ABSTRACT

The initial phase of a compiler is lexical analysis. It uses modified source code written in the form of sentences from language preprocessors. By removing any whitespace or comments from the source code, the lexical analyzer breaks these syntaxes down into a series of tokens.

The goal of this activity is to read source code from a file and then produce tokens. The Lexical Analyzer should handle all constructions in execution when provided an input file. During the lex call, the program handles the following constructs:

- Data Types: int, char data types with all its sub-types. Syntax: int a=3;
- Comments: Single line and multiline comments,
- Keywords: char, else, for, if, int, long, return, short, signed, struct, unsigned, void,
- while, main
- Identification of valid identifiers used in the language,
- Looping Constructs: It will support nested for and while loops. Syntax: inti;
- Conditional Constructs: if...else-if...else statements,
- Operators: ADD (+), MULTIPLY (*), DIVIDE (/), MODULO (%), AND (&), OR (|)
- Delimiters: SEMICOLON (;), COMMA (,)

The lexical Analyzer generates an error if a token is found to be incorrect. The lexical Analyzer and the syntax Analyzer work in tandem. When the syntax Analyzer demands it, it pulls character streams from the source code, verifies for legal tokens, and passes the data to it.

Table of Contents

Chapter 1

INTRODUCTION

1.1 Overview

Lexers and parsers are commonly employed in computer technology for compilers, but they can also be used for other computer language tools like pretty printers and linters. The scanning step divides the input string into syntactic units called lexemes, which are then classified into token classes; and the evaluating stage, which translates lexemes into processed values.

Lexers are commonly constructed by a lexer generator, such as lex or derivatives, and are often relatively simple, with most of the complexity postponed until the parser or semantic analysis phases.
Lexers, on the other hand, can sometimes incorporate considerable complexity, such as phrase structure processing to make input easier and simplify the parser, and may be constructed entirely or partially by hand, either to accommodate more features or for performance reasons.
Lexical analysis, in which text or sound waves are segmented into words and other units, is also a crucial early stage in natural language processing. This necessitates a number of non-standardized judgments, and the quantity of tokens produced by systems differs for strings.

The lexical grammar, which defines the lexical syntax, is frequently included in the specification of a computer language. The lexical syntax is usually a regular language with regular expressions as grammar rules; they define a token's set of possible character sequences (lexemes). A lexer recognizes strings, and the lexical program performs an action for each type of string detected, most commonly issuing a token.

A lexical token, also known as a token, is a string that has been allocated and so identifiable with a certain meaning. It's made up of a token name and an optional token value in the form of a pair. The token name is a lexical unit category. Keywords, Identifiers, Operators, and Separators are a few examples.

A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, although scanner is also a term for the first stage of a lexer. A lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth.

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High-level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis

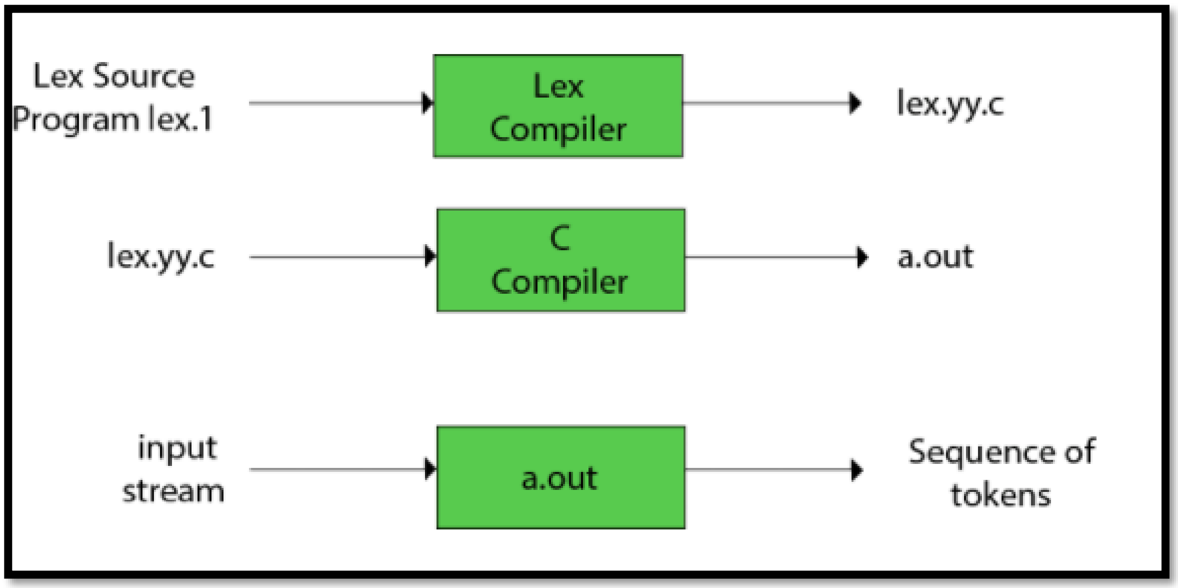
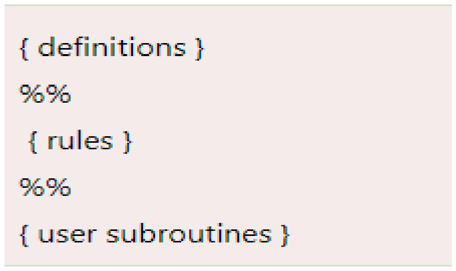


Figure 1.1 Typical function of Lexical Analyzer

1.2 LEX Structure

A lex program consists of three parts:

1. The definition section,
2. The rules section, and
3. The user subroutines.



```
{ definitions }
%%
 { rules }
%%
{ user subroutines }
```

Figure 1.2 Structure of Lex program

The parts are separated by lines consisting of two percent signs. The first two parts are required, although a part may be empty. The third part and the preceding %% line may be omitted.

1.2.1 Definition Section

The definition section can include the literal block, definitions, internal table declarations, start conditions, and translations. Lines that start with whitespace are copied verbatim to the C file. Typically, this is used to include comments enclosed in “/*” and “*/”, preceded by whitespace.

1.2.2 Rules Section

The rules section contains pattern lines and C code. A line that starts with whitespace, or material enclosed in “%{” and “%}” is C code. A line that starts with anything else is a pattern line.

1.2.3 User subroutines

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

1.3 Problem Statement

The goal of this application is to show the basic implementation of a Lexical Analyzer that uses grammar and production to evaluate and identify basic C statements. This is accomplished by executing LEX applications on a compatible system.

Chapter 2

SYSTEM REQUIREMENTS

2.1 Software Requirements

Software requirements define the software resource needs and prerequisites that must be installed on a computer in order for an application to work properly.

The following are the software requirements for the application:

- Operating System: Unix System
- Compiler: GNU C/C++ Compiler

2.2 Hardware Requirements

The most common set of requirements defined by any operating system or
software application is the physical computer resources, also known as hardware.

- CPU: Intel Core I7 10 Generation
- Cores: 6-Core (Quad-Core recommended)
- RAM: 8GB or above (>4GB recommended)
- Graphics: NVIDIA
- Secondary Storage: 20GB or above

2.3 Design

The program is written in the lex construct language. Lex aids with the creation of programmes that transform structured data. This covers a wide range of applications, from a basic text search programme that searches for patterns in its input file to a C compiler that converts source code into optimised object code.

Two jobs that occur often in systems with structured input are separating the input into meaningful units and then discovering the relationship between the units. The units are variable names, constants, strings, operators, punctuation, and so on. Lexical analysis, or lexing for short, is the separation of information into units (typically termed tokens).

Lex assists you by taking a list of probable token descriptions and generating a C function, known as a lexical analyzer, or lexer, or scanner for short, that can detect those tokens. A lex specification is the set of descriptions you give to lex.

Regular expressions are enlarged versions of the familiar patterns used by the grep and egrep tools, which lex utilises to describe tokens. Lex converts these regular expressions into a format that the lexer can employ to scan the input text very quickly, regardless of how many expressions it is trying to match. A lex lexer is always faster than a hand-written lexer written in C.

A software must frequently establish the relationship between tokens because the input is separated into tokens. A compiler must locate the program's expressions, statements, declarations, blocks, and procedures. Parsing is the term for this process, and a grammar is a set of rules that specify the relationships that the software recognises.

Yacc takes a brief description of a grammar and generates a parser, which is a C programme that can parse the grammar. When a sequence of input tokens matches one of the grammar's rules, the yacc parser discovers a syntax error, and when it doesn't match any of the rules, it detects a syntax error.
A yacc parser is often slower than a parser written by hand, but the ease with which it can be written and modified makes up for any speed loss. In any case, the amount of time a program spends in a parser is rarely enough to cause a problem.

Chapter 3

IMPLEMENTATION

3.1 Implementation Code

lexer.l

Code:

```
%{
#include<stdio.h>
extern int yyval;
%}

%%
"#include<stdio.h>" {fprintf(yyout,"%s \tPRE-PROCESSOR DIRECTIVE\n",yytext);}
"/"[a-zA-a' '\t\n]+"/" {fprintf(yyout,"%s \t\tMULTI-LINE COMMENT\n",yytext);}
"//"[a-zA-Z].* {fprintf(yyout,"%s \tSINGLE-LINE COMMENT\n",yytext);}
[0-9]+ {fprintf(yyout,"%s \t\t\tNUMERIC CONSTANT\n",yytext);}
int|float|char|double|void|return|signed|unsigned|main{fprintf(yyout,"%s
\t\t\tKEYWORD\n",yytext);}

"if"|"else"|"else if" {fprintf(yyout,"%s \t\t\tCONDITIONAL STATEMENT\n",yytext);}
"while"|"for"|"do" {fprintf(yyout,"%s \t\t\tLOOPING STATEMENT\n",yytext);}
"printf" {fprintf(yyout,"%s \t\tOUTPUT FUNCTION\n",yytext);}
"scanf" {fprintf(yyout,"%s \t\t\tINPUT FUNCTION\n",yytext);}
"(" {fprintf(yyout,"%c \t\t\tOPENING BRACKET\n",yytext[0]);}
")" {fprintf(yyout,"%c \t\t\tCLOSING BRACKET\n",yytext[0]);}
"{" {fprintf(yyout,"%c \t\t\tOPENING BRACES\n",yytext[0]);}
"}" {fprintf(yyout,"%c \t\t\tCLOSING BRACES\n",yytext[0]);}
";" {fprintf(yyout,"%c \t\t\tSEMICOLON DELIMITER\n",yytext[0]);}
"," {fprintf(yyout,"%c \t\t\tCOMMA DELIMITER\n",yytext[0]);}
"+"|"-"|"*"|"/"|"&"|"=" {fprintf(yyout,"%s \t\t\tOPERATOR\n",yytext);}
"=="|">"|"<"|"!="|">="|"<=" {fprintf(yyout,"%s \t\t\tRELATIONAL
OPERATOR\n",yytext);}

"++" {fprintf(yyout,"%s \t\t\tINCREMENT OPERATOR\n",yytext);}
"--" {fprintf(yyout,"%s \t\t\tDECREMENT OPERATOR\n",yytext);}
"\"%"[dsfuic]"\"" {fprintf(yyout,"%s \t\t\tFORMAT SPECIFIER\n",yytext);}
"\\t" {fprintf(yyout,"%s \t\t\tTAB HORIZONTAL SPACE\n",yytext);}
"\\n" {fprintf(yyout,"%s \t\t\tNEW LINE CHARACTER\n",yytext);}
"\"" {fprintf(yyout,"%s \t\t\tDOUBLE QUOTES\n",yytext);}
[a-zA-Z]+ {fprintf(yyout,"%s \t\t\tIDENTIFIER\n",yytext);}
%%
```

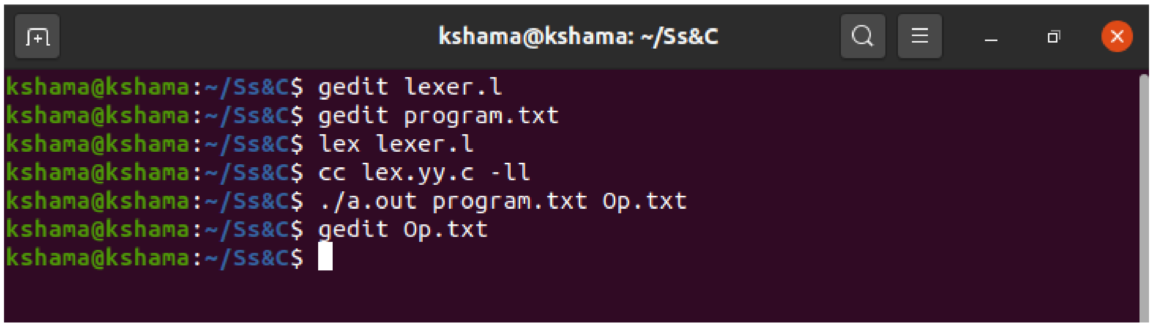
```
void main()
{
yyin=fopen("program.txt","r");
yyout=fopen("Op.txt","w");
yylex();
fclose(yyin);
fclose(yyout);
}
```

program.txt:
Code:

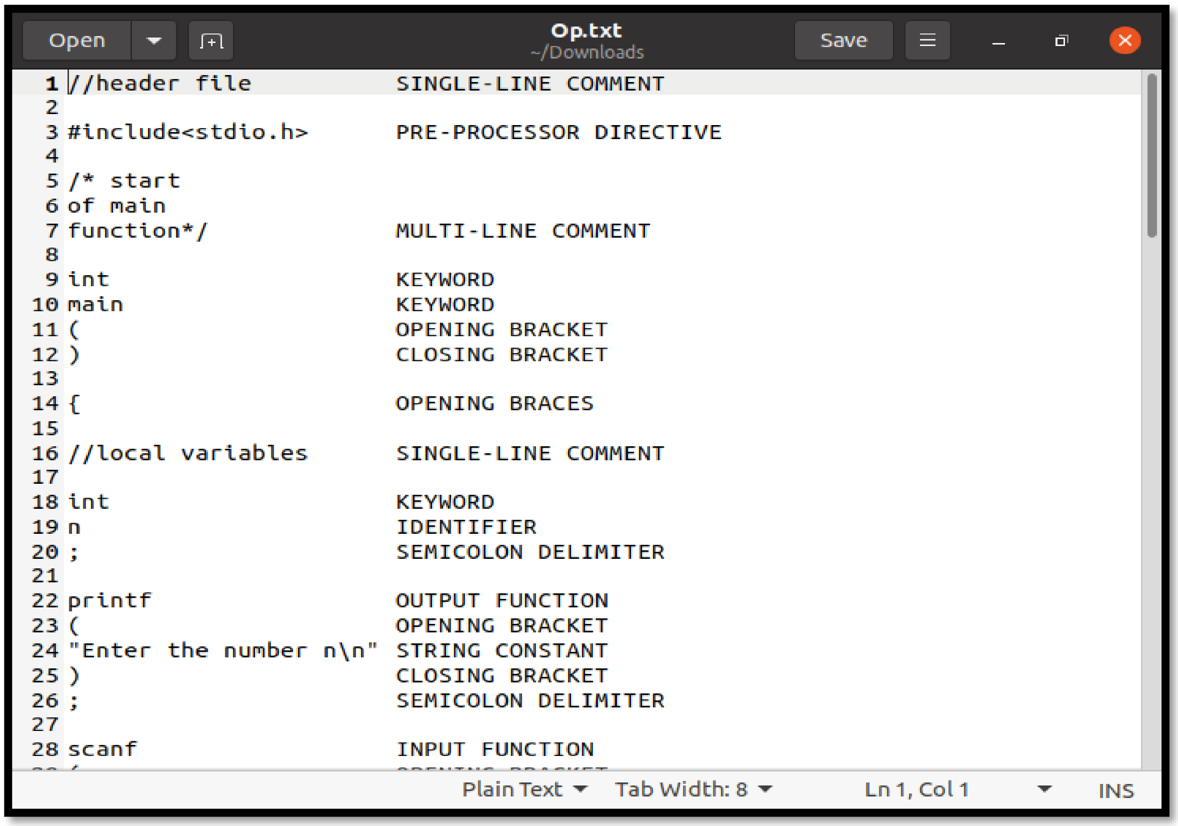
```
//header file
#include<stdio.h>
/* start
of main
function*/
int main()
{
//local variables
int n;
printf("Enter the number n\n");
scanf("%d",&n);
printf("The number is %d\n",n);
for(i=0;i<n;i++)
{
for(j=0;j<i;j++)
{
printf("Okay\n");
}
}
if(i==n)
{
if(j==n)
{
printf("Let's Loop!");
}
}
}
```

Chapter 4

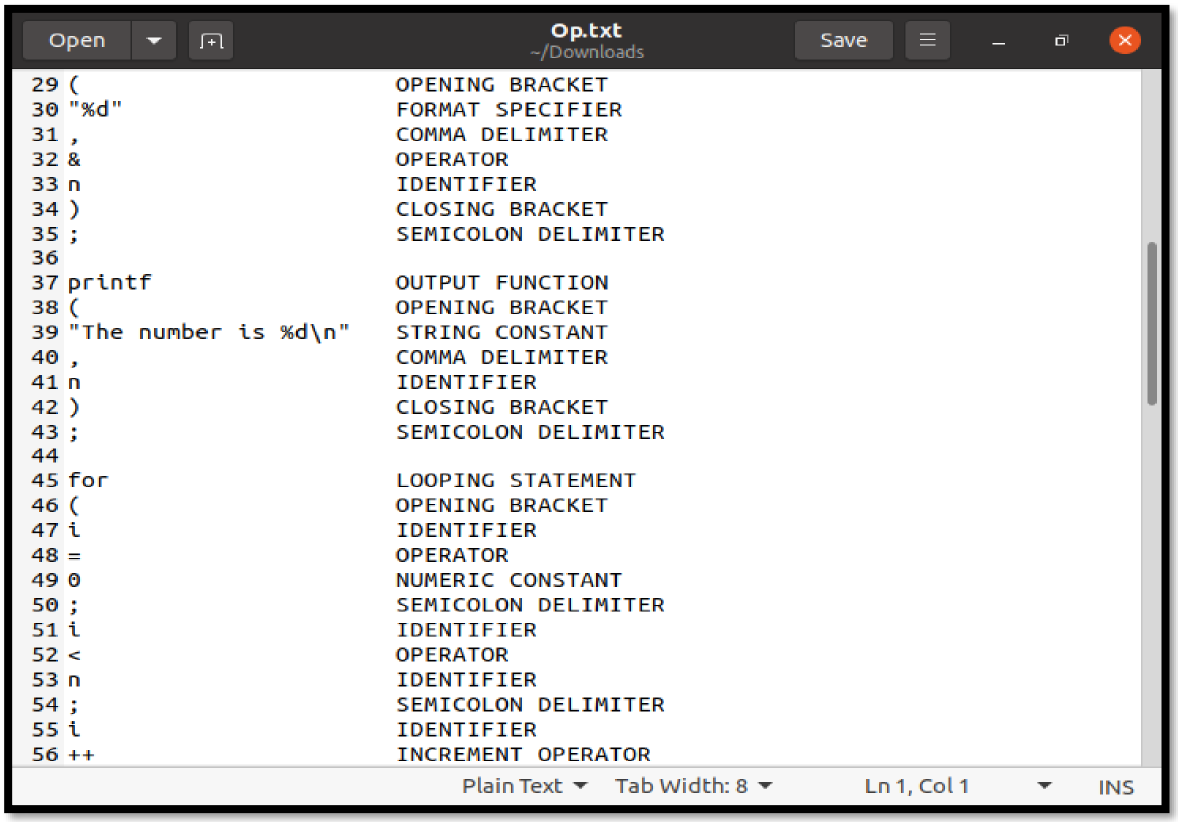
RESULTS



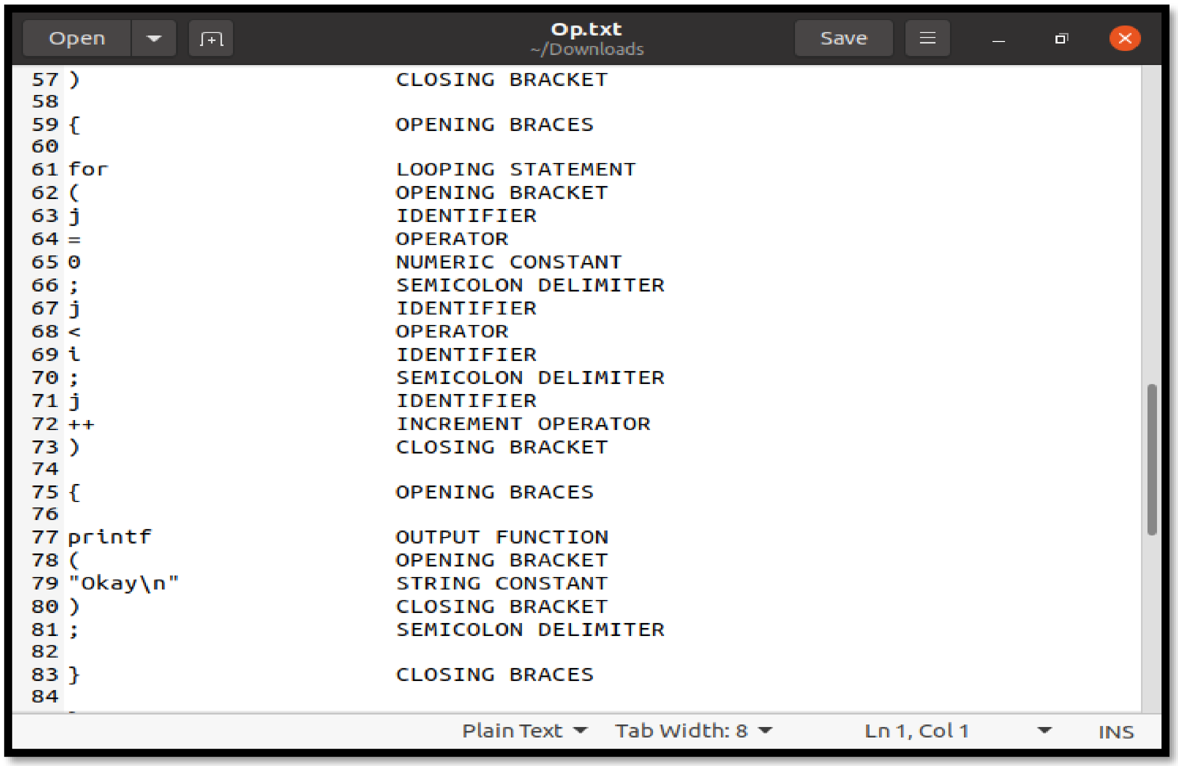
```
kshama@kshama:~/Ss&C$ gedit lexer.l
kshama@kshama:~/Ss&C$ gedit program.txt
kshama@kshama:~/Ss&C$ lex lexer.l
kshama@kshama:~/Ss&C$ cc lex.yy.c -ll
kshama@kshama:~/Ss&C$ ./a.out program.txt Op.txt
kshama@kshama:~/Ss&C$ gedit Op.txt
kshama@kshama:~/Ss&C$ 
```



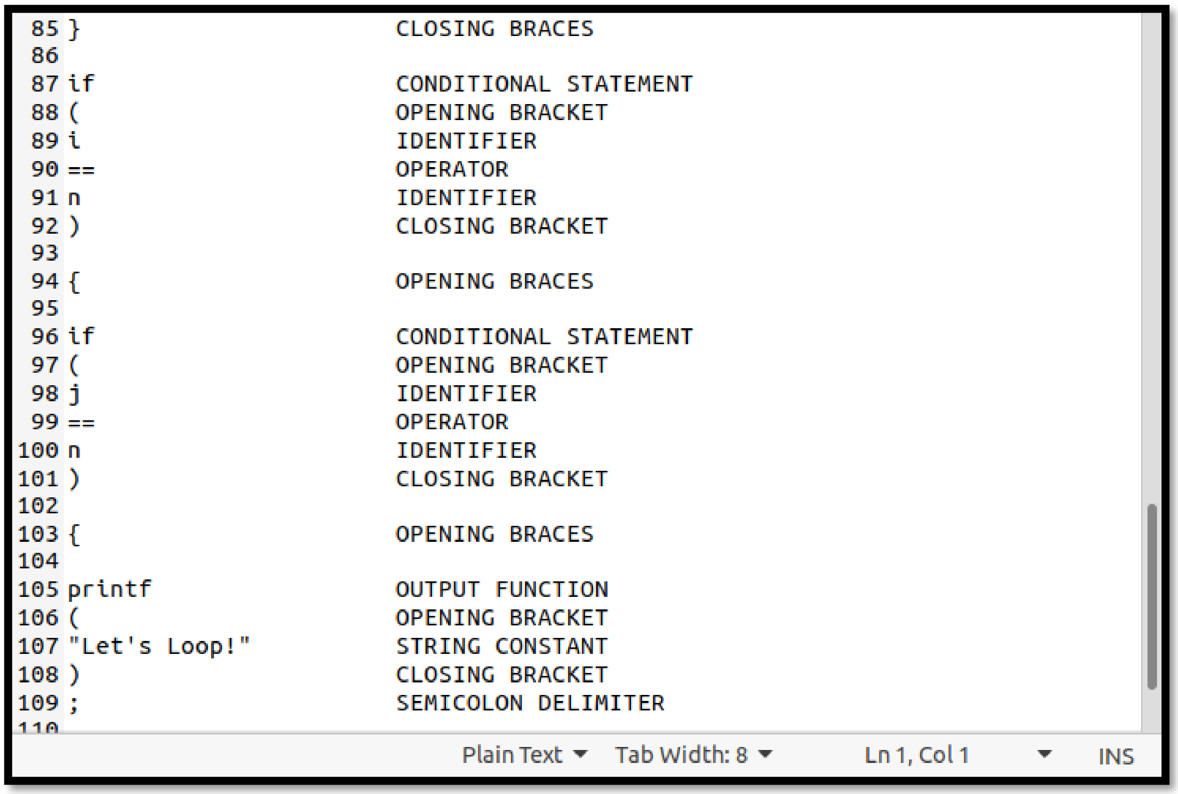
```
//header file             SINGLE-LINE COMMENT

#include<stdio.h>         PRE-PROCESSOR DIRECTIVE

/* start
of main
function*/                MULTI-LINE COMMENT

int                       KEYWORD
main                      KEYWORD
(                         OPENING BRACKET
)                         CLOSING BRACKET

{                         OPENING BRACES

//local variables         SINGLE-LINE COMMENT

int                       KEYWORD
n                         IDENTIFIER
;                         SEMICOLON DELIMITER

printf                    OUTPUT FUNCTION
(                         OPENING BRACKET
"Enter the number n\n"    STRING CONSTANT
)                         CLOSING BRACKET
;                         SEMICOLON DELIMITER

scanf                     INPUT FUNCTION
```



```
(                       OPENING BRACKET
"%d"                    FORMAT SPECIFIER
,                       COMMA DELIMITER
&                       OPERATOR
n                       IDENTIFIER
)                       CLOSING BRACKET
;                       SEMICOLON DELIMITER

printf                  OUTPUT FUNCTION
(                       OPENING BRACKET
"The number is %d\n"    STRING CONSTANT
,                       COMMA DELIMITER
n                       IDENTIFIER
)                       CLOSING BRACKET
;                       SEMICOLON DELIMITER

for                     LOOPING STATEMENT
(                       OPENING BRACKET
i                       IDENTIFIER
=                       OPERATOR
0                       NUMERIC CONSTANT
;                       SEMICOLON DELIMITER
i                       IDENTIFIER
<                       OPERATOR
n                       IDENTIFIER
;                       SEMICOLON DELIMITER
i                       IDENTIFIER
++                      INCREMENT OPERATOR
```

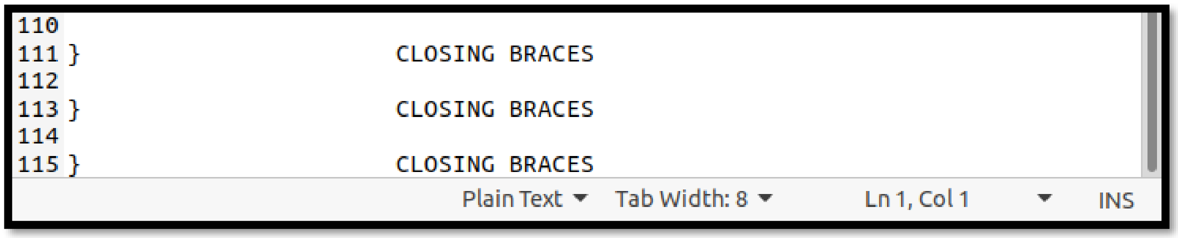


```
)                       CLOSING BRACKET

{                       OPENING BRACES

for                     LOOPING STATEMENT
(                       OPENING BRACKET
j                       IDENTIFIER
=                       OPERATOR
0                       NUMERIC CONSTANT
;                       SEMICOLON DELIMITER
j                       IDENTIFIER
<                       OPERATOR
i                       IDENTIFIER
;                       SEMICOLON DELIMITER
j                       IDENTIFIER
++                      INCREMENT OPERATOR
)                       CLOSING BRACKET

{                       OPENING BRACES

printf                  OUTPUT FUNCTION
(                       OPENING BRACKET
"Okay\n"                STRING CONSTANT
)                       CLOSING BRACKET
;                       SEMICOLON DELIMITER

}                       CLOSING BRACES

```



```
}                       CLOSING BRACES

if                      CONDITIONAL STATEMENT
(                       OPENING BRACKET
i                       IDENTIFIER
==                      OPERATOR
n                       IDENTIFIER
)                       CLOSING BRACKET

{                       OPENING BRACES

if                      CONDITIONAL STATEMENT
(                       OPENING BRACKET
j                       IDENTIFIER
==                      OPERATOR
n                       IDENTIFIER
)                       CLOSING BRACKET

{                       OPENING BRACES

printf                  OUTPUT FUNCTION
(                       OPENING BRACKET
"Let's Loop!"           STRING CONSTANT
)                       CLOSING BRACKET
;                       SEMICOLON DELIMITER
```

Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS



```

}                       CLOSING BRACES

}                       CLOSING BRACES

}                       CLOSING BRACES
```

Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

CONCLUSION

The project "LEXICAL ANALYZER" was created with the best of our efforts in a short amount of time. The package's functionalities have been implemented and tested to ensure operational efficiency, and they have shown to be fairly satisfactory. After finishing the project's development and research, I've come to the conclusion that Lexers can be utilized to create more efficient tools for identifying all structures in a C program.

The following are some of the future scopes:

- The project can be developed specifically for different types of constructs.
- It can be used with Parsers to demonstrate the working of Parsers

REFERENCES

- https://www.geeksforgeeks.org/introduction-to-yacc/
- https://en.wikipedia.org/wiki/Lex_(software)
- https://silcnitc.github.io/ywl.html
- https://www.oreilly.com/library/view/lex-yacc/9781565920002/ch01.html