SYNTAX ANALYZER FOR C LANGUAGE

A Project Report

Submitted by

ISHAAN KANGRIWALA,
PAARTH KAPASI,
DHRUV PATEL,
NIHAL SHETTY.

Under the Guidance of

PROF. AMEYAA BIWALKAR

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY CSBS

COMPUTER ENGINEERING

At

MUKESH PATEL SCHOOL OF TECHNOLOGY MANAGEMENT AND ENGINEERING

OCTOBER, 2021.

DECLARATION

We, **Ishaan Kangriwala, Paarth Kapasi, Dhruv Patel and Nihal Shetty,** Roll Nos. **E019, E020**, **E035 and E050,** B. Tech CSBS (Computer Engineering), V semester understand that plagiarism is defined as anyone or combination of the following:

1. Un-credited verbatim copying of individual sentences, paragraphs, or illustration (such as graphs, diagrams, etc.) from any source, published or unpublished, including the internet.
2. Un-credited improper paraphrasing of pages paragraphs (changing a few words phrases, or rearranging the original sentence order)
3. Credited verbatim copying of a major portion of a paper (or thesis chapter) without clear delineation of who did wrote what. (Source: IEEE, The institute, Dec. 2004)
4. I have made sure that all the ideas, expressions, graphs, diagrams, etc., that are not a result of my work, are properly credited. Long phrases or sentences that had to be used verbatim from published literature have been clearly identified using quotation marks.
5. I affirm that no portion of my work can be considered as plagiarism, and I take full responsibility if such a complaint occurs. I understand fully well that the guide of the seminar/ project report may not be in a position to check for the possibility of such incidences of plagiarism in this body of work.

Signature of the Student:

Name: Ishaan Kangriwala	Paarth Kapasi	Dhruv Patel	Nihal Shetty
Roll No. E019	E020	E035	E050

Place: Mumbai

Date: 21/10/21

CERTIFICATE

This is to certify that the project entitled **“Syntax Analyzer for C Language”** is the bonafide work carried out by **Ishaan Kangriwala, Paarth Kapasi, Dhruv Patel and Nihal Shetty,** of B.Tech CSBS (Computer Engineering), MPSTME (NMIMS), Mumbai, during the V semester of the academic year 2021-2022, in partial fulfillment of the requirements for the award of the Degree of Bachelors of Engineering as per the norms prescribed by NMIMS. The project work has been assessed and found to be satisfactory.

Prof. Ameyaa Biwalkar

Internal Mentor

Examiner 1

Examiner 2

Director

Table of contents

List of Figures

****

Abstract

A compiler is a tool that translates the source code written in a programming language (the source language) into another computer language (the target language). When executing, the compiler first parses all the language statements, checking for the syntax along with it as well, one after the other and then, in one or more passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Four Phases of the frontend compiler are Lexical phase, Syntax phase, Semantic phase and Intermediate code generation. Once the parse tree is generated the Semantic Analyser will check actual meaning of the statement parsed in parse tree. In this project we have implemented the Syntax Analyzer for a C language Compiler and have done syntax specification for looping construct, data type, operators, arrays, functions, keywords, comments, constant errors and if-else statements. The Syntax analyzer and the preceding lexical analyzer has been implemented using the Flex and Bison tools and compiled using GCC (the C compiler).

Keywords: Lexical, Syntax, flex, Bison, GCC, Compiler, Errors, Analyzer, Operators.

Chapter 1

Introduction

1.1 Project Overview

A compiler is a tool that converts a high-level language into a low-level machine understandable language. The compiler is divided into two major phases: Front-end and Back-end. The front-end can also be called the Analysis phase of the compiler.
Analysis Phase – An intermediate representation is created from the given source code:

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator

Lexical analyser divides the program into "tokens", Syntax analyser recognizes "sentences" in the program using syntax of language and Semantic analyser checks static semantics of each construct. Intermediate Code Generator generates "abstract" code.
Syntax analysis is the process of checking the syntactical correctness of the code. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a grammar. A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e., what is a valid sentence in the language). There can be more than one grammar for a given language. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree. The syntax analyzer for the C language can be created by writing two scripts, one that acts as a lexical analyzer (lexer) and outputs a stream of tokens, and the other one that acts as a parser. The lexer is known as the lex program. Lex reads an input stream specifying the lexical

analyzer and outputs source code implementing the lexer in the C programming language. In this project we aim to implement a Syntax analyser for the C language, and we do so by using the Flex and Bison tools and the GCC compiler. Syntax analysis logically follows the Lexical phase and thus we do the lexical analysis before implementing the Syntax analyser.

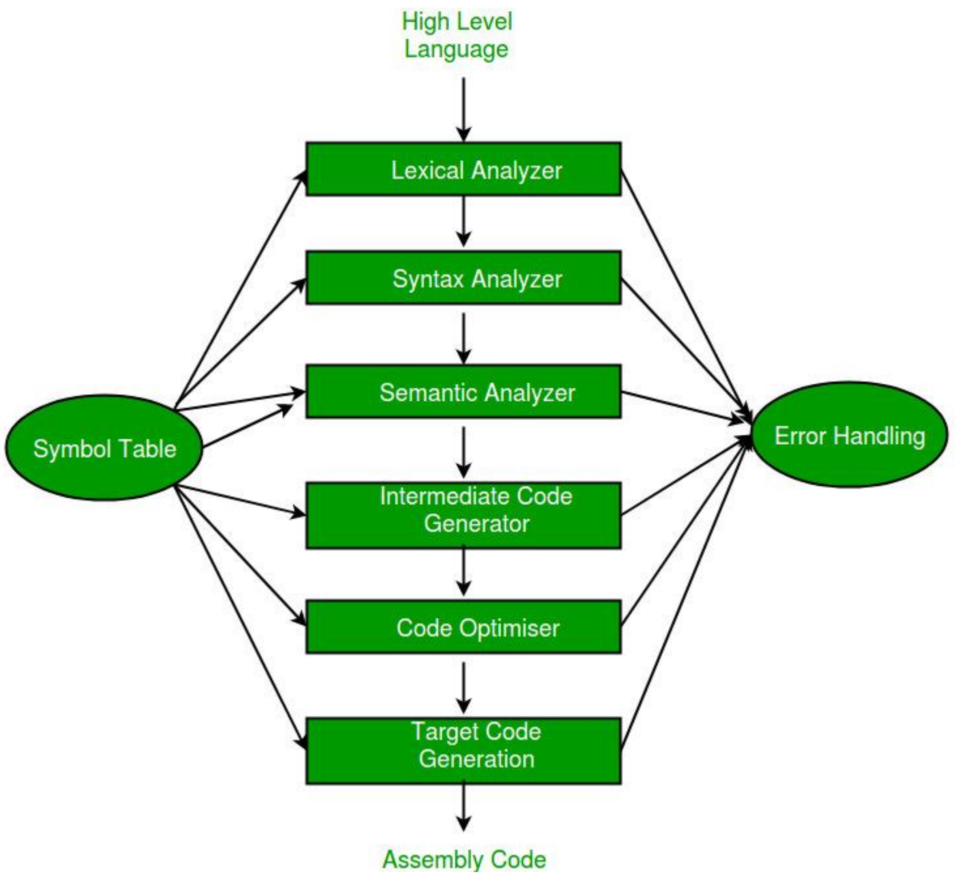


Figure 1.1 Phases of a compiler

1.1.1 Formatting Guidelines

The document is prepared using Microsoft Word 2021 and has used the font type 'Times New Roman'. The fixed font size that has been used to type this document is 12pt. with 1.5 line spacing. It has used the bold property to set the headings of the document. All pages except the cover page are numbered, the numbers appear on the lower right- hand corner of the page. Every image and data table are numbered and referred to the in the main text.

1.2 Hardware Specifications

The hardware interface for the project should be a Personal Computer/Laptop with a Windows 10 Operating System. There is no specific hardware component requirements since our project is not a memory intensive or high storage consuming in nature. The purpose of this windows computer is to provide information of the data entered by the user to the program without any bottlenecking.

The minimum system requirements needed for the software to run are as follows:

● Processor: 1 gigahertz (GHz) or faster processor

● Hard Disk Space (HDD/SSD): 500 MB

● RAM: 1 GB and above

● Display: Minimum 800 x 600 Resolution

● Graphics: 512MB with DirectX 9.0 and later versions

Display Unit: A Display unit is necessary for the input and output operations. It is used to help the user with the interface, display information about the software and to accept the input and display the output that has been obtained based on it. It can also display video/audio media as a form of entertainment for the user.

1.3 Software Used

1.4.1 FLEX

Flex, which stands for Fast Lexical Analyzer Generator, is a free and open-source computer program that generates lexical analyzers, also known as scanners or lexers (programs which recognize lexical patterns in text) which is why it is also sometimes referred to as a scanner generator. It is frequently used as the lex implementation together with GNU Bison which is a version of YACC. Flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, ‘lex.yy.c’, which defines a routine ‘yylex()’. This file is compiled and linked with the ‘-lfl’ library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code. A Flex lexical analyzer usually has time complexity O(n) in the length of the input. That is, it performs a constant number of operations for each input symbol. By default, the scanner generated by Flex is not re-entrant i.e., multiple invocations can safely run concurrently on multiple processors, or on a single processor system, where a re-entrant procedure can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. This can cause serious problems for programs that use the generated scanner from different threads. Flex can only generate code for C and C++. To use the scanner code generated by flex from other languages a language binding tool.

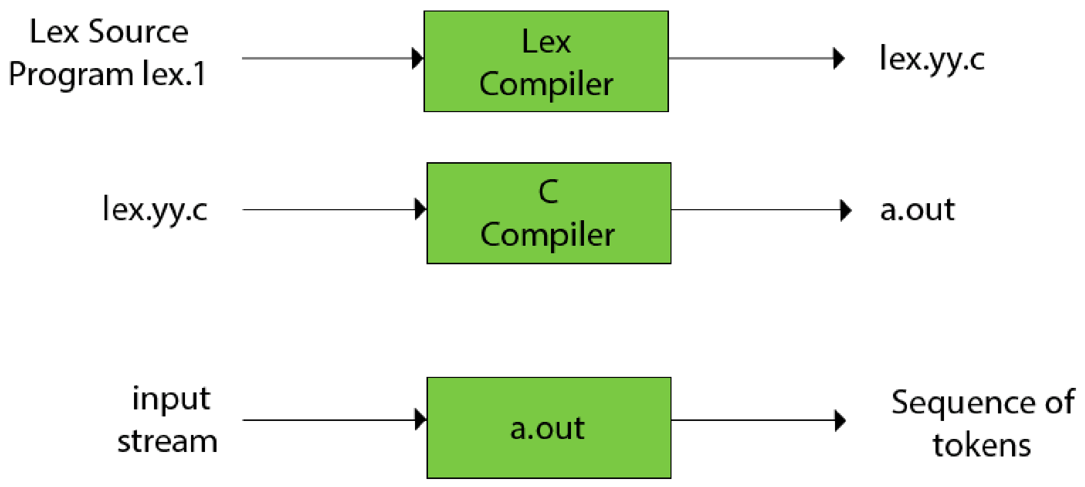


Figure 1.2 Compiling a Lex file

1.4.2 BISON YACC

Bison is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. The generated parsers are portable: they do not require any specific compilers. Bison by default generates LALR (1) parsers but it can also generate Canonical LR (CALR), IELR (1) and GLR parsers but it is an experimental feature, not a complete implementation. Bison can be used to develop a wide range of language parsers from the ones used in simple desk calculators to complex programming languages. Bison is upward compatible with YACC, so all properly written YACC grammars should work with Bison with no change. You need to be fluent in C or C++ programming in order to use Bison. Java is also supported as an experimental feature. Bison automatically does the parsing for us, remembering what rules have been matched, so the action code maintains the values associated with each symbol. Bison parsers also perform side effects such as creating data structures for later use or printing out results.

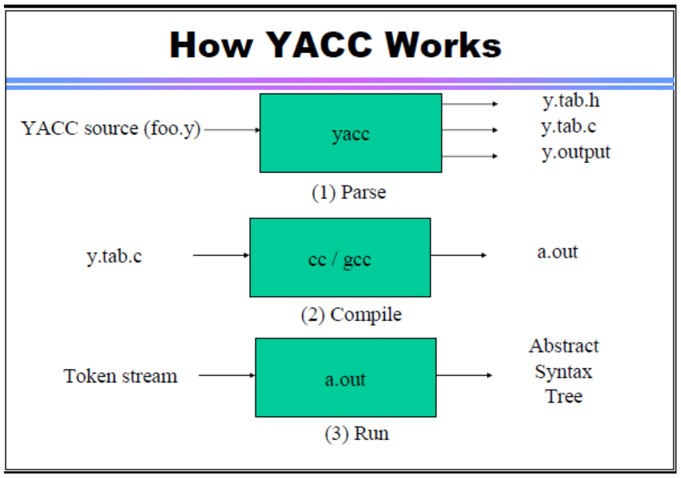


Figure 1.3 Working of YACC

1.4.3 DEV-CPP

Dev-C++ is a free IDE for Windows that uses either MinGW or TDM-GCC as underlying compiler. For our project we have used the Dev-CPP IDE for creating and editing codes. The GCC compiler of Dev-CPP is also used to run the "lex.yy.c" and "parser.tab.c" files generated by compiling the lex program and the YACC program.

Chapter 2

Review of Literature

For the literature review of this project, we reviewed the documentation of Flex, Bison and GCC software and reviewed the documentation of the C language to get a better understanding of the structure, syntax and logic of various sections of the language. For the purpose of this project, we also reviewed 2 projects which have implemented the front-end of the C compiler.

In this section we will be discussing our findings from this review and explain in brief the structure of a Flex file and structure of a YACC file. We will also discuss the existing projects implemented and finalize the functionalities which will be feasible to implement for our project.

2.1 Structure of a Lex file

The structure of our flex script is intentionally similar to that of a YACC file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code. The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

2.2 Structure of a Bison YACC file

A YACC source program is structurally similar to a LEX one.

Declarations

%%

Rules

%%

Routines

- The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.
- Blanks, tabs and newlines are ignored except that they may not appear in names.

Declarations Section

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token
- Declaration of the start symbol using the keyword %start.
- C declarations: included files, global variables, types.
- C code between % {and %}

Rules Section

A rule has the form:

Nonterminal: sentential form

| sentential form

| sentential form

;

2.3 Review of Existing Projects

As mentioned earlier, we have reviewed 2 existing projects which have implemented a mini-C language compiler. In this section we will be discussing the insights we gained from those projects and determine the functionalities that can be feasibly implemented by us.

2.3.1 Mini C Compiler by Kaushik Kalmady and Karthik M

This is a project in which the developers have implemented a C compiler using Flex and Bison on a Linux OS. The development was phased according to the phases of a compiler and documented periodically. This project is for a subset of the C language as the language is very vast and implementing all the functionalities would not be feasible. Some of the functionalities implemented in this project are:

- The keywords identified are: int, long, short, long long, signed, unsigned, for, break, continue, if, else, return.
- Identifiers are identified and added to the symbol table. The rule followed is represented by the regular expression (_|{letter})({letter}|{digit}|_){0,31} .
- Single and multiline comments are identified. Single line comments are identified by //. * Regular expression.

The parser implemented checks for the declaration of variables, type mismatch, scope of the variables, return types of functions, correct use of continue and break statements, redeclaration of variables and functions, etc.

The data structures implemented are symbol table and constant table which are used to store the general information of the code. The data structures used in the source code are hash tables and structs.

2.3.2 Mini C Compiler by Mishal Shah, Samyak Jain and Pavan Vachhani

This is a project by the students of 3rd year Engineering in NIT Surathkal, they too have implemented the project in a phased manner and have used bison and flex as the software for implementation.

The lexical analyzer implemented by them took care of almost all C constructs with some notable ones being:

- The Regex for Identifiers
- Multiline comments should be supported
- Literals
- Error Handling for Incomplete String
- Error Handling for Nested Comments

The Parser was implemented using context free grammar and checked for type checking, handling the scope of the variables, function parameters type checking, number of parameters matching in function call and array dimensionality check along with array index type checking. Intermediate Code Generation was done by generating postfix expressions, 3AC, and syntax trees. They have generated a hash table for storing the symbols with columns for specifying scope, datatypes, parameters for functions, etc.

After a detailed review of these projects and after careful consideration of the time constraints we have identified the problem statement and scope of our project i.e., Implementing a Syntax analyzer for a subset of C language.

Chapter 3

Analysis and Design

3.1 System Flow

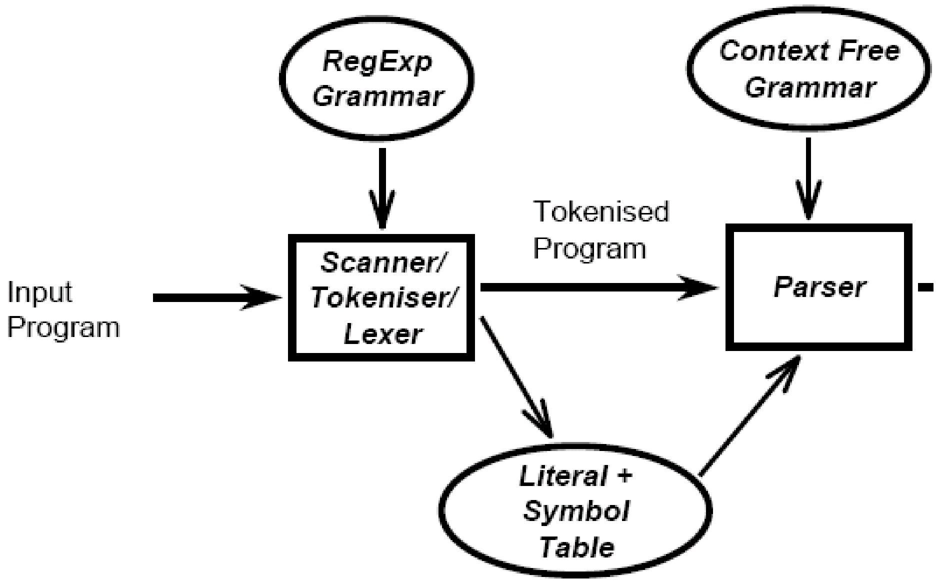


Figure 3.1 System Architecture

This image describes the flow of the system from the user's perspective as well the working of the syntax analyzer. The user inputs a source code in c language to the lexer. This code is then broken down into a stream of tokens and these tokens are stored in a symbol table. This breaking own of the source code is done in accordance to the regular expression defined in the lex program. The tokenized program is then passed to the parser along with the symbol table and the syntax analysis is performed in the parser in accordance with the context free grammar defined in the yacc program.

3.2 The Lexical Analyzer

The lexical analyzer is the first phase to be implemented in our project. In this phase, we defined the keywords, identifiers, function names, operators, return statements, etc. using regular expressions. Once the lexical analyser comes across a pre-defined match, it returns the keyword associated with it to the parser.

3.2.1 Tokens

Tokens are essentially just a group of characters which have some meaning or relation when put together. The Lexical Analyzer detects these tokens with the help of 'Regular Expressions'. While writing the Lexical Analyzer, we have to specify rules for each Token type using Regular Expression. These rules are used to check whether a certain group of characters fall under a given token category or not. An example, in this case, would be an 'Identifier' token. We specify the rules for an identifier as follows: Any string of characters, that start with an _ or an alphabet, followed by any number of _'s, alphabets or numbers. The regular expression for Identifiers is {alpha}({alpha}|{digit}) * where alpha is [a-zA-z_] and digit is [0-9].

3.2.2 Definition Section:

In the definition section of the program, all necessary header files were included. The structure declaration for the symbol table is also included. In order to convert a string of the source program into a particular integer value a hash function was written that takes a string as input and converts it into a particular integer value. Standard table operations like look-up and insert were also written. Linear Probing hashing technique was used to implement the symbol table i.e., if there is a collision, then after the point of collision, the table is searched linearly in order to find an empty slot. Function to print the symbol table was also written.

3.2.3 Rules section:

In this section, rules related to the specification of C language were written in the form of valid regular expressions. E.g., for a valid C identifier, the regex written was [A-Za-z_][A-Za-z_0-9]* which means that a valid identifier need to start with an alphabet or underscore followed by 0 or more occurrence of alphabets, numbers or underscore. In order to resolve conflicts, we used lookahead method of scanner by which a scanner decides whether an expression is valid token or not by looking at its adjacent character. E.g., in order to differentiate between comments and division operator lookahead characters of a valid operator were also given in the regular expression to resolve a conflict. If none of the patterns matched with the input, we said it was a lexical error as it does not match with any valid pattern of the source language. Each character/pattern along with its token class was also printed.

3.3.4 C code section:

In this section the symbol table was initialized to 0 and yylex() function was called to run the program on the given input file. Also multiple functions like void display (), void insertToHash (), int hashIndex, multicomment() and singlecomment() were defined. After that, the symbol table containing the Token and Token type was printed in order to show the result.

The flex script recognizes the following classes of tokens from the input:

- Pre-processor instructions
    - Statements processed: #include<stdio.h>, #define var1 var2
    - Token generated: DEFINE
- Errors in pre-processor instructions
    - Statements processed: #include<stdio.h>, #include<stdio.?
    - Token generated: Error with line number
- Single-line comments
    - Statements processed: //..........
    - Token generated: singlecomment()
- Multi-line comments
    - Statements processed: /*..........*/, /*.../*...*/
    - Token generated: multicomment()
- Errors for unmatched comments
    - Statements processed: /*.........
    - Token generated: Error with line number
- Errors for nested comments
    - Statements processed: /*....../*....*/....*/
    - Token generated: Error with line number
- Parentheses (all types)
    - Statements processed: (..), {..}, [..]
    - Token generated: (,), {,}, [,] respectively
- Operators
    - Token are generated for unary, binary, logical and relational operator
- Literals (integer, float, string)
    - Statements processed: int, float, char
    - Tokens generated: INT, FLOAT, CHAR
- Errors for unclean integers and floating-point numbers
    - Statements processed: 123rf
    - Tokens generated: Error with line number

- Keywords
    - Statements processed: if, else, void, while, do, int, float, break and so on.
    - Tokens generated: Keyword
- Identifiers
    - Statements processed: a, abc, a_b, a12b4
    - Tokens generated: Identifier
- Errors for any invalid character used that is not in C character set.
    - Keywords accounted for: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.

3.3 The Syntax Analyzer

The Yacc program specifies productions for the following:

- Looping construct: while, for, do-while
- Data types: (signed/unsigned) int, float
- Arithmetic and Relational Operators
- Data structure: Arrays
- User defined functions
- Keywords of C language
- Single and Multi-line comments
- Identifiers and Constant errors
- Selection statement: (nested) if-else

The productions for most of them are straight-forward. A few important ones are:

- compound_statement

```
: '{' '}'

| '{' statement_list '}'| '{' declaration_list '}'

| '{' declaration_list statement_list '}'
;
```

- selection_statement

```
: IF '(' expression ')' statement    %prec NO_ELSE
| IF '(' expression ')' statement ELSE statement
;
```

- iteration_statement

```
: WHILE '(' expression ')' statement

| DO statement WHILE '(' expression ')' ';'

| FOR '(' expression_statement expression_statement ')' statement

| FOR '(' expression_statement expression_statement expression ')' statement
;
```

- jump_statement

```
: CONTINUE ';'
| BREAK ';'

| RETURN ';'

| RETURN expression ';'
;
```

- statement

```
: compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
;
```

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ message on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console.

When a grammar is not carefully thought out, the parser generated from the grammar may face two kinds of dilemmas.

3.4 Conflicts and their resolution

3.4.1 Shift-Reduce Conflict:

Enough terms have been read and a grammar rule can be recognized according to the accumulated terms. In this situation, the parser can make a reduction. If, however, there is also another grammar rule which calls for more terms to be accumulated and the look ahead token is just what the second grammar rule expected. In this situation, the parser may also make a shift operation. This dilemma faced by the parser is called the Shift/Reduce Conflict.

3.4.2 Reduce-Reduce Conflict:

Enough terms have been read and two grammar rules are recognized based on the accumulated terms. If the parser then decides to make a reduction, should it reduce the accumulated terms according to the first or second grammar rules? This type of difficulty faced by the parser is called the Reduce/Reduce Conflict.

3.4.3 Handling Shift-Reduce and Reduce-Reduce Conflicts

The yacc command is built with two internal rules for resolving these two ambiguities, sometimes called the disambiguating rules.

1. yacc resolves the Shift/Reduce Conflict in favor of shift operation. In plain english, this just means that it matches the longest input possible.
2. yacc resolves the Reduce/Reduce Conflict in favor of the first grammar rule.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, it is important to write unambiguous grammar with no conflicts.

3.4.4 Solving Dangling Else Problem

Whenever it is not possible to associate an ‘else’ to a closest ‘if’ in if-else statements, it gives rise to dangling else problem. In this case, the problem of dangling else occurs here, represented by #:

IF '(' expression ')' statement # ELSE statement

The question the parser must answer is "should I shift, or should I reduce". Usually, you want to bind the else to the closest if, which means you want to shift the else token now. Reducing now would mean that you want the else to wait to be bound to an older if.

We should specify the parser generator that "when there is a shift/reduce conflict between the token ELSE and the rule “selection_statement -> IF (expression) statement", then the token must win". To do so, a name is given to the precedence of your rule (e.g., NO_ELSE), and specify that NO_ELSE has less precedence than ELSE.

```
//Precedences go increasing, So, NO_ELSE < ELSE
%nonassoc NO_ELSE
%nonassoc ELSE

%%
selection_statement
: IF '(' expression ')' statement
%prec NO_ELSE
| IF '(' expression ')' statement ELSE statement
;
%%
```

Chapter 4

Methods Implemented

- void insertToConstTable (char *num, int l, char *type): This function is called when a constant is identified by the lexer and its respective value and datatype are passed as parameters to the function. It is used to insert the constants and their datatype into a instance array of the structure. The count of constants gets incremented as soon as it is added to the array.

- int hashIndex(char *token): This function is called within the insertToHash function while generating a new node for the token to be added to the symbol table. It is used to generate an index value for the hash-table. A hash function is used to calculate the hash index. For our hash table we have defined the hash function as an addition of the current hashIndex value and the total ASCII value of the token generated by the lexer. Then the hashIndex value is brought within the range by perform modulo division by the hash-table size

- void insertToHash(char *token, char *attr, int l): This function is called when a new node need to be created in the linked list to store a token and its data type. The parameters passed to the function are the token name and its data type. In case the linked list has no nodes, a node is created and the head pointer is assigned to it. In case they are existing nodes then the function traverses through the list to check if there exists a token with same name and checks if there is redeclaration of a variable or not. In case there is redeclaration the latest datatype is overwritten. If not, a new node gets inserted at the beginning and the head pointer points towards the new node.

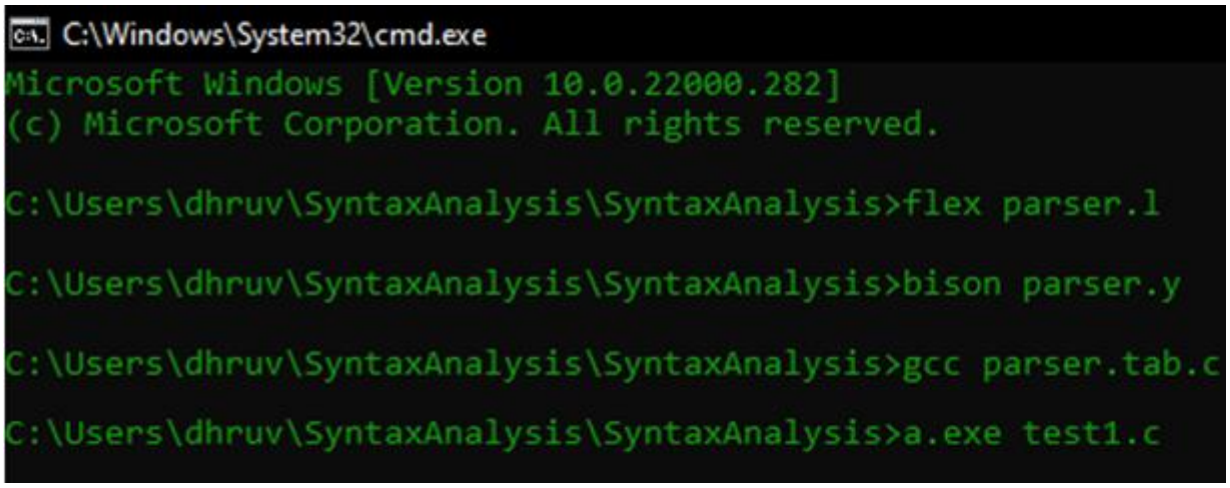
- void display (): This function displays in a tabular format the values stored in the hash table and cnode array as a part of Symbol Tables and Constant Tables respectively.

- multicomment (): This function is called when “/*” is encountered. The function checks if the corresponding “*/” exists or not. If it does, the comments get ignored else the program throws an error.

- singlecomment (): This function is called when “//” is encountered. The function keeps reading the stream until the “\n” is encountered. Once the “\n” is encountered the current line gets ignored as a comment and the function continues to read the next line for parsing.

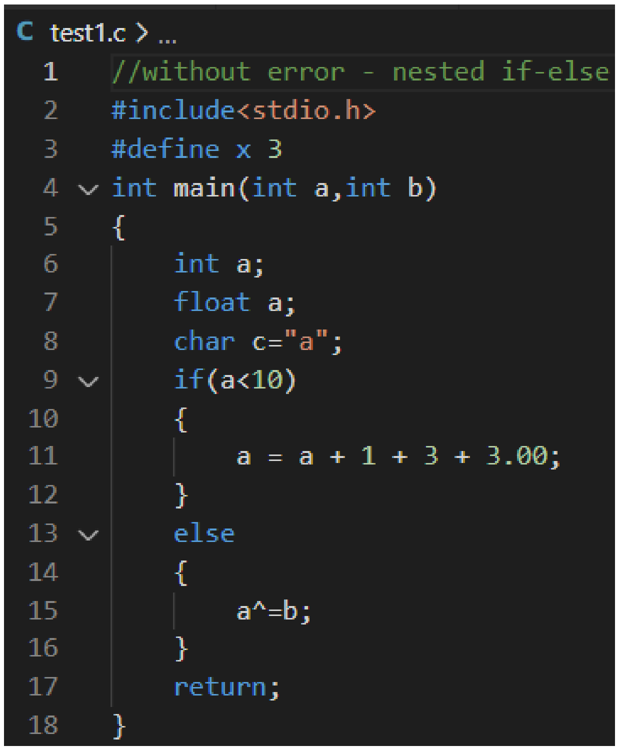
Chapter 5

Results and Discussions



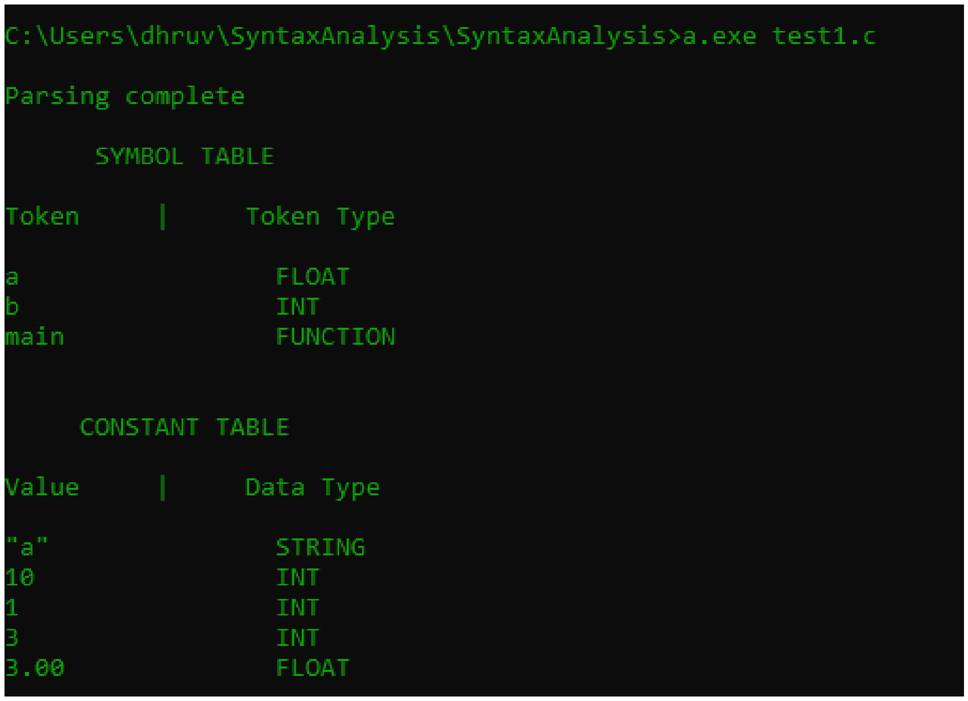
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.22000.282]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dhruv\SyntaxAnalysis\SyntaxAnalysis>flex parser.l

C:\Users\dhruv\SyntaxAnalysis\SyntaxAnalysis>bison parser.y

C:\Users\dhruv\SyntaxAnalysis\SyntaxAnalysis>gcc parser.tab.c

C:\Users\dhruv\SyntaxAnalysis\SyntaxAnalysis>a.exe test1.c
```

Figure 5.1 How to Compile



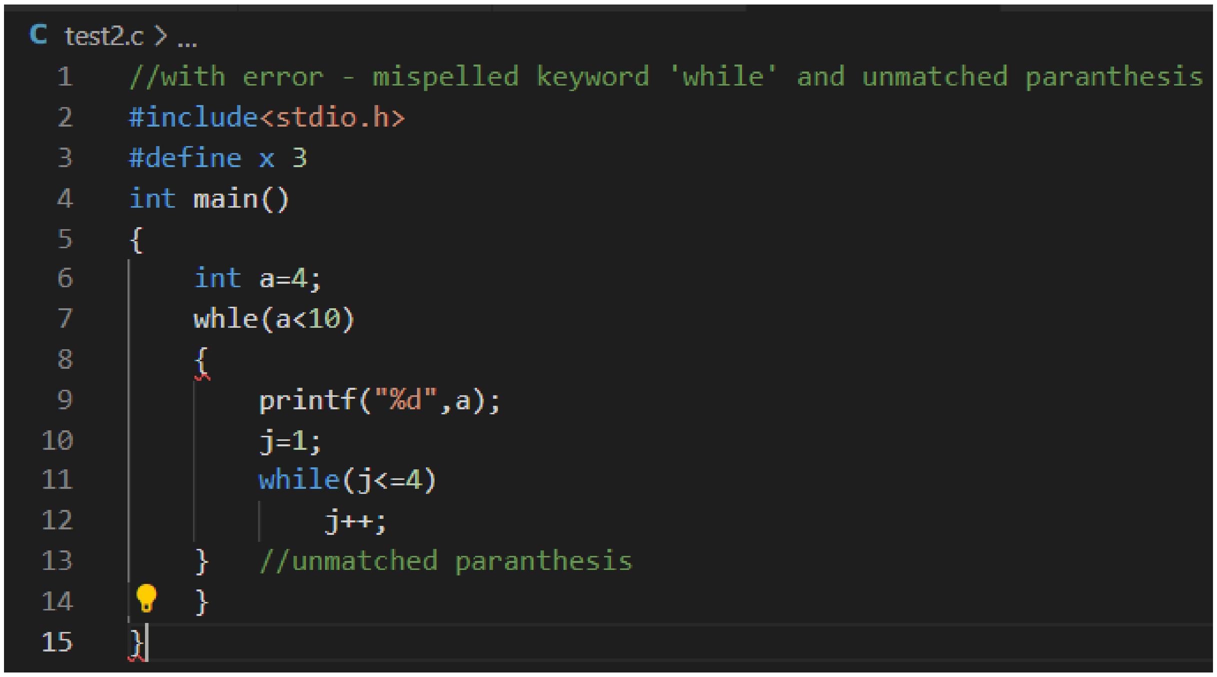
```c
test1.c > ...
//without error - nested if-else
#include<stdio.h>
#define x 3
int main(int a,int b)
{
    int a;
    float a;
    char c="a";
    if(a<10)
    {
        a = a + 1 + 3 + 3.00;
    }
    else
    {
        a^=b;
    }
    return;
}
```

Figure 5.2.1 Test Case 1 Input



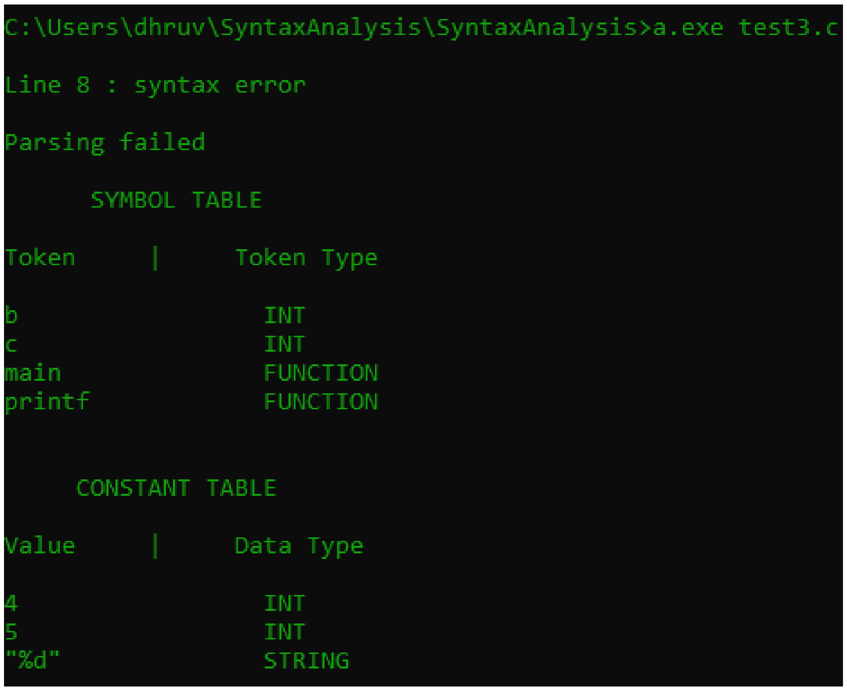
```
C:\Users\dhruv\SyntaxAnalysis\SyntaxAnalysis>a.exe test1.c

Parsing complete

      SYMBOL TABLE

Token      |      Token Type

a                   FLOAT
b                   INT
main                FUNCTION


     CONSTANT TABLE

Value      |      Data Type

"a"                 STRING
10                  INT
1                   INT
3                   INT
3.00                FLOAT
```

Figure 5.2.2 Test Case 1



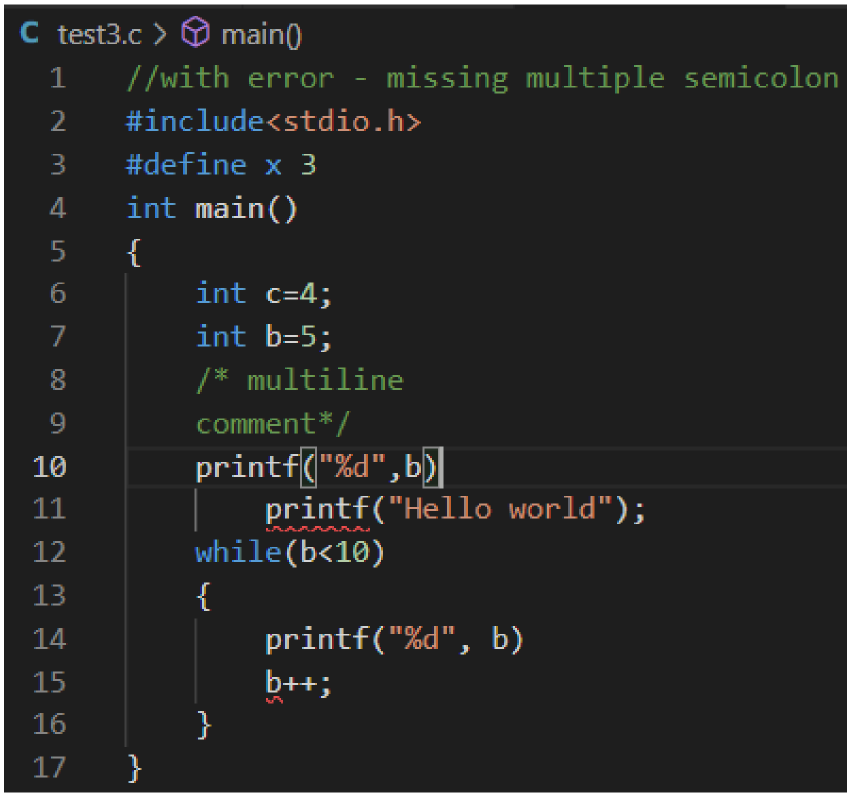
```c
//with error - mispelled keyword 'while' and unmatched paranthesis
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    whle(a<10)
    {
        printf("%d",a);
        j=1;
        while(j<=4)
            j++;
    }   //unmatched paranthesis
    }
}
```

Figure 5.3.1 Test Case 2 Input



```
C:\Users\dhruv\SyntaxAnalysis\SyntaxAnalysis>a.exe test3.c

Line 8 : syntax error

Parsing failed

      SYMBOL TABLE

Token      |     Token Type

b                  INT
c                  INT
main               FUNCTION
printf             FUNCTION


     CONSTANT TABLE

Value      |     Data Type

4                  INT
5                  INT
"%d"               STRING
```

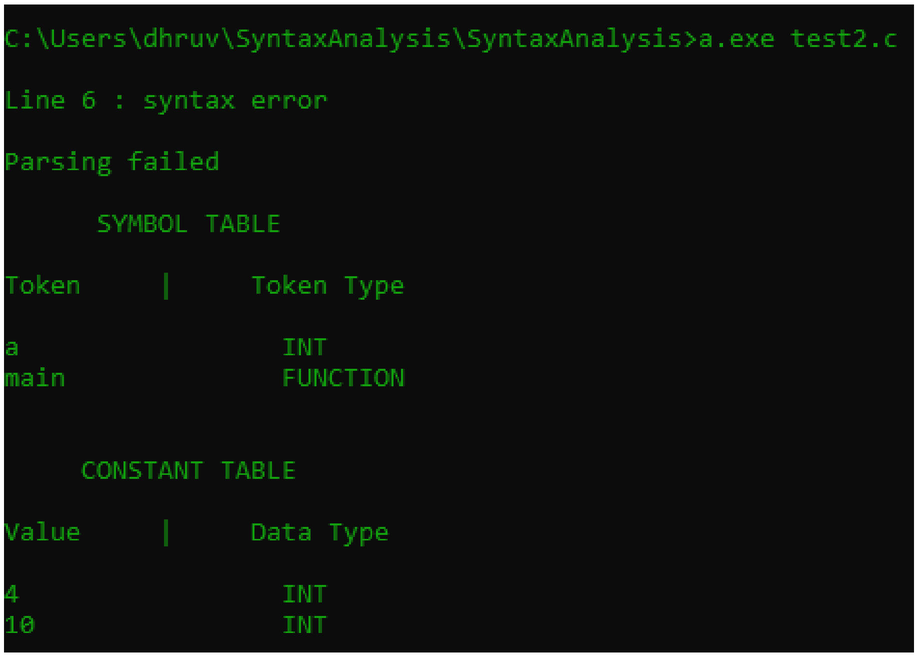
Figure 5.3.2 Test Case 2



C test3.c > main()

```c
//with error - missing multiple semicolon
#include<stdio.h>
#define x 3
int main()
{
    int c=4;
    int b=5;
    /* multiline
    comment*/
    printf("%d",b)
        printf("Hello world");
    while(b<10)
    {
        printf("%d", b)
        b++;
    }
}
```

Figure 5.4.1 Test Case 3 Input



```
C:\Users\dhruv\SyntaxAnalysis\SyntaxAnalysis>a.exe test2.c

Line 6 : syntax error

Parsing failed

      SYMBOL TABLE

Token      |      Token Type

a                   INT
main                FUNCTION


     CONSTANT TABLE

Value      |      Data Type

4                   INT
10                  INT
```

Figure 5.4.2 Test Case 3

Chapter 6

Conclusion And Future Work

To conclude, we have worked on a system which has helped us understand the working of a compiler and how errors are detected in detail. It has also helped us find the answer to a very common question that arises when someone first learns a programming language: “How does the computer interpret the instructions which we provide in the form of code? Is there any code which exists to analyze our own code?” This project has acted as a mode of practical implementation of the concepts learnt in the Compiler Design course. Our project can also be expanded to form a new language which is easy to learn, is faster, has more in-built features and has many more qualities of a good programming language.

FUTURE WORK

During the course of this project, we have observed and noted the following avenues which are a potential topic to work on in the future with the aim of expanding and improving on our project:

1. Semantic Analysis: We plan on working on a Semantic Analyzer to further increase the scope of our project.
2. ICG: We also plan on working on the Intermediate Code Generation phase after completing the Semantic Analysis phase.
3. Custom Syntax Error Messages: In our current project the standard error message for a syntax error is “Syntax Error”. This is because we have used yyerror() which can only provide a standardized message, and we plan on providing custom error messages for specific syntax error cases.
4. Wider range of rules: Adding more rules to our project would make it a much more inclusive compiler in terms of error detection, Symbol Table generation and would also benefit our Semantic Analyzer once it has been completed.
5. Custom Language: Based on the knowledge gained during the course of this project, we also plan on working on a compiler for a custom language which would serve a specific purpose and would be a useful tool in the real world.

References

- https://github.com/westes/flex
- https://www.gnu.org/software/bison/
- https://github.com/mishal23/mini-c-compiler
- https://github.com/kaushiksk/mini-c-compiler
- https://silcnitc.github.io/yacc.html
- https://www.javatpoint.com/lex
- https://www.researchgate.net/publication/330667340_Compiler_report
- https://www.geeksforgeeks.org/phases-of-a-compiler/
- https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html