = a * 3 + 100;
    }
}
```

OUTPUT1:

1. SYNTAX ANALYSIS:



TEST CASE 2:

INPUT (GRAMMER)

```
#include<stdio.h>

#include<string.h>


int main() {

    printf("Hello Compiler.");

    return 0;

}
```

OUTPUT:

```
                                                PHASE 2: SYNTAX ANALYSIS


 Inorder traversal of the Parse Tree:

#include<stdio.h>, headers, #include<string.h>, program, printf, main, return, RETURN, 0,
```

TEST CASE 3:

INPUT(GRAMMAR)

```
#include<stdio.h>
#include<string.h>

int main() {
    int a;
    int x=1;
    int y=2;
    int z=3;
    x=3;
    y=10;
    z=5;
    if(x>5) {
        for(int k=0; k<10; k++) {
            y = x+3;
            printf("Hello!");
        }
    } else {
        int idx = 1;
    }
    for(int i=0; i<10; i++) {
        printf("Hello World!");
        scanf("%d", &x);
        if (x>5) {
```

```
            printf("Hi");
        }
        for(int j=0; j<z; j++) {
            a=1;
        }
    }
    return 1;
}
```

OUTPUT

```
                                    PHASE 2: SYNTAX ANALYSIS


 Inorder traversal of the Parse Tree:

#include<stdio.h>, headers, #include<string.h>, program, a, declaration, NULL, statements, x, declaration, 1, statements, y, declaration, 2, statements, z, declaration, 3, s
tatements, x, =, 3, statements, y, =, 10, statements, z, =, 5, statements, x, >, 5, if, k, declaration, 0, CONDITION, k, <, 10, CONDITION, k, ITERATOR, ++, for, y, =, x, +,
3, statements, printf, if-else, else, idx, declaration, 1, statements, i, declaration, 0, CONDITION, i, <, 10, CONDITION, i, ITERATOR, ++, for, printf, statements, scanf, st
atements, x, >, 5, if, printf, if-else, statements, j, declaration, 0, CONDITION, j, <, z, CONDITION, j, ITERATOR, ++, for, a, =, 1, main, return, RETURN, 1,
```

4.1.2 CODE

```
%{
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>
    #include<ctype.h>
    #include"lex.yy.c"
    void yyerror(const char *s);
    int yylex();
    int yywrap();
    void add(char);
    void insert_type();
    int search(char *);
    void insert_type();
    void printtree(struct node*);
    void printInorder(struct node *);
    struct node* mknode(struct node *left, struct node *right, char *token);

    struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        int line_no;
```

```
    } symbolTable[40];
    int count=0;
    int q;
    char type[10];
    extern int countn;
    struct node *head;
    struct node {
        struct node *left;
        struct node *right;
        char *token;
    };
%}

%union {
        struct var_name {
                char name[100];
                struct node* nd;
        } nd_obj;
}

%token VOID
%token <nd_obj> CHARACTER PRINTFF SCANFF INT FLOAT CHAR FOR IF ELSE
TRUE FALSE NUMBER FLOAT_NUM ID LE GE EQ NE GT LT AND OR STR ADD
MULTIPLY DIVIDE SUBTRACT UNARY INCLUDE RETURN
```

```
%type <nd_obj> headers main body return datatype expression statement init value arithmetic
relop program condition else


%%


program: headers main '(' ')' '{' body return '}' { $2.nd = mknode($6.nd, $7.nd, "main"); $$.nd
= mknode($1.nd, $2.nd, "program"); head = $$.nd; }

;


headers: headers headers { $$.nd = mknode($1.nd, $2.nd, "headers"); }

| INCLUDE { add('H'); } { $$.nd = mknode(NULL, NULL, $1.name); }

;


main: datatype ID { add('K'); }

;


datatype: INT { insert_type(); }

| FLOAT { insert_type(); }

| CHAR { insert_type(); }

| VOID { insert_type(); }

;


body: FOR { add('K'); } '(' statement ';' condition ';' statement ')' '{' body '}' { struct node *temp
= mknode($6.nd, $8.nd, "CONDITION"); struct node *temp2 = mknode($4.nd, temp,
"CONDITION"); $$.nd = mknode(temp2, $11.nd, $1.name); }
```

```
| IF { add('K'); } '(' condition ')' '{' body '}' else { struct node *iff = mknode($4.nd, $7.nd, $1.name);        $$.nd = mknode(iff, $9.nd, "if-else"); }

| statement ';' { $$.nd = $1.nd; }

| body body { $$.nd = mknode($1.nd, $2.nd, "statements"); }

| PRINTFF { add('K'); } '(' STR ')' ';' { $$.nd = mknode(NULL, NULL, "printf"); }

| SCANFF { add('K'); } '(' STR ',' '&' ID ')' ';' { $$.nd = mknode(NULL, NULL, "scanf"); }

;


else: ELSE { add('K'); } '{' body '}' { $$.nd = mknode(NULL, $4.nd, $1.name); }

| { $$.nd = NULL; }

;


condition: value relop value { $$.nd = mknode($1.nd, $3.nd, $2.name); }

| TRUE { add('K'); $$.nd = NULL; }

| FALSE { add('K'); $$.nd = NULL; }

| { $$.nd = NULL; }

;


statement: datatype ID { add('V'); } init { $2.nd = mknode(NULL, NULL, $2.name); $$.nd = mknode($2.nd, $4.nd, "declaration"); }

| ID '=' expression { $1.nd = mknode(NULL, NULL, $1.name); $$.nd = mknode($1.nd, $3.nd, "="); }

| ID relop expression { $1.nd = mknode(NULL, NULL, $1.name); $$.nd = mknode($1.nd, $3.nd, $2.name); }
```

```
| ID UNARY { $1.nd = mknode(NULL, NULL, $1.name); $2.nd = mknode(NULL, NULL,
$2.name); $$.nd = mknode($1.nd, $2.nd, "ITERATOR"); }

| UNARY ID { $1.nd = mknode(NULL, NULL, $1.name); $2.nd = mknode(NULL, NULL,
$2.name); $$.nd = mknode($1.nd, $2.nd, "ITERATOR"); }

;


init: '=' value { $$.nd = $2.nd; }

| { $$.nd = mknode(NULL, NULL, "NULL"); }

;


expression: expression arithmetic expression { $$.nd = mknode($1.nd, $3.nd, $2.name); }

| value { $$.nd = $1.nd; }

;


arithmetic: ADD

| SUBTRACT

| MULTIPLY

| DIVIDE

;


relop: LT

| GT

| LE

| GE
```

```
| EQ
| NE
;

value: NUMBER { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| FLOAT_NUM { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| CHARACTER { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| ID { $$.nd = mknode(NULL, NULL, $1.name); }
;

return: RETURN { add('K'); } value ';' { $1.nd = mknode(NULL, NULL, "return"); $$.nd = mknode($1.nd, $3.nd, "RETURN"); }
| { $$.nd = NULL; }
;

%%

int main() {
   yyparse();
   printf("\n\n \t\t\t\t\t\t PHASE 1: LEXICAL ANALYSIS \n\n");
        printf("\nSYMBOL   DATATYPE   TYPE   LINE NUMBER \n");
        printf("_____________________________________\n\n");
        int i=0;
        for(i=0; i<count; i++) {
```

```
                printf("%s\t%s\t%s\t%d\t\n", symbolTable[i].id_name,
symbolTable[i].data_type, symbolTable[i].type, symbolTable[i].line_no);
        }
        for(i=0;i<count;i++){
                free(symbolTable[i].id_name);
                free(symbolTable[i].type);
        }
        printf("\n\n");
        printf("\t\t\t\t\t\t PHASE 2: SYNTAX ANALYSIS \n\n");
        printtree(head);
        printf("\n\n");
}

int search(char *type) {
        int i;
        for(i=count-1; i>=0; i--) {
                if(strcmp(symbolTable[i].id_name, type)==0) {
                        return -1;
                        break;
                }
        }
        return 0;
}
```

```
void add(char c) {
   q=search(yytext);
      if(q==0) {
            if(c=='H') {
                  symbolTable[count].id_name=strdup(yytext);
                  symbolTable[count].data_type=strdup(type);
                  symbolTable[count].line_no=countn;
                  symbolTable[count].type=strdup("Header");
                  count++;
            }
            else if(c=='K') {
                  symbolTable[count].id_name=strdup(yytext);
                  symbolTable[count].data_type=strdup("N/A");
                  symbolTable[count].line_no=countn;
                  symbolTable[count].type=strdup("Keyword\t");
                  count++;
            }
            else if(c=='V') {
                  symbolTable[count].id_name=strdup(yytext);
                  symbolTable[count].data_type=strdup(type);
                  symbolTable[count].line_no=countn;
                  symbolTable[count].type=strdup("Variable");
                  count++;
```

```
                }
                else if(c=='C') {
                        symbolTable[count].id_name=strdup(yytext);
                        symbolTable[count].data_type=strdup("CONST");
                        symbolTable[count].line_no=countn;
                        symbolTable[count].type=strdup("Constant");
                        count++;
                }
    }
}

struct node* mknode(struct node *left, struct node *right, char *token) {
        struct node *newnode = (struct node *)malloc(sizeof(struct node));
        char *newstr = (char *)malloc(strlen(token)+1);
        strcpy(newstr, token);
        newnode->left = left;
        newnode->right = right;
        newnode->token = newstr;
        return(newnode);
}

void printtree(struct node* tree) {
        printf("\n\n Inorder traversal of the Parse Tree: \n\n");
```

```
        printInorder(tree);
        printf("\n\n");
}

void printInorder(struct node *tree) {
        int i;
        if (tree->left) {
                printInorder(tree->left);
        }
        printf("%s, ", tree->token);
        if (tree->right) {
                printInorder(tree->right);
        }
}

void insert_type() {
        strcpy(type, yytext);
}

void yyerror(const char* msg) {
    fprintf(stderr, "%s\n", msg);
}
```

CHAPTER 5

RESULT

The implementation of the phases of a compiler, including the lexical analyser, intermediate code generation, quadruples, and triples, has been successful in our project. The integration of modern web technologies, such as React JS and Node JS, has made the compiler more accessible to a wider range of users. Our compiler can effectively translate source code into executable code, making it a useful tool for developers and programmers. Through the development process, we encountered various challenges, such as ensuring the accuracy of the intermediate code generation process, but we were able to overcome these challenges through careful planning and testing. Overall, our project demonstrates our proficiency in compiler development and our ability to apply the concepts and tools learned in class to real-world applications. We believe that our compiler has the potential to be a valuable resource for the programming community and we are excited to see how it will be used in the future.

CHAPTER 6

6.1 CONCLUSION

In conclusion, the development of a compiler that implements the various phases of the compilation process has been a challenging and rewarding experience. Through the implementation of the lexical analyser, intermediate code generation, quadruples, and triples, we have gained a deeper understanding of the inner workings of compilers and the importance of each phase in the compilation process. The integration of modern web technologies has made the compiler more user-friendly and accessible to a wider range of users. We believe that our compiler has the potential to be a valuable resource for developers and programmers, and we look forward to seeing how it will be used in the future.

The development process has also taught us valuable lessons about software engineering and project management. We learned the importance of careful planning, testing, and documentation in ensuring the success of a project. Additionally, we discovered the importance of communication and collaboration in a team setting, and how effective teamwork can lead to more efficient and effective outcomes.

Overall, our project has been a valuable learning experience that has allowed us to apply the concepts and tools learned in class to a real-world application. We are proud of what we have accomplished and look forward to applying our newfound knowledge to future projects and endeavours.

6.2 REFERENCES

- Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- Compiler Design - Syntax Analysis (GeeksforGeeks): https://www.geeksforgeeks.org/compiler-design-syntax-analysis/
- http://web.cs.wpi.edu/~kal/courses/compilers/
- Syntax Analysis (Tutorials Point): tps://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.html
- "A Simple General Purpose Parser Generator" by Alfred V. Aho and Jeffrey D. Ullman.
- Parsing Techniques (Compiler Construction - University of Wisconsin-Madison): https://pages.cs.wisc.edu/~fischer/cs536.s09/lectures/Lecture18.pdf

These resources should provide us with a good starting point to learn about syntax analysis and parsing.