C Compiler using Lex and Yacc

Automata and Compiler Design (IT250) Mini-Project Report

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

by

Divyam Gupta (Roll No. 221IT025)

Kevin Saji Jacob (Roll No. 221IT038)

Sricharan Sridhar (Roll No. 221IT066)

Department of Information Technology

National Institute of Technology Karnataka

Surathkal - 575025, India

March, 2024

D E C L A R A T I O N

I hereby declare that the IT250 report entitled “C Compiler using Lex and Yacc”which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a bonafide report of the work carried out by us. The material contained in this report has not been submitted to any University or Institution for the award of any degree.

Divyam Gupta	Kevin Saji Jacob	Sricharan Sridhar
221IT025	221IT038	221IT066

CERTIFICATE

This is to certify that the work presented in this report titled "C Compiler using Lex and Yacc" is an authentic record of work done by Divyam Gupta, Kevin Saji Jacob and Sricharan Sridhar under my supervision. To the best of my knowledge, the contents of this report have not been submitted, in part or full, for any other degree or diploma at this or any other institution.

Anupama H C
National Institute of Technology Karnataka, Surathkal
March, 2024

Table of Content

List of Figures

CHAPTER 1 INTRODUCTION

The field of computer science has seen significant advancements in the development of compilers, which are crucial for translating high-level programming languages into machine code. This report focuses on the creation of a compiler using Lex and Yacc, two widely used tools in compiler construction.

1.1 OVERVIEW OF LEX AND YACC

1.1.1 Lex

Lex is a computer program generator for lexical analyzers. It was originally developed by Mike Lesk and Eric Schmidt at Bell Labs in the 1970s. Lex takes a set of regular expressions and corresponding actions as input and generates a C program to recognize and process text based on those expressions.

The key components of a Lex program include regular expression patterns and the corresponding actions to take when a pattern is matched. Lex operates by scanning input text, matching patterns, and executing the associated actions. It is commonly used in combination with the yacc parser generator to build complete compilers and interpreters. Lex has been influential in the development of programming language tools and is often used in software development for tasks such as tokenizing input, syntax highlighting, and parsing. While Lex itself generates C code, there are also versions and adaptations for other programming languages.

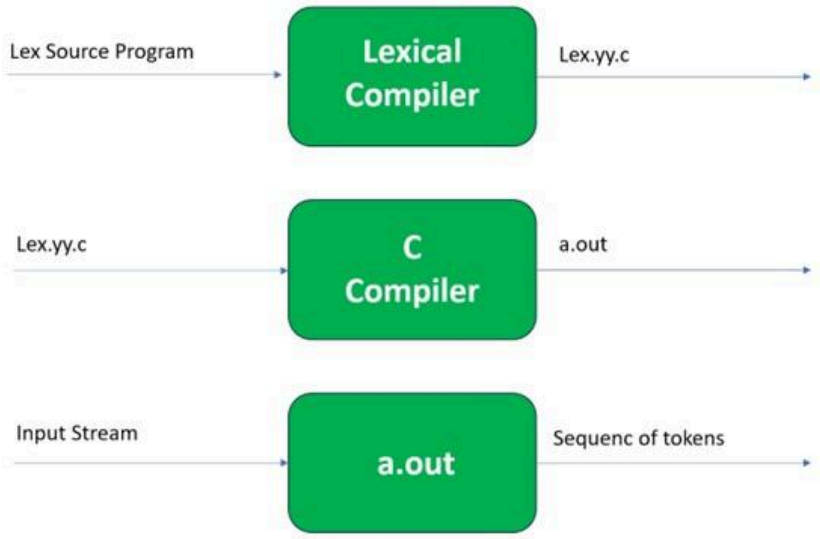

Figure 1.1
Working of Lex

1.1.2 Yacc

Yacc, short for "Yet Another Compiler Compiler," is a tool used in the generation of parsers and compilers for programming languages. It was developed by Stephen C. Johnson at AT&T Bell Laboratories in the 1970s.

Yacc takes a formal grammar specification as input, typically written in Backus-Naur Form (BNF), and generates a parser in C code (or other target languages in some implementations) that can recognize and process input according to the specified grammar rules. The key components of a Yacc program include grammar rules, which define the syntax of the language being parsed, and actions associated with each rule, which specify the processing to be done when that rule is matched. Yacc-generated parsers typically use a technique called "bottom-up parsing" to analyze input text and build a parse tree based on the grammar rules.

Yacc is commonly used in combination with Lex, to create complete compilers and interpreters. Lex handles tokenizing input text into a stream of tokens, while Yacc processes these tokens according to the grammar rules to determine the structure and meaning of the input. Yacc has been widely used in the development of programming language compilers, interpreters, and other software tools where parsing and syntax analysis are required.

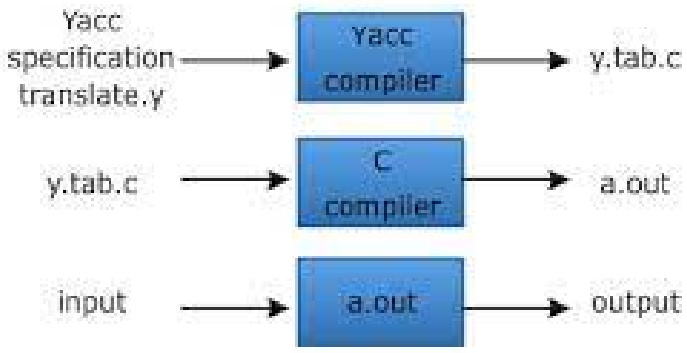

Figure 1.2
Working of Yacc

1.2 WHAT IS A COMPILER

A compiler is a software tool that translates high-level programming language code written by a programmer into machine code that can be executed directly by a computer's processor. It essentially acts as an intermediary between human-readable source code and the binary code that computers understand. The process typically involves several stages, including lexical analysis, parsing, semantic analysis, optimization, and code generation. The result is an executable program that can be run on a computer. Compilers play a crucial role in software development by enabling programmers to write code in high-level languages and abstracting away the complexities of the underlying hardware architecture.

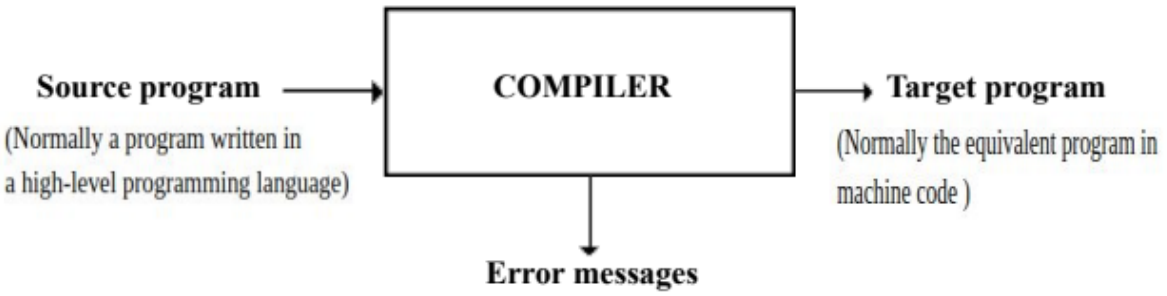

Figure 1.3
Working of compiler

CHAPTER 2 OBJECTIVE

The primary objective of creating a C compiler using Lex and Yacc is to develop a robust and efficient tool for translating human-readable C code into executable machine instructions. By leveraging the capabilities of Lex for lexical analysis and Yacc for syntax parsing, the aim is to automate the process of language interpretation and transformation. This endeavor seeks to facilitate software development by providing developers with a reliable means to compile C programs, thereby bridging the gap between high-level programming languages and machine code execution. Through the implementation of lexical and syntactic analysis, the compiler ensures adherence to the syntax and semantics of the C language, enabling error-free translation of source code.

We will be producing a machine language code for a particular input program which consists of the following

- Conditional statements
- Print statements
- Simple arithmetic operations
- Assignment operations

CHAPTER 3 METHODOLOGY

The methodology for creating a C compiler using Lex and Yacc involves a systematic approach encompassing several key steps. Firstly, the language specification is meticulously defined, outlining the syntax rules, data types, control structures, and other language features of C. This serves as the foundation for constructing the lexical and syntactic rules necessary for accurate interpretation and translation of C source code. With the language specification in hand, the next step involves implementing the lexical analyzer using Lex. This entails defining regular expressions to recognize tokens such as identifiers, keywords, constants, and operators. Lex generates a lexical analyzer based on these specifications, which tokenizes the input source code, breaking it down into a sequence of tokens for further processing.

Following the lexical analysis phase, the syntax parser is developed using Yacc to enforce the grammatical rules of the C language. A context-free grammar (CFG) is crafted, specifying the syntactic structure of valid C programs. Yacc processes this grammar specification and generates a parser capable of constructing a parse tree from the tokenized input. The parse tree represents the hierarchical structure of the program, enabling subsequent phases of compilation such as semantic analysis, optimization, and code generation. Sample C programs are compiled and executed, with thorough debugging and error handling mechanisms in place to identify and rectify any issues encountered during the compilation process.

CHAPTER 4 FLOW CHART

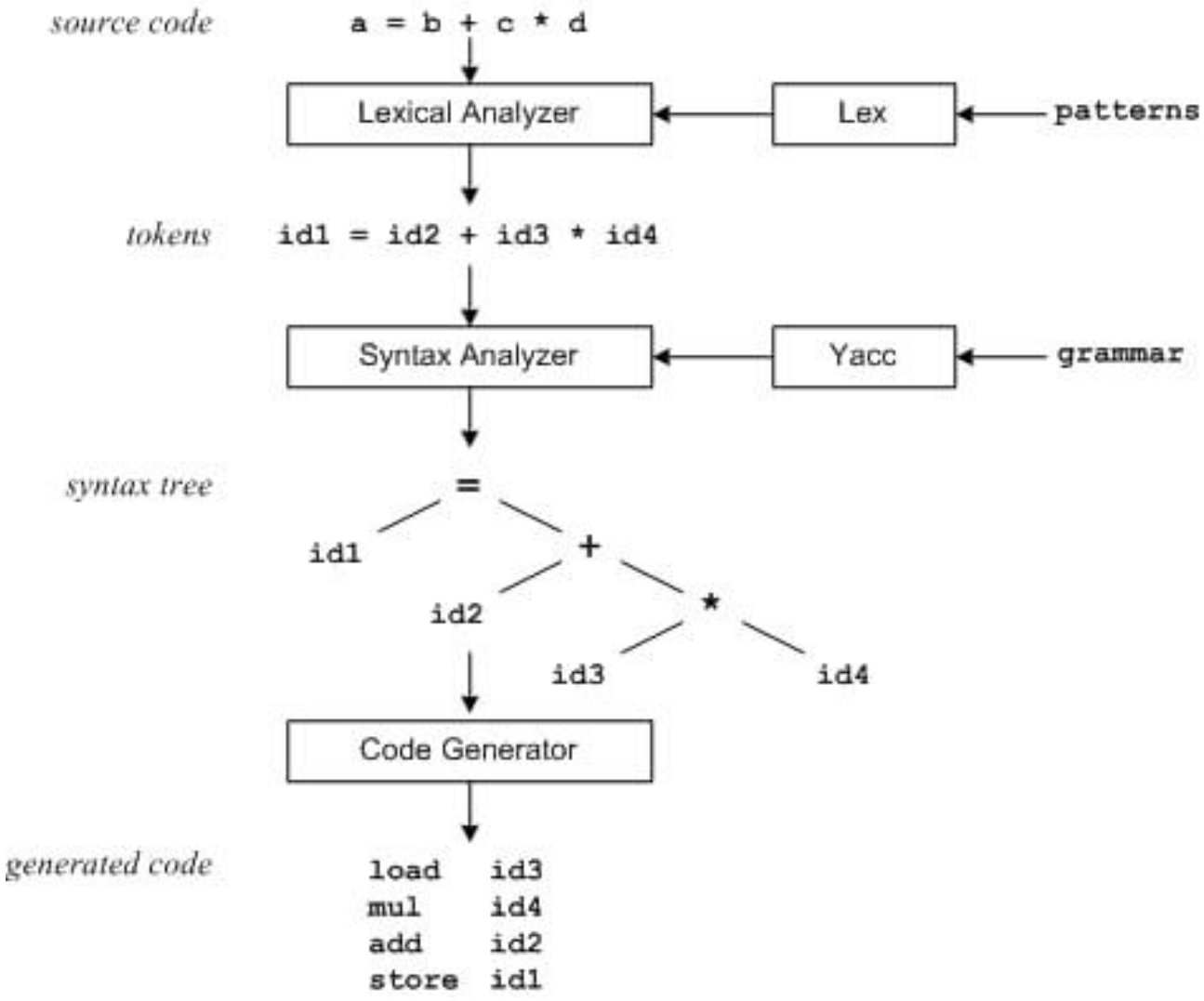

Figure 4.1

Flowchart

CHAPTER 5 IMPLEMENTATION

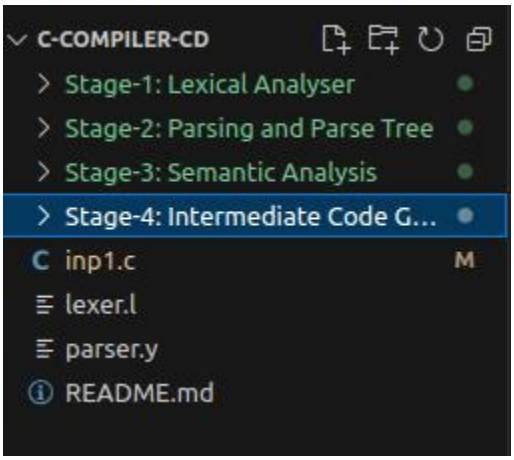

Figure 5.1
Files used to implement the program

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

int main() {
        int x = 8;
        int y = 10;
        y = x*2 + 5;
        float c;

        if (y > x){
        printf("\nWelcome to Program ! \nCheck 1 Completed .. \n");
                y = x + 100;
        if (y > x){
                        printf("\nCheck 1 and 2 Completed ..\n");
                        y = x;
        }

                else{
              x = y;
                        printf("\nCheck 2 Failed ..\n");
        }
```

```
   }

        else{
        x = y + 10;
                printf("\nCheck 1 Failed .. \n");
        }

        for(int i=0; i<10; i++) {
      printf("Hello World!");
      scanf("%d", &x);
        }

        printf("\n\nThank you. Exiting Program .. \n\n");
        return 0;
}
```

Snippet 5.1
The test program used

```
void push(char c) {
  q=search(yytext); //Checking whether the symbol is already pushed to the symbol table or
not.
  if(!q) {          //If not, then check for type
                    if(c == 'H') {                                          //Check if Header
                              symbol_table[count].id_name=strdup(yytext);
                              symbol_table[count].data_type=strdup(type);
                              symbol_table[count].line_no=countn;
                              symbol_table[count++].type=strdup("Header");
                    }
                    else if(c == 'K') {                                     //Check if Keyword
                              symbol_table[count].id_name=strdup(yytext);
                              symbol_table[count].data_type=strdup("N/A");
                              symbol_table[count].line_no=countn;
                              symbol_table[count++].type=strdup("Keyword\t");
                    }
                    else if(c == 'V') {                                     //Check if Variable
                              symbol_table[count].id_name=strdup(yytext);
                              symbol_table[count].data_type=strdup(type);
                              symbol_table[count].line_no=countn;
                              symbol_table[count++].type=strdup("Variable");
                    }
```

```
                else if(c == 'C') {                                     //Check if Constant
                        symbol_table[count].id_name=strdup(yytext);
                        symbol_table[count].data_type=strdup("CONST");
                        symbol_table[count].line_no=countn;
                        symbol_table[count++].type=strdup("Constant");
                }
                else if(c == 'F') {                                     //Check if function
                        symbol_table[count].id_name=strdup(yytext);
                        symbol_table[count].data_type=strdup(type);
                        symbol_table[count].line_no=countn;
                        symbol_table[count++].type=strdup("Function");
                }
        }
}
```

Snippet 5.2
Function for pushing a function into the Symbol Table

```
%%
"main"                  { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"<="                    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
">="                    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"=="                    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"!="                    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
">"                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"<"                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"="                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"&&"                    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"||"                    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"+"                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"-"                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"/"                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"*"                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
\/\/.*                  { ; }
\/\(.\n).\*\/           { ; }
[ \t]*                  { ; }
[\n]                    { countn++; }
["].*["]                { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
['].[']                 { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }

"printf"                { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"scanf"                 { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
```

```
"int"                   { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"float"                 { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }

"char"                  { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"void"                  { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"return"                { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"for"                   { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"if"                    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"else"                  { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
^"#include"[ ]*<.+\.h>      { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"true"                  { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
"false"                 { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
[-]?{digit}+            { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
[-]?{digit}+\.{digit}{1,6}   { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
{alpha}({alpha}|{digit})*    { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
{unary}                 { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
[(){};]                     { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
.                       { fprintf(stdout,"%d\t\t%s\n",countn,yytext); }
%%
```

Snippet 5.3
Function for Token Recognition in Lexical Analysis

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

void printtree(struct node* tree) {
        printf("\n\nParse Tree (Inorder Traversal) :- \n\n");
        printInorder(tree);
        printf("\n\n");
}
```

Snippet 5.4
Function for adding a node and printing the parse tree

```
int check_types(char *type1, char *type2){
	// declaration with no init
	if(!strcmp(type2, "null"))
		return -1;
	// both datatypes are same
	if(!strcmp(type1, type2))
		return 0;
	// both datatypes are different
	if(!strcmp(type1, "int") && !strcmp(type2, "float"))
		return 1;
	if(!strcmp(type1, "float") && !strcmp(type2, "int"))
		return 2;
	if(!strcmp(type1, "int") && !strcmp(type2, "char"))
		return 3;
	if(!strcmp(type1, "char") && !strcmp(type2, "int"))
		return 4;
	if(!strcmp(type1, "float") && !strcmp(type2, "char"))
		return 5;
	if(!strcmp(type1, "char") && !strcmp(type2, "float"))
		return 6;
}
```

Snippet 5.5

Function for for checking similarity of data types in the Semantic Analysis code

CHAPTER 6 RESULTS

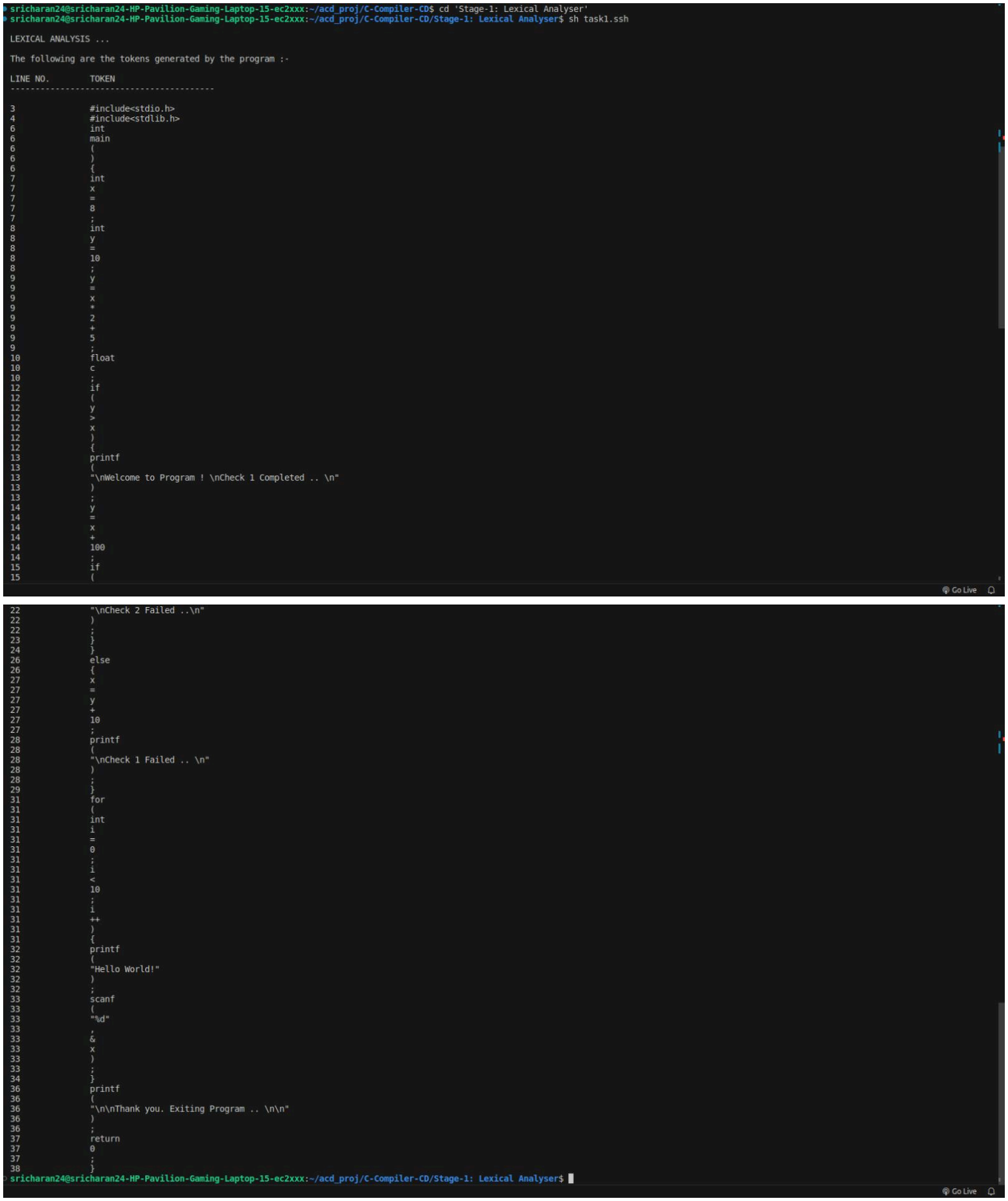

Figure 6.1 and 6.2
Lexical Analysis

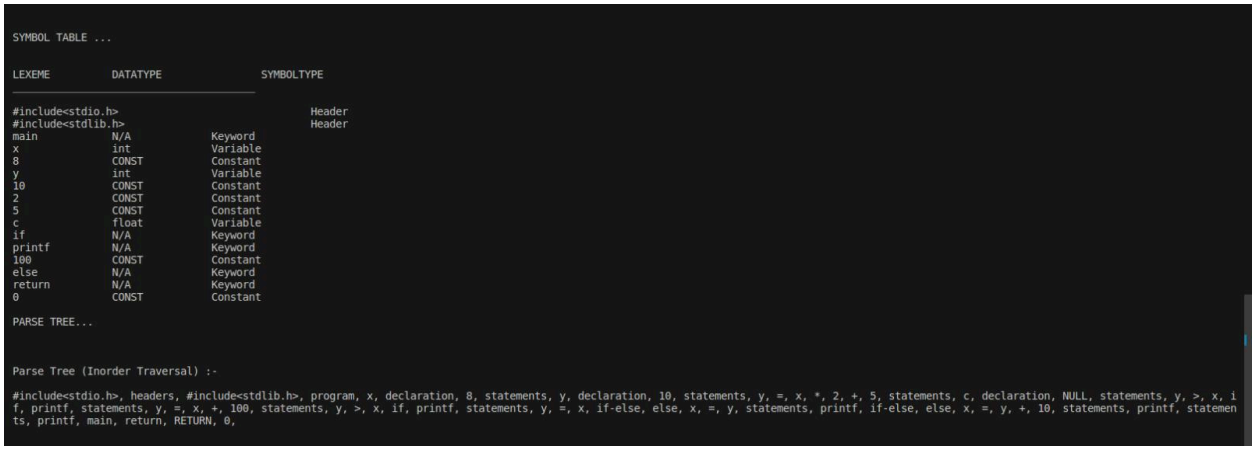

Figure 6.3

Symbol table generation with parse tree

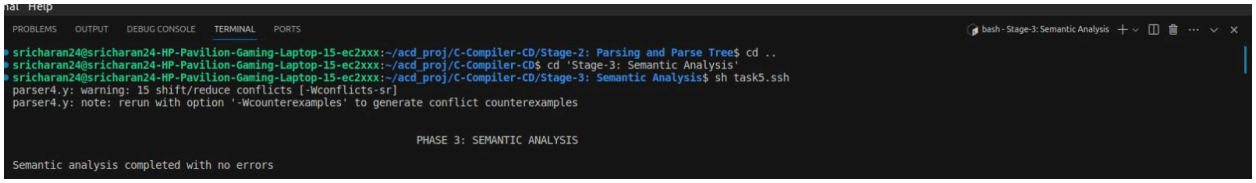

Figure 6.4

Semantic Analysis

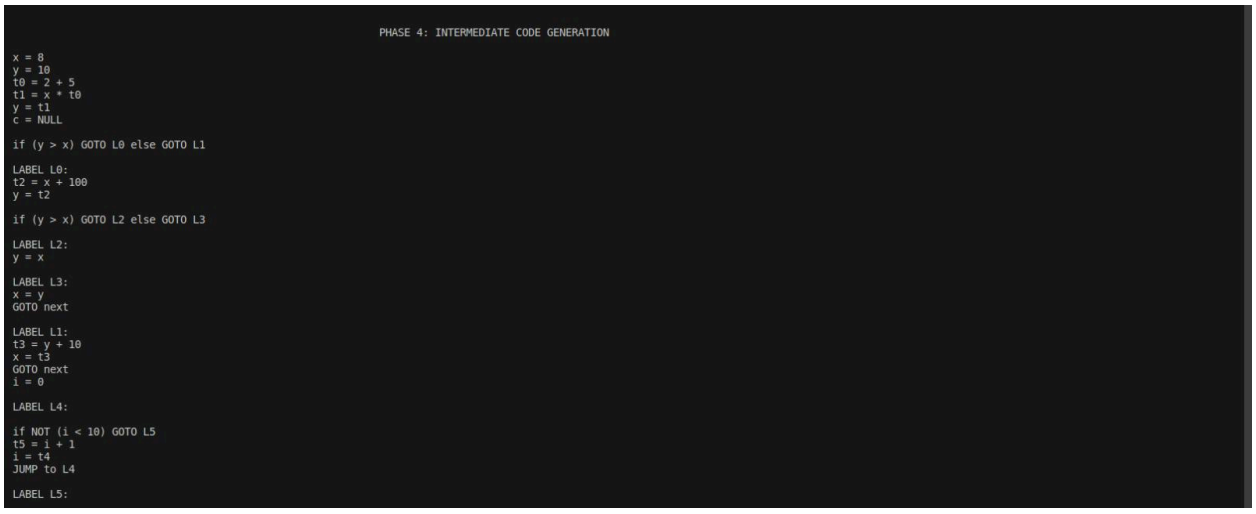

Figure 6.5

Intermediate Code Generation

The C Compiler designed works for the following constructs

- For loops
- Nested loops
- Conditional Statements
- Declarations
- Comparisons

6.1 Future work

Enhance the compiler's functionality to encompass the parsing and compilation of functions declared outside the main function, and tailor it to support structs in the C programming language.

6.2 References

1. Trinidad, Alicia & Morales, Rafael & Abril, Miguel & Iciarra, Luis & Cardenas Vazquez, M. & Rabaza, Ovidio & Ballesta, Alejandro & Sánchez Carrasco, Miguel & Becerril, S. & González, Pedro. (2010). A simple way to build an ANSI-C like compiler from scratch and embed it on the instrument's software. Proceedings of SPIE - The International Society for Optical Engineering. 10.1117/12.856385.
2. Guimarães, José. (2007). Learning compiler construction by examples. SIGCSE Bulletin. 39. 70-74. 10.1145/1345375.1345418.