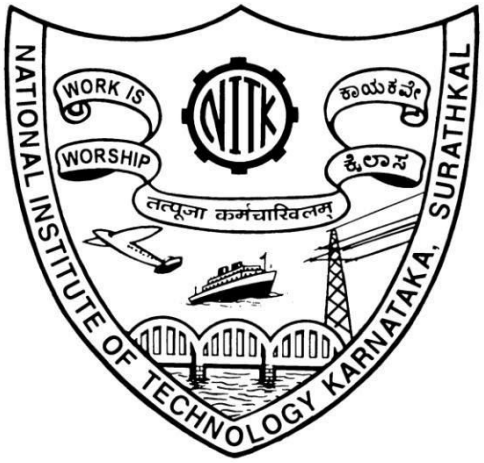
Lexical Analyzer for the C Language



National Institute of Technology Karnataka Surathkal

Date: 21-01-2020

Submitted To:
Vinnakota Saran Chaitanya

Group Members:
Chandan Naik - 17CO212
Jelwin Rodrigues - 17CO218
Siddhartha M - 17CO246

Abstract:

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Phases of Compiler

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.
The phases are as below:
Analysis

1. Lexical Analysis:
2. Parsing:
3. Semantic Analysis:
4. Intermediate Code Generation:

Synthesis

1. Code Optimization:
2. Code Generation:

Objectives:

This project aims to undertake a sequence of experiments to design and implement various phases of a compiler for the C programming language. Following constructs will be handled by the mini-compiler :

1. Data Types: int , char data types with all its sub-types. Syntax : int a=3;
2. Comments: Single line and multiline comments,
3. Keywords: char, else, for, if, int, long, return, short, signed, struct, unsigned, void, while, main.
4. Identification of valid identifiers used in the language,
5. Looping Constructs: It will support nested for and while loops. Syntax: int i; for(i=0;i<n;i++){ } int x; while(x<10){ ... x++}
6. Conditional Constructs: if...else-if...else statements,
7. Operators: ADD(+), MULTIPLY(*), DIVIDE(/), MODULO(%), AND(&), OR(|)
8. Delimiters: SEMICOLON(;), COMMA(,)
9. Structure construct of the language, Syntax: struct pair{ int a; int b};
10. Function construct of the language, Syntax: int func(int x)
11. Support of nested conditional statement,
12. Support for a 1-Dimensional array. Syntax : char a[10];

Contents:

List of Figures and Tables:

Introduction

Lexical Analysis

The Lexical Analyzer is the first phase of the Analysis (front end) stage of a compiler. In layman's terms, the Lexical Analyzer (or Scanner) scans through the input source program character by character, and identifies 'Lexemes' and categorizes them into 'Tokens'. These 'tokens' are represented as a symbol table, and is given as input to the Parser (second phase of the front end of a compiler).

Flex Script

The script written by us is a program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

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.

C Program

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in the account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input.

Design of Program:

Code:

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

      int size=1002;

      struct Table
      {
            char name[100];
            char type[100];
            int length;
      }symTbl[1002],constTbl[1002];

      int hash(char *str);
      int searchSymTbl(char *str);
      int searchConstTbl(char *str);
      void insertSymTbl(char *str1, char *str2);
      void insertConstTbl(char *str1, char *str2);
      void printSymTbl();
      void printConstTbl();

%}

DE "define"
IN "include"

operator
[[<][=]|[>][=]|[=][=]|[!][=]|[>]|[<]|[\|][\|]|[&][&]|[\!]|[=]|[\^]|[\+][=]
|[\-][=]|[\*][=]|[\/][=]|[\%][=]|[\+][\+]|[\-][\-]|[\+]|[\-]|[\*]|[\/]|[\%
]|[&]|[\|]|[~]|[<][<]|[>][>]]


%%
\n   {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|"
"|"\t"] {printf("%s \t-Pre Processor directive\n",yytext);} //Matches
#include<stdio.h>
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]
{printf("%s \t-Macro\n",yytext);} //Matches macro
\/\/(.*) {printf("%s \t- SINGLE LINE COMMENT\n", yytext);}

\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/  {printf("%s \t- MULTI LINE
COMMENT\n", yytext);}
```

```
[ \n\t] ;
; {printf("%s \t- SEMICOLON DELIMITER\n", yytext);}
, {printf("%s \t- COMMA DELIMITER\n", yytext);}
\{ {printf("%s \t- OPENING BRACES\n", yytext);}
\} {printf("%s \t- CLOSING BRACES\n", yytext);}
\( {printf("%s \t- OPENING BRACKETS\n", yytext);}
\) {printf("%s \t- CLOSING BRACKETS\n", yytext);}
\[ {printf("%s \t- SQUARE OPENING BRACKETS\n", yytext);}
\] {printf("%s \t- SQUARE CLOSING BRACKETS\n", yytext);}
\: {printf("%s \t- COLON DELIMITER\n", yytext);}
\\ {printf("%s \t- FSLASH\n", yytext);}
\. {printf("%s \t- DOT DELIMITER\n", yytext);}
auto|break|case|char|const|continue|default|do|double|else|enum|extern|flo
at|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|
switch|typedef|union|unsigned|void|volatile|while|main/[\(|"
"|\{|;|:|"\n"|"\t"] {printf("%s \t- KEYWORD\n", yytext);
insertSymTbl(yytext, "KEYWORD");}
\"[^\n]*\"/[;|,|\)] {printf("%s \t- STRING CONSTANT\n", yytext);
insertConstTbl(yytext,"STRING CONSTANT");}
\'[A-Z|a-z]\'/[;|,|\)|:] {printf("%s \t- Character CONSTANT\n", yytext);
insertConstTbl(yytext,"Character CONSTANT");}
[a-z|A-Z|_]([a-z|A-Z|_]|[0-9])*/\[[1-9][0-9]*\] {printf("%s \t- ARRAY
IDENTIFIER\n", yytext); insertSymTbl(yytext, "IDENTIFIER");}

{operator}/[a-z]|[0-9]|;|" "|[A-Z]|\(|\"|\'|\)|\n|\t {printf("%s \t-
OPERATOR\n", yytext);}

[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{printf("%s \t- NUMBER CONSTANT\n", yytext); insertConstTbl(yytext,
"NUMBER CONSTANT");}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{printf("%s \t- Floating CONSTANT\n", yytext); insertConstTbl(yytext,
"Floating CONSTANT");}
[A-Za-z_][A-Za-z_0-9]*/["
"|;|,|\(|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\.|\{|\^|\t] {printf("%s \t-
IDENTIFIER\n", yytext); insertSymTbl(yytext, "IDENTIFIER");}


(.?) {
          if(yytext[0]=='#')
          {
               printf("line No %d :Error in Pre-Processor
directive\n",yylineno);
          }
         else if(yytext[0]=='/')
         {
```

```
                printf("line No %d :ERR_UNMATCHED_COMMENT\n",yylineno);
          }
          else if(yytext[0]=='"')
          {
                printf("line No %d :ERR_INCOMPLETE_STRING\n",yylineno);
          }
          else
          {
                printf("line No %d :ERROR\n",yylineno);
          }
          printf("%s\n", yytext);
          return 0;
}

%%

int main(int argc , char **argv){


     int i;
     for (i=0;i<size;i++){
          symTbl[i].length=0;
          constTbl[i].length=0;
     }

     yyin = fopen("input.c","r");
     yylex();

     printf("\n\nSYMBOL TABLE\n\n");
     printSymTbl();
     printf("\n\nCONSTANT TABLE\n\n");
     printConstTbl();
}

int yywrap(){
    return 1;
}

int hash(char *str)
{
     int value = 0;
     for(int i = 0 ; i < strlen(str) ; i++)
     {
          value = 10*value + (str[i] - 'A');
          value = value % size;
          while(value < 0)
```

```
                value = value + size;
        }
        return value;
}

int searchSymTbl(char *str)
{
        int value = hash(str);
        if(symTbl[value].length == 0)
        {
                return 0;
        }
        else if(strcmp(symTbl[value].name,str)==0)
        {
                return 1;
        }
        else
        {
                for(int i = value + 1 ; i!=value ; i = (i+1)%size)
        {
                if(strcmp(symTbl[i].name,str)==0)
                {
                        return 1;
                }
        }
        return 0;
      }
}

int searchConstTbl(char *str)
{
        int value = hash(str);
        if(constTbl[value].length == 0)
                return 0;
        else if(strcmp(constTbl[value].name,str)==0)
                return 1;
        else
        {
                for(int i = value + 1 ; i!=value ; i = (i+1)%size)
                {
                        if(strcmp(constTbl[i].name,str)==0)
                        {
                                return 1;
                        }
                }
                return 0;
```

```
	}
}

void insertSymTbl(char *str1, char *str2)
{
	if(searchSymTbl(str1))
	{
	    return;
	}
	else
	{
		int value = hash(str1);
		if(symTbl[value].length == 0)
		{
			strcpy(symTbl[value].name,str1);
			strcpy(symTbl[value].type,str2);
			symTbl[value].length = strlen(str1);
			return;
		}

		int pos = 0;

		for (int i = value + 1 ; i!=value ; i = (i+1)%size)
		{
			if(symTbl[i].length == 0)
			{
				pos = i;
				break;
			}
		}

		strcpy(symTbl[pos].name,str1);
		strcpy(symTbl[pos].type,str2);
		symTbl[pos].length = strlen(str1);
	}
}

void insertConstTbl(char *str1, char *str2)
{
	if(searchConstTbl(str1))
		return;
	else
	{
		int value = hash(str1);
		if(constTbl[value].length == 0)
		{
```

```
                strcpy(constTbl[value].name,str1);
                strcpy(constTbl[value].type,str2);
                constTbl[value].length = strlen(str1);
                return;
          }

          int pos = 0;

          for (int i = value + 1 ; i!=value ; i = (i+1)%size)
          {
                if(constTbl[i].length == 0)
                {
                      pos = i;
                      break;
                }
          }

          strcpy(constTbl[pos].name,str1);
          strcpy(constTbl[pos].type,str2);
          constTbl[pos].length = strlen(str1);
     }
}

void printSymTbl()
{
     for(int i = 0 ; i < size ; i++)
     {
          if(symTbl[i].length == 0)
          {
                continue;
          }

          printf("%s\t%s\n",symTbl[i].name, symTbl[i].type);
     }
}

void printConstTbl()
{
     for(int i = 0 ; i < size ; i++)
     {
          if(constTbl[i].length == 0)
                continue;

          printf("%s\t%s\n",constTbl[i].name, constTbl[i].type);
     }
}
```

Explanation:

Definition Section:
In the definition section of the program, all necessary header files were included. Apart from that structure declaration for both the symbol table and constant table were made. 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. Functions to print the symbol table and constant table were also written.

Rules section:
In this section rules related to the specification of C language were written in the form Page 12 of valid regular expressions. E.g. for a valid C identifier the regex is written was [A-Za-z_][A-Za-z_0-9]* which means that a valid identifier needs 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 the lookahead method of the scanner by which a scanner decides whether an expression is a 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 is 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.

C code section:
In this section both, the symbol table and the constants table are initialized to 0 and yylex() function was called to run the program on the given input file. After that, both the symbol table and the constant table were generated and printed to show the result.

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

- Pre-processor instructions
  - #include<stdio.h>
  - #define a 3
- Single-line comments
  - `//..........`
- Multi-line comments
  - `/*..........*/`
  - `/*.../*...*/`
- Errors for unmatched comments
  - `/*.........`
- Errors for nested comments
  - `/*....../*....*/....*/`
- Parentheses (all types)
  - `(..), {..}, [..]`
- Operators
  - `+, -, =`
- Literals (integer, float, string)
  - `int, float, char`
- Errors for unclean integers and floating-point numbers
  - 346ab
- Errors for incomplete strings
  - Char a[]=”dyhb
- Keywords
  - If, else , while, void , do
- Identifiers
  - a, abc, a_b, a12b4

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, main.

Test Cases:

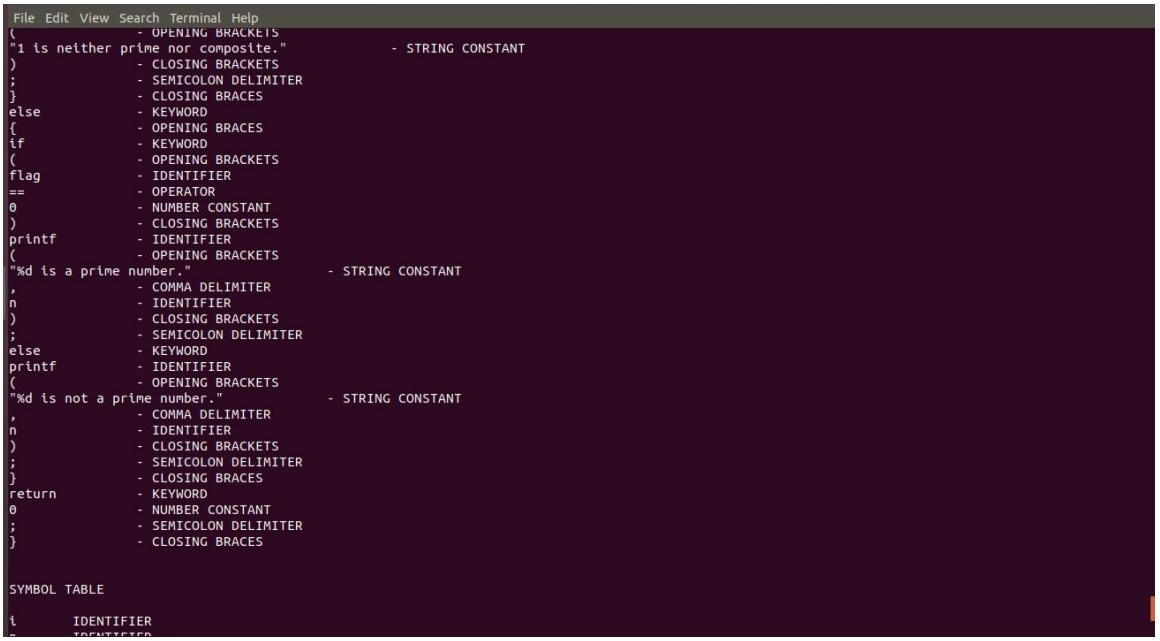
Test 1: **Error-free code**
(datatypes, keywords, identifiers, for loop, nested-if, if statement, single-line comment, multiline comment, conditional statement etc.)

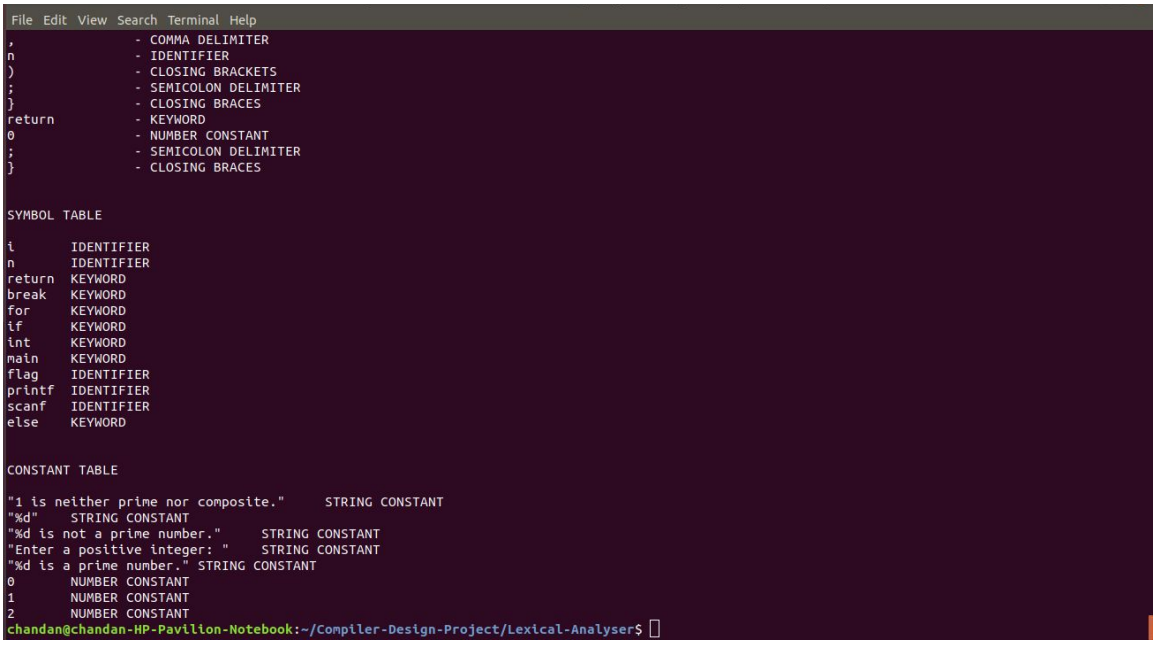
```c
#include <stdio.h>
int main() {
    int n, i, flag = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    for (i = 2; i <= n / 2; ++i) {
        // condition for non-prime
        if (n % i == 0) {
            flag = 1;
            break;
        }
    }
    /* if flag is 0 then number has no multiples
       if flag is 1 then it has atleast one multiple */
    if (n == 1) {
        printf("1 is neither prime nor composite.");
    }
    else {
        if (flag == 0)
            printf("%d is a prime number.", n);
        else
            printf("%d is not a prime number.", n);
    }
    return 0;
}
```

Output:

```
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$ ./a.out
#include <stdio.h>              -Pre Processor directive
int             - KEYWORD
main            - KEYWORD
(               - OPENING BRACKETS
)               - CLOSING BRACKETS
{               - OPENING BRACES
int             - KEYWORD
n               - IDENTIFIER
,               - COMMA DELIMITER
i               - IDENTIFIER
,               - COMMA DELIMITER
flag            - IDENTIFIER
=               - OPERATOR
0               - NUMBER CONSTANT
;               - SEMICOLON DELIMITER
printf          - IDENTIFIER
(               - OPENING BRACKETS
"Enter a positive integer: "            - STRING CONSTANT
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
scanf           - IDENTIFIER
(               - OPENING BRACKETS
"%d"            - STRING CONSTANT
,               - COMMA DELIMITER
&               - OPERATOR
n               - IDENTIFIER
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
for             - KEYWORD
(               - OPENING BRACKETS
i               - IDENTIFIER
=               - OPERATOR
2               - NUMBER CONSTANT
;               - SEMICOLON DELIMITER
i               - IDENTIFIER
<=              - OPERATOR
n               - IDENTIFIER
```

```
File  Edit  View  Search  Terminal  Help
<=              - OPERATOR
n               - IDENTIFIER
/               - OPERATOR
2               - NUMBER CONSTANT
;               - SEMICOLON DELIMITER
++              - OPERATOR
i               - IDENTIFIER
)               - CLOSING BRACKETS
{               - OPENING BRACES
// condition for non-prime              - SINGLE LINE COMMENT
if              - KEYWORD
(               - OPENING BRACKETS
n               - IDENTIFIER
%               - OPERATOR
i               - IDENTIFIER
==              - OPERATOR
0               - NUMBER CONSTANT
)               - CLOSING BRACKETS
{               - OPENING BRACES
flag            - IDENTIFIER
=               - OPERATOR
1               - NUMBER CONSTANT
;               - SEMICOLON DELIMITER
break           - KEYWORD
;               - SEMICOLON DELIMITER
}               - CLOSING BRACES
}               - CLOSING BRACES
/* if flag is 0 then number has no multiples
      if flag is 1 then it has atleast one multiple */               - MULTI LINE COMMENT
if              - KEYWORD
(               - OPENING BRACKETS
n               - IDENTIFIER
==              - OPERATOR
1               - NUMBER CONSTANT
)               - CLOSING BRACKETS
{               - OPENING BRACES
printf          - IDENTIFIER
(               - OPENING BRACKETS
"1 is neither prime nor composite "             - STRING CONSTANT
```





STATUS: PASS

Test 2: Error-free code
(identifiers, while loop, nested-if, if statement, single line comment, struct statements, Functions,arrays)

```
// a linked list
#include <stdio.h>
#include <stdlib.h>

#define size 10
struct pair{
    int a;
    int b;
```

```
};

int fun(int x){
    return x*x;
}

int main(){
    int a=2,b,c,d,e,f,g,h;


    char s[10]="Welcome!!";
    char s[]="Welcome!!";
    int a[2] = {1, 2};
    char S[20];

    int p;
    if(s[0]=='W'){
        if(s[1]=='e'){
            if(s[2]=='l'){
                printf("Welcome!!");
            }

            else printf("Bug1\n");
        }
        else printf("Bug2\n");
    }

    else printf("Bug3\n");

    int i=size;

    while(i--)
    {
        printf("hello world\n");
    }

}
```

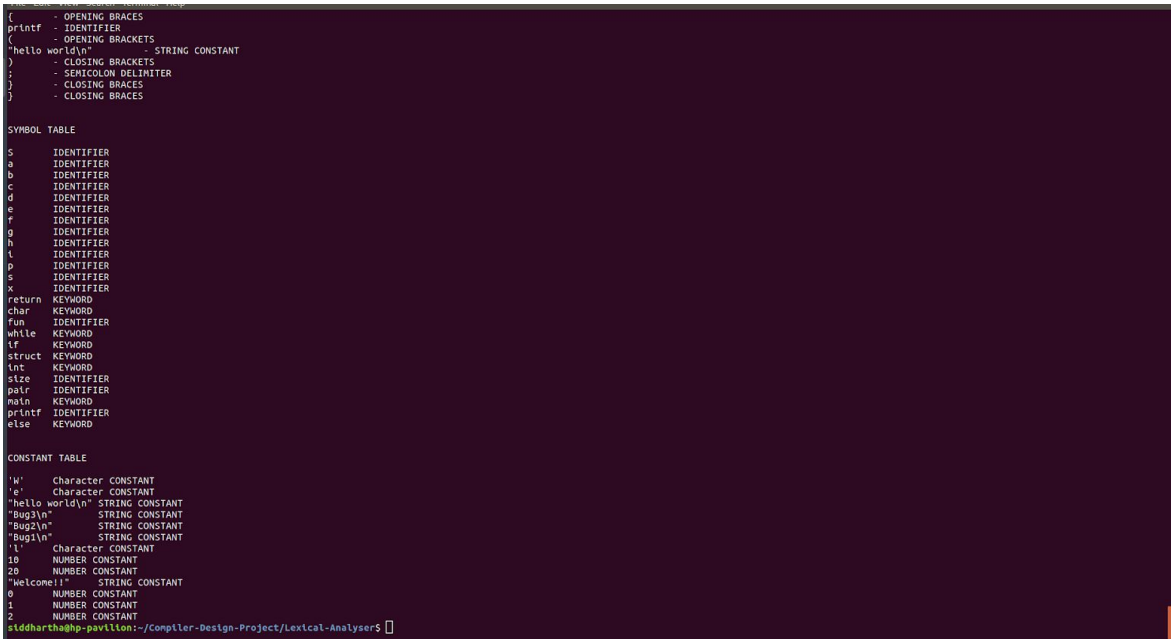
Output:

STATUS = PASS

```
siddhartha@hp-pavilion:~/Compiler-Design-Project/Lexical-Analyser$ gcc lex.yy.c
siddhartha@hp-pavilion:~/Compiler-Design-Project/Lexical-Analyser$ ./a.out
// a linked list        - SINGLE LINE COMMENT
#include <stdio.h>      -Pre Processor directive
#include <stdlib.h>     -Pre Processor directive
#define size 10         -Macro
struct  - KEYWORD
pair    - IDENTIFIER
{       - OPENING BRACES
int     - KEYWORD
a       - IDENTIFIER
;       - SEMICOLON DELIMITER
int     - KEYWORD
b       - IDENTIFIER
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
;       - SEMICOLON DELIMITER
int     - KEYWORD
fun     - IDENTIFIER
(       - OPENING BRACKETS
int     - KEYWORD
x       - IDENTIFIER
)       - CLOSING BRACKETS
{       - OPENING BRACES
return  - KEYWORD
x       - IDENTIFIER
+       - OPERATOR
x       - IDENTIFIER
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
int     - KEYWORD
main    - KEYWORD
(       - OPENING BRACKETS
)       - CLOSING BRACKETS
{       - OPENING BRACES
int     - KEYWORD
a       - IDENTIFIER
=       - OPERATOR
2       - NUMBER CONSTANT
,       - COMMA DELIMITER
b       - IDENTIFIER
,       - COMMA DELIMITER
c       - IDENTIFIER
,       - COMMA DELIMITER
d       - IDENTIFIER
,       - COMMA DELIMITER
e       - IDENTIFIER
,       - COMMA DELIMITER
f       - IDENTIFIER
,       - COMMA DELIMITER
g       - IDENTIFIER
,       - COMMA DELIMITER
h       - IDENTIFIER
;       - SEMICOLON DELIMITER
char    - KEYWORD
```

```
char    - KEYWORD
s       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
10      - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
=       - OPERATOR
"Welcome!!"     - STRING CONSTANT
;       - SEMICOLON DELIMITER
char    - KEYWORD
s       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
]       - SQUARE CLOSING BRACKETS
=       - OPERATOR
"Welcome!!"     - STRING CONSTANT
;       - SEMICOLON DELIMITER
int     - KEYWORD
a       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
2       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
=       - OPERATOR
{       - OPENING BRACES
1       - NUMBER CONSTANT
,       - COMMA DELIMITER
2       - NUMBER CONSTANT
}       - CLOSING BRACES
;       - SEMICOLON DELIMITER
char    - KEYWORD
S       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
20      - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
;       - SEMICOLON DELIMITER
int     - KEYWORD
p       - IDENTIFIER
;       - SEMICOLON DELIMITER
if      - KEYWORD
(       - OPENING BRACKETS
s       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
0       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
==      - OPERATOR
'W'     - Character CONSTANT
)       - CLOSING BRACKETS
{       - OPENING BRACES
if      - KEYWORD
(       - OPENING BRACKETS
s       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
1       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
==      - OPERATOR
'e'     - Character CONSTANT
)       - CLOSING BRACKETS
{       - OPENING BRACES
```

```
)       - CLOSING BRACKETS
{       - OPENING BRACES
if      - KEYWORD
(       - OPENING BRACKETS
s       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
2       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
==      - OPERATOR
'l'     - Character CONSTANT
)       - CLOSING BRACKETS
{       - OPENING BRACES
printf  - IDENTIFIER
(       - OPENING BRACKETS
"Welcome!!"     - STRING CONSTANT
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
else    - KEYWORD
printf  - IDENTIFIER
(       - OPENING BRACKETS
"Bug1\n"        - STRING CONSTANT
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
else    - KEYWORD
printf  - IDENTIFIER
(       - OPENING BRACKETS
"Bug2\n"        - STRING CONSTANT
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
else    - KEYWORD
printf  - IDENTIFIER
(       - OPENING BRACKETS
"Bug3\n"        - STRING CONSTANT
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
int     - KEYWORD
t       - IDENTIFIER
=       - OPERATOR
size    - IDENTIFIER
;       - SEMICOLON DELIMITER
while   - KEYWORD
(       - OPENING BRACKETS
t       - IDENTIFIER
--      - OPERATOR
)       - CLOSING BRACKETS
{       - OPENING BRACES
printf  - IDENTIFIER
(       - OPENING BRACKETS
"hello world\n"         - STRING CONSTANT
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
```



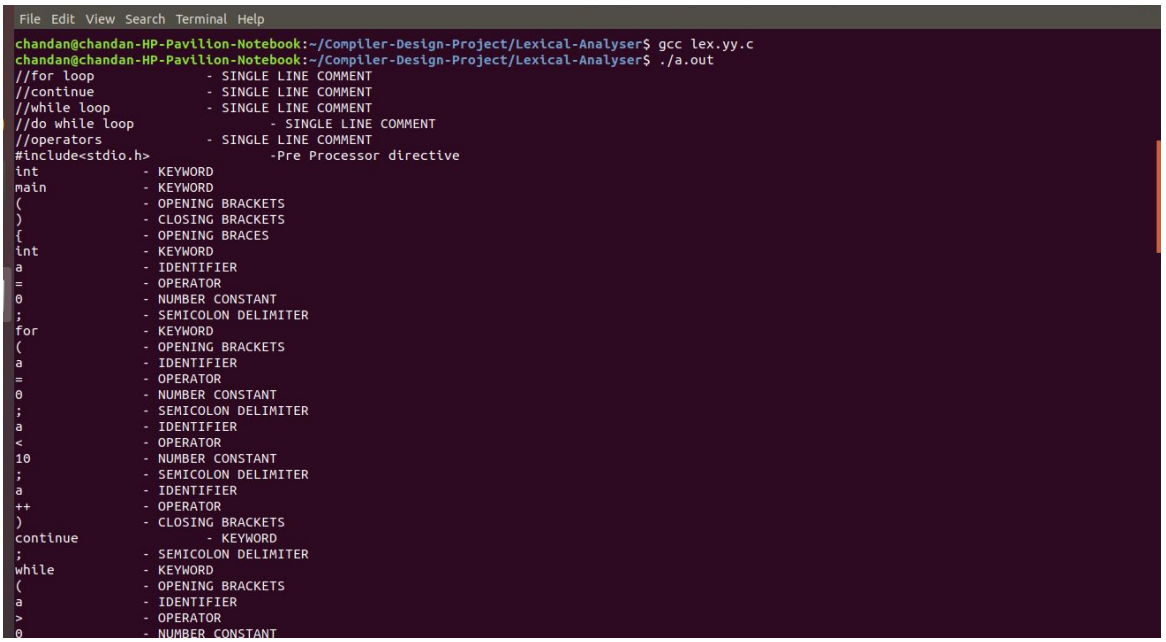
Test 3: Error-free code
(do-while loop, all operators, while loop, for-loop)

```c
#include<stdio.h>

int main()
{
    int a=0;
    for (a = 0; a < 10; a++)
        continue;

    while(a>0) {
        a--;
    }

    do {
        a++;
    }while(a<10);

    int a=2,b,c,d,e,f,g,h;

    c=a+b;
    d=a*b;
    e=a/b;
    f=a%b;
    g=a&&b;
    h=a||b;
    h=a*(a+b);
    h=a*a+b*b;
    h=fun(b);
```

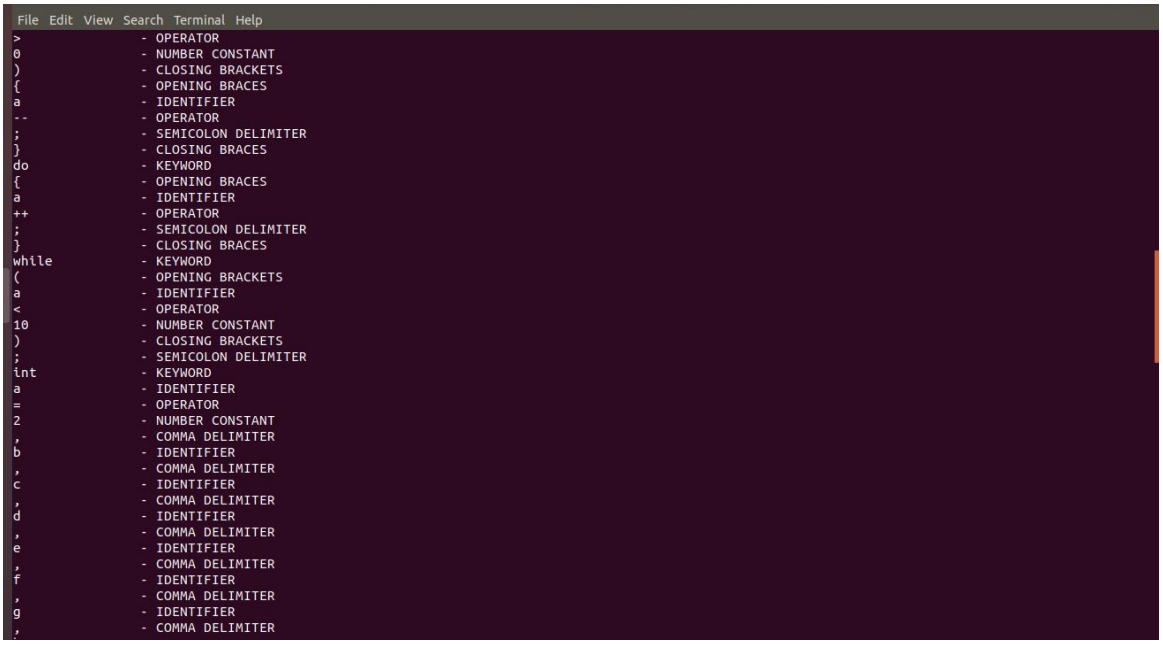
```
}
```

Output:

Status: PASS



```
File  Edit  View  Search  Terminal  Help
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$ gcc lex.yy.c
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$ ./a.out
//for loop              - SINGLE LINE COMMENT
//continue              - SINGLE LINE COMMENT
//while loop            - SINGLE LINE COMMENT
//do while loop                 - SINGLE LINE COMMENT
//operators             - SINGLE LINE COMMENT
#include<stdio.h>               -Pre Processor directive
int             - KEYWORD
main            - KEYWORD
(               - OPENING BRACKETS
)               - CLOSING BRACKETS
{               - OPENING BRACES
int             - KEYWORD
a               - IDENTIFIER
=               - OPERATOR
0               - NUMBER CONSTANT
;               - SEMICOLON DELIMITER
for             - KEYWORD
(               - OPENING BRACKETS
a               - IDENTIFIER
=               - OPERATOR
0               - NUMBER CONSTANT
;               - SEMICOLON DELIMITER
a               - IDENTIFIER
<               - OPERATOR
10              - NUMBER CONSTANT
;               - SEMICOLON DELIMITER
a               - IDENTIFIER
++              - OPERATOR
)               - CLOSING BRACKETS
continue                - KEYWORD
;               - SEMICOLON DELIMITER
while           - KEYWORD
(               - OPENING BRACKETS
a               - IDENTIFIER
>               - OPERATOR
0               - NUMBER CONSTANT
```



```
File  Edit  View  Search  Terminal  Help
>               - OPERATOR
0               - NUMBER CONSTANT
)               - CLOSING BRACKETS
{               - OPENING BRACES
a               - IDENTIFIER
--              - OPERATOR
;               - SEMICOLON DELIMITER
}               - CLOSING BRACES
do              - KEYWORD
{               - OPENING BRACES
a               - IDENTIFIER
++              - OPERATOR
;               - SEMICOLON DELIMITER
}               - CLOSING BRACES
while           - KEYWORD
(               - OPENING BRACKETS
a               - IDENTIFIER
<               - OPERATOR
10              - NUMBER CONSTANT
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
int             - KEYWORD
a               - IDENTIFIER
=               - OPERATOR
2               - NUMBER CONSTANT
,               - COMMA DELIMITER
b               - IDENTIFIER
,               - COMMA DELIMITER
c               - IDENTIFIER
,               - COMMA DELIMITER
d               - IDENTIFIER
,               - COMMA DELIMITER
e               - IDENTIFIER
,               - COMMA DELIMITER
f               - IDENTIFIER
,               - COMMA DELIMITER
g               - IDENTIFIER
,               - COMMA DELIMITER
```

```
File  Edit  View  Search  Terminal  Help
h               - IDENTIFIER
;               - SEMICOLON DELIMITER
c               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
+               - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
d               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
*               - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
e               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
/               - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
f               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
%               - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
g               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
&&              - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
h               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
||              - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
```

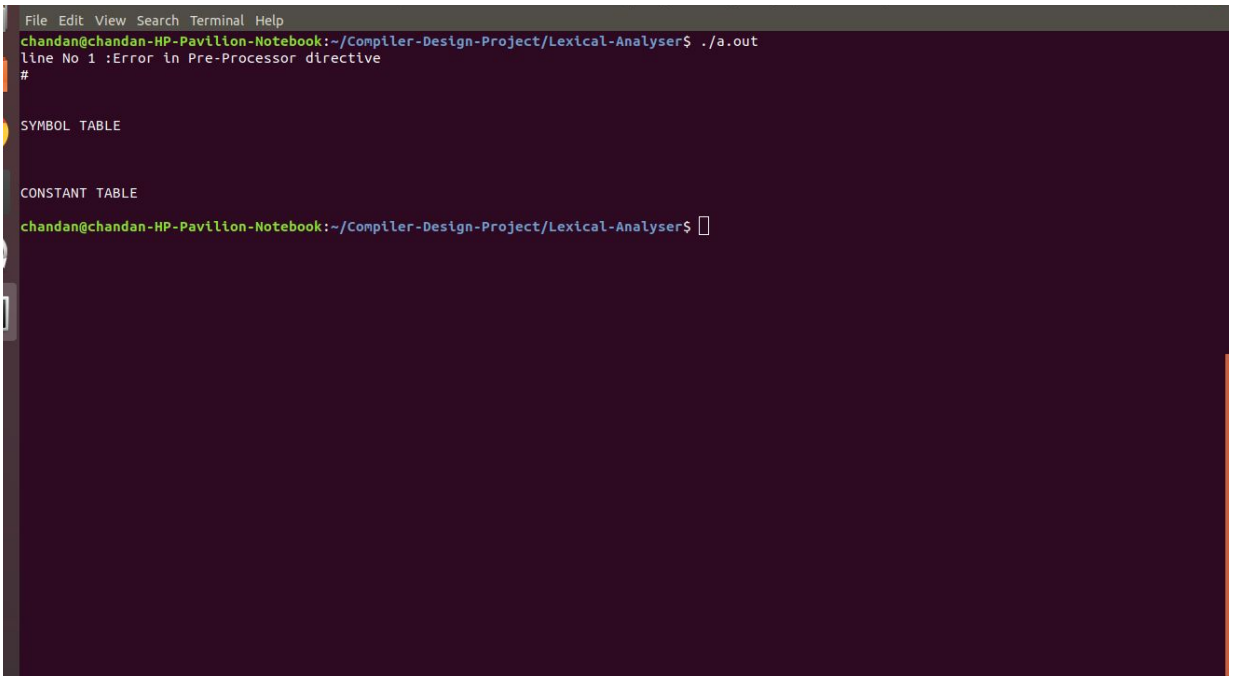
```
File  Edit  View  Search  Terminal  Help
h               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
*               - OPERATOR
(               - OPENING BRACKETS
a               - IDENTIFIER
+               - OPERATOR
b               - IDENTIFIER
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
h               - IDENTIFIER
=               - OPERATOR
a               - IDENTIFIER
*               - OPERATOR
a               - IDENTIFIER
+               - OPERATOR
b               - IDENTIFIER
*               - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
h               - IDENTIFIER
=               - OPERATOR
fun             - IDENTIFIER
(               - OPENING BRACKETS
b               - IDENTIFIER
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
}               - CLOSING BRACES


SYMBOL TABLE

a       IDENTIFIER
b       IDENTIFIER
c       IDENTIFIER
d       IDENTIFIER
e       IDENTIFIER
f       IDENTIFIER
```

```
File  Edit  View  Search  Terminal  Help
*               - OPERATOR
b               - IDENTIFIER
;               - SEMICOLON DELIMITER
h               - IDENTIFIER
=               - OPERATOR
fun             - IDENTIFIER
(               - OPENING BRACKETS
b               - IDENTIFIER
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
}               - CLOSING BRACES


SYMBOL TABLE

a       IDENTIFIER
b       IDENTIFIER
c       IDENTIFIER
d       IDENTIFIER
e       IDENTIFIER
f       IDENTIFIER
g       IDENTIFIER
h       IDENTIFIER
for     KEYWORD
fun     IDENTIFIER
while   KEYWORD
do      KEYWORD
int     KEYWORD
continue        KEYWORD
main    KEYWORD


CONSTANT TABLE

10      NUMBER CONSTANT
0       NUMBER CONSTANT
2       NUMBER CONSTANT
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$
```

Test 4: With Error
(Error in preprocessor directive)

```
#incude<stdio.h>
#define MAX 5
void main()
{
   int a[10];
   a[0]=1;
   printf("%d",a[0]);
}
```

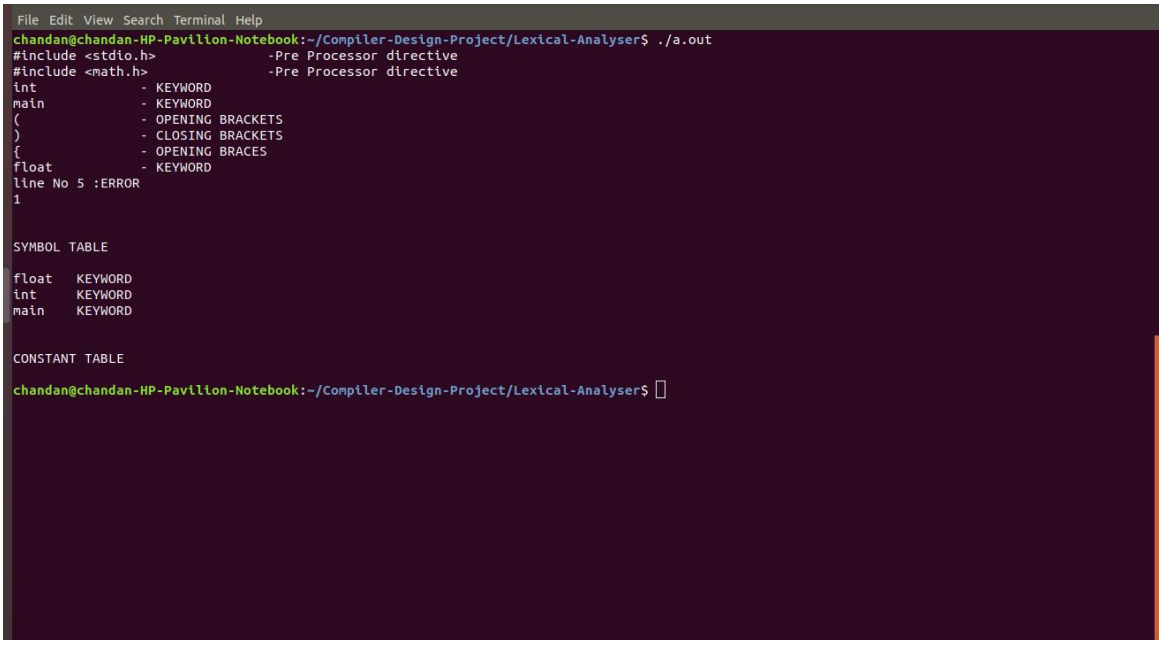
Output:
STATUS: PASS



Test 5: With Error
Error in the variable name

```c
#include <stdio.h>
#include <math.h>
int main()
{
    float 1num, root;
    printf("Enter a number: ");
    scanf("%f", &1num);
    return 0;
}
```

Output:
STATUS: PASS

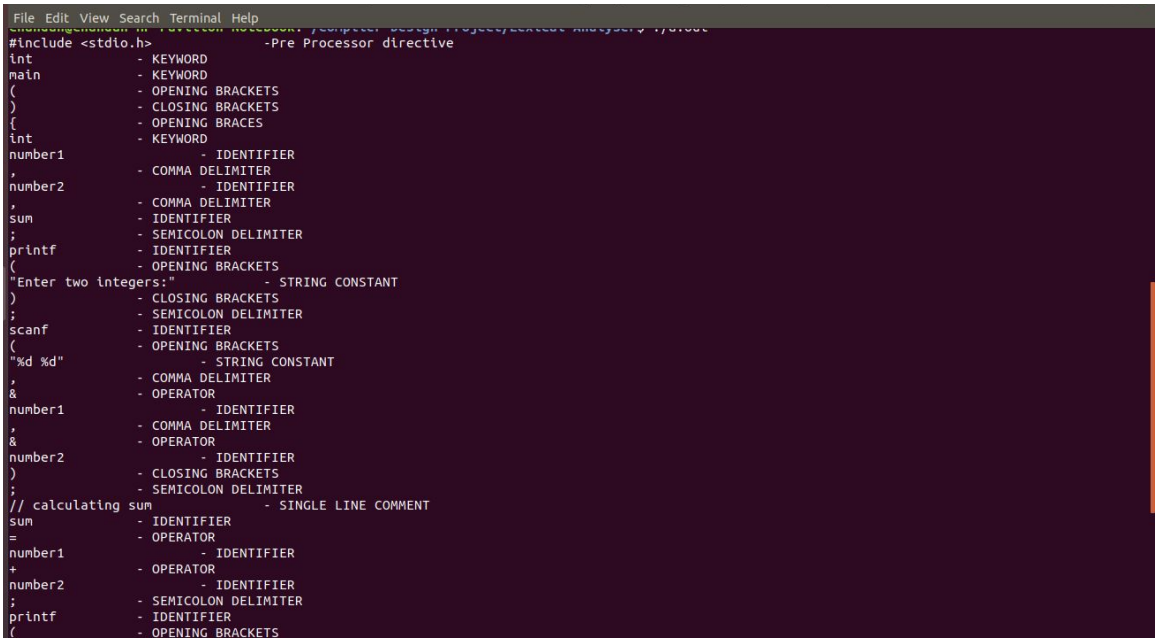


Test 6: With Error
Error in the incomplete string format

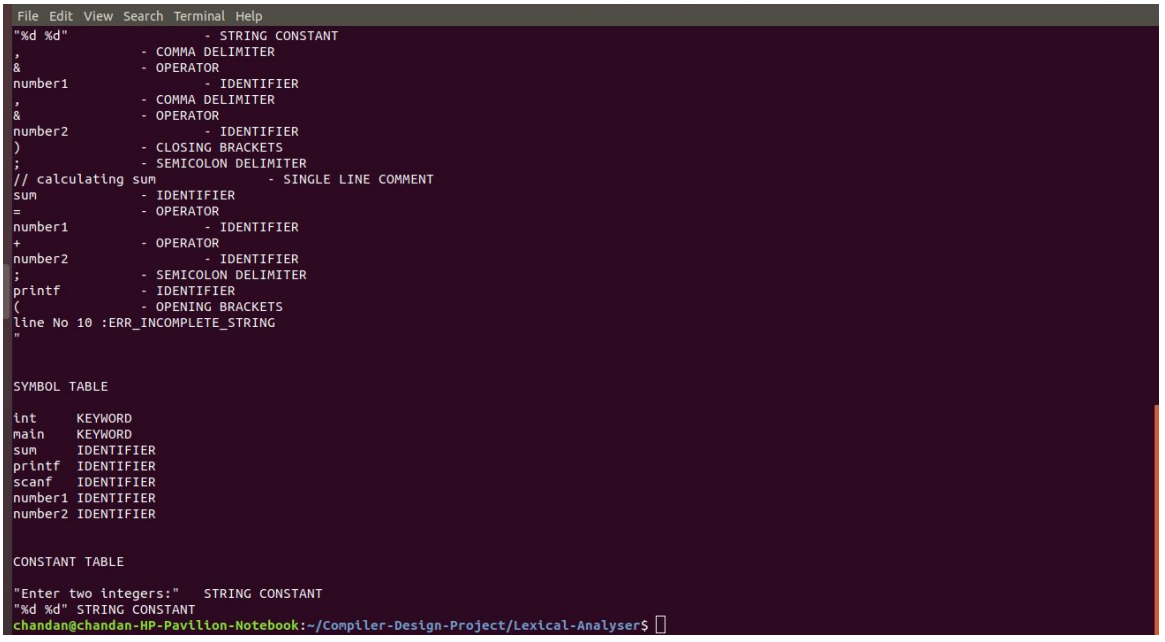
```c
#include <stdio.h>
int main() {
    int number1, number2, sum;

    printf("Enter two integers:");
    scanf("%d %d", &number1, &number2);
    // calculating sum
    sum = number1 + number2;

    printf("%d + %d = %d, number1, number2, sum);
    return 0;
}
```

Output:

STATUS: PASS



```
File  Edit  View  Search  Terminal  Help
#include <stdio.h>                -Pre Processor directive
int             - KEYWORD
main            - KEYWORD
(               - OPENING BRACKETS
)               - CLOSING BRACKETS
{               - OPENING BRACES
int             - KEYWORD
number1                 - IDENTIFIER
,               - COMMA DELIMITER
number2                 - IDENTIFIER
,               - COMMA DELIMITER
sum             - IDENTIFIER
;               - SEMICOLON DELIMITER
printf          - IDENTIFIER
(               - OPENING BRACKETS
"Enter two integers:"           - STRING CONSTANT
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
scanf           - IDENTIFIER
(               - OPENING BRACKETS
"%d %d"                 - STRING CONSTANT
,               - COMMA DELIMITER
&               - OPERATOR
number1                 - IDENTIFIER
,               - COMMA DELIMITER
&               - OPERATOR
number2                 - IDENTIFIER
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
// calculating sum              - SINGLE LINE COMMENT
sum             - IDENTIFIER
=               - OPERATOR
number1                 - IDENTIFIER
+               - OPERATOR
number2                 - IDENTIFIER
;               - SEMICOLON DELIMITER
printf          - IDENTIFIER
(               - OPENING BRACKETS
```

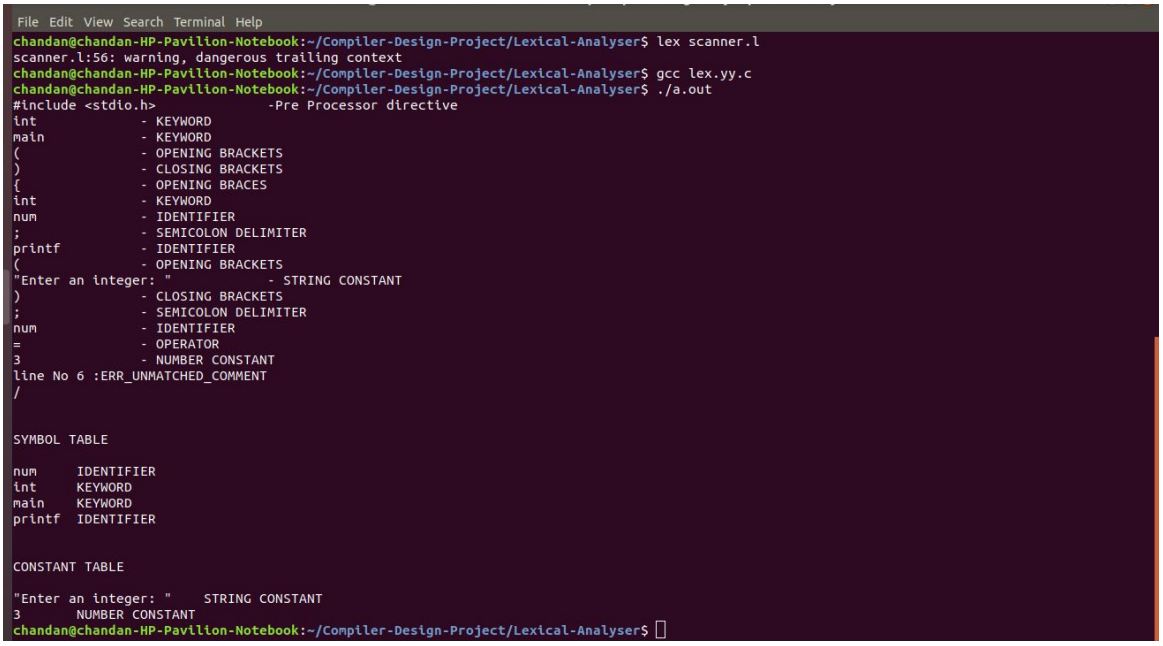


```
File  Edit  View  Search  Terminal  Help
"%d %d"                 - STRING CONSTANT
,               - COMMA DELIMITER
&               - OPERATOR
number1                 - IDENTIFIER
,               - COMMA DELIMITER
&               - OPERATOR
number2                 - IDENTIFIER
)               - CLOSING BRACKETS
;               - SEMICOLON DELIMITER
// calculating sum              - SINGLE LINE COMMENT
sum             - IDENTIFIER
=               - OPERATOR
number1                 - IDENTIFIER
+               - OPERATOR
number2                 - IDENTIFIER
;               - SEMICOLON DELIMITER
printf          - IDENTIFIER
(               - OPENING BRACKETS
line No 10 :ERR_INCOMPLETE_STRING
"


SYMBOL TABLE

int     KEYWORD
main    KEYWORD
sum     IDENTIFIER
printf  IDENTIFIER
scanf   IDENTIFIER
number1 IDENTIFIER
number2 IDENTIFIER


CONSTANT TABLE

"Enter two integers:"   STRING CONSTANT
"%d %d" STRING CONSTANT
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$
```

Test 7: With Error
Error in the incomplete multi-line comment:

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    num=3
    /* even if num is perfectly divisible by 2
     else it is odd
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```

Output:
STATUS: PASS


File Edit View Search Terminal Help
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$ lex scanner.l
scanner.l:56: warning, dangerous trailing context
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$ gcc lex.yy.c
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$ ./a.out
#include <stdio.h> -Pre Processor directive
int - KEYWORD
main - KEYWORD
(- OPENING BRACKETS
) - CLOSING BRACKETS
{ - OPENING BRACES
int - KEYWORD
num - IDENTIFIER
; - SEMICOLON DELIMITER
printf - IDENTIFIER
(- OPENING BRACKETS
"Enter an integer: " - STRING CONSTANT
) - CLOSING BRACKETS
; - SEMICOLON DELIMITER
num - IDENTIFIER
= - OPERATOR
3 - NUMBER CONSTANT
line No 6 :ERR_UNMATCHED_COMMENT
/
SYMBOL TABLE
num IDENTIFIER
int KEYWORD
main KEYWORD
printf IDENTIFIER
CONSTANT TABLE
"Enter an integer: " STRING CONSTANT
3 NUMBER CONSTANT
chandan@chandan-HP-Pavilion-Notebook:~/Compiler-Design-Project/Lexical-Analyser$

Implementation

The Regular Expressions for most of the features of C are fairly straightforward. However, a few features require a significant amount of thought, such as:

- **The Regex for Identifiers:** The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.

- **Multiline comments should be supported:** This has been supported by using custom regular algorithm especially robust in cases where tricky characters like * or / are used within the comments.

- **Literals:** Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, strings, etc.

- **Error Handling for Incomplete String:** Open and close quote missing, both kinds of errors have been handled in the rules written in the script.

- **Error Handling for Nested Comments:** This use-case has been handled by the custom-defined regular expressions that help throw errors when comment opening or closing is missing.

At the end of the token recognition, the lexer prints a list of all the identifiers and constants present in the program. We use the following technique to implement this:

● We maintain two structures one for symbol table and other for constant table corresponding to identifiers and other to the constants.
● Four functions have been implemented `searchSymTbl()`, `searchConstTbl()`, these functions return true if the identifier and constant respectively are already present in the table. `insertSymTbl()`, `insertConstTbl()` help to insert identifier/constant in the appropriate table.
● Whenever we encounter an identifier/constant, we call the `insertConstTbl()` or `insertConstTbl()` function which in turns call `searchSymTbl()` or `searchConstTbl()` and adds it to the corresponding structure.
● In the end, in `main()` function, after yylex returns, we call `printSymTbl()` and `printConstTbl()`, which in turn prints the list of identifier and constants in a proper format.

Results and Future Work:

Result:

The output of the program contains

- ➔ Tokens with their respective token classes,
- ➔ Symbol table which contains tokens and attributes,
- ➔ Constant table which contains tokens and attributes.
- ➔ It also contains possible errors if present along with their error messages.

Future work:

The flex script presented in this report takes care of all the rules of C language but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

References:

- https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/