Lexical Analyzer for the C Language

National Institute of Technology Karnataka, Surathkal

Date: 22 August 2020

Submitted To:

Prof. P. Santhi Thilagam

CSE Dept, NITK

Group Members:

Bhaskar Kataria, 181CO213

Ketan Bhujange, 181CO227

Omanshu Mahawar, 181CO237

Shrvan Warke, 181CO151

Abstract

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Compiler operates in 4 phases, namely **Lexical Analyzer, Parser**, **Semantic Analysis** and **Intermediate Code generator.** This module focuses on Lexical Analyzer.

LEXICAL ANALYSIS is the very first phase in the compiler designing. Lexical analyzer reads the characters from source code and converts it into tokens.

Analysis of the following class of tokens and statements are made by the Lexical Analyzer:

Keywords
`auto, const, default, enum, extern, register, return, sizeof, static, struct, typedef, union, volatile, break, continue, goto, else, switch, if, case, for, do, while, char, double, float, int, long, short, signed, unsigned, void`

Comments
Single line and multiline comments,

Identifiers
Identification of valid identifiers used in the language,

It supports nested for and while loops, nested if...else-if...else statements, and nested conditional statement,

Operators
ADD(+), MULTIPLY(*), DIVIDE(/), MODULO(%), AND(&), OR(|)

Function construct of the language, Syntax: int func(int x)

Contents

- Introduction
  - Lexical Analyzer
  - Flex Script
  - C Program
- Design of Programs
  - Code
  - Explanation
- Test Cases
  - Without Errors
  - With Errors
- Implementation
- Results / Future work
- References

List of Figures and Tables

1. Table 1 : Test Cases without errors
2. Table 2 : Test Cases with errors
3. Figure 1 : Input for identifying comments
4. Figure 2: Output of Identifying comments
5. Figure 3: Input for identifying Nested Loop
6. Figure 4: Output for identifying Nested Loop
7. Figure 5: Input for missing parenthesis
8. Figure 6: Output for missing parenthesis
9. Figure 7 : Input for incomplete string
10. Figure 8: Output for incomplete string
11. Figure 9: Input for error in preprocessor directive
12. Figure 10: Output for error in preprocessor directive

Introduction

Lexical Analyzer

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**. The main task of the lexical analyzer is to read the input characters of the source program, group them into **lexemes**, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis.

Tokens: A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
Example

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. E.g., “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”.

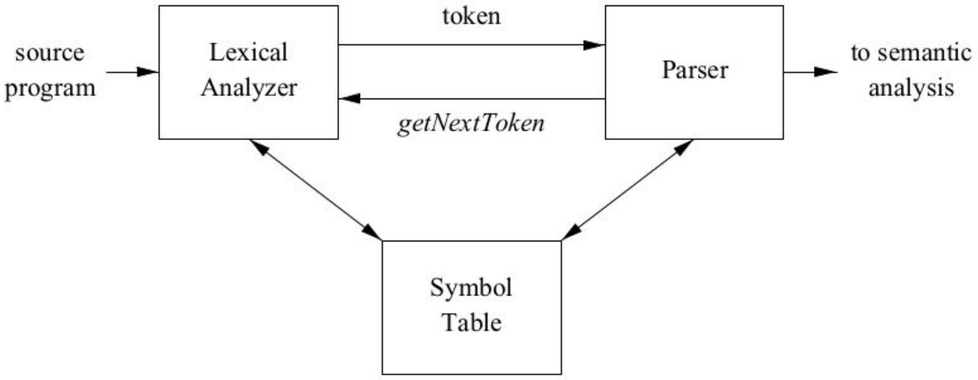


Figure 3.1: Interactions between the lexical analyzer and the parser

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Flex Script

It is a tool/computer program for generating 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.

Program Structure:
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:

```
%{
        // Definitions
%}

%%
        // 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.

Symbol Table and Constant Table

The symbol table is an important data structure created and maintained by the compiler to keep track of semantics of variables, i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

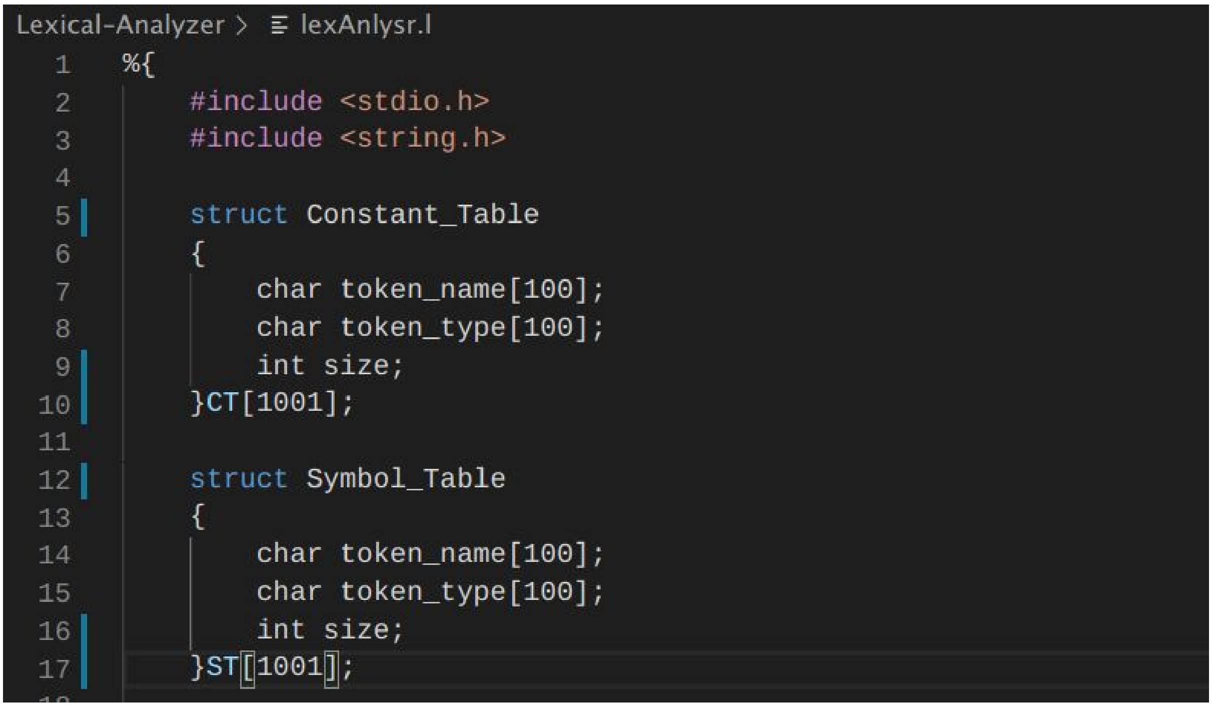
C Program

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 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. The script also has an option to take standard input instead of taking input from a file.

Design of C Programs

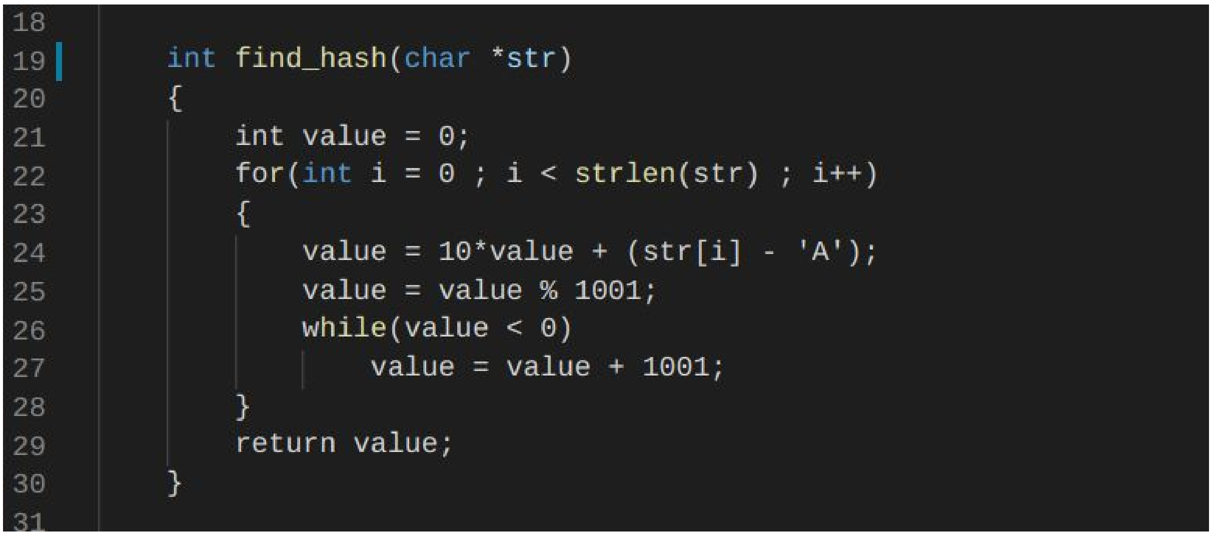
Code

Symbol Table and Constant Table



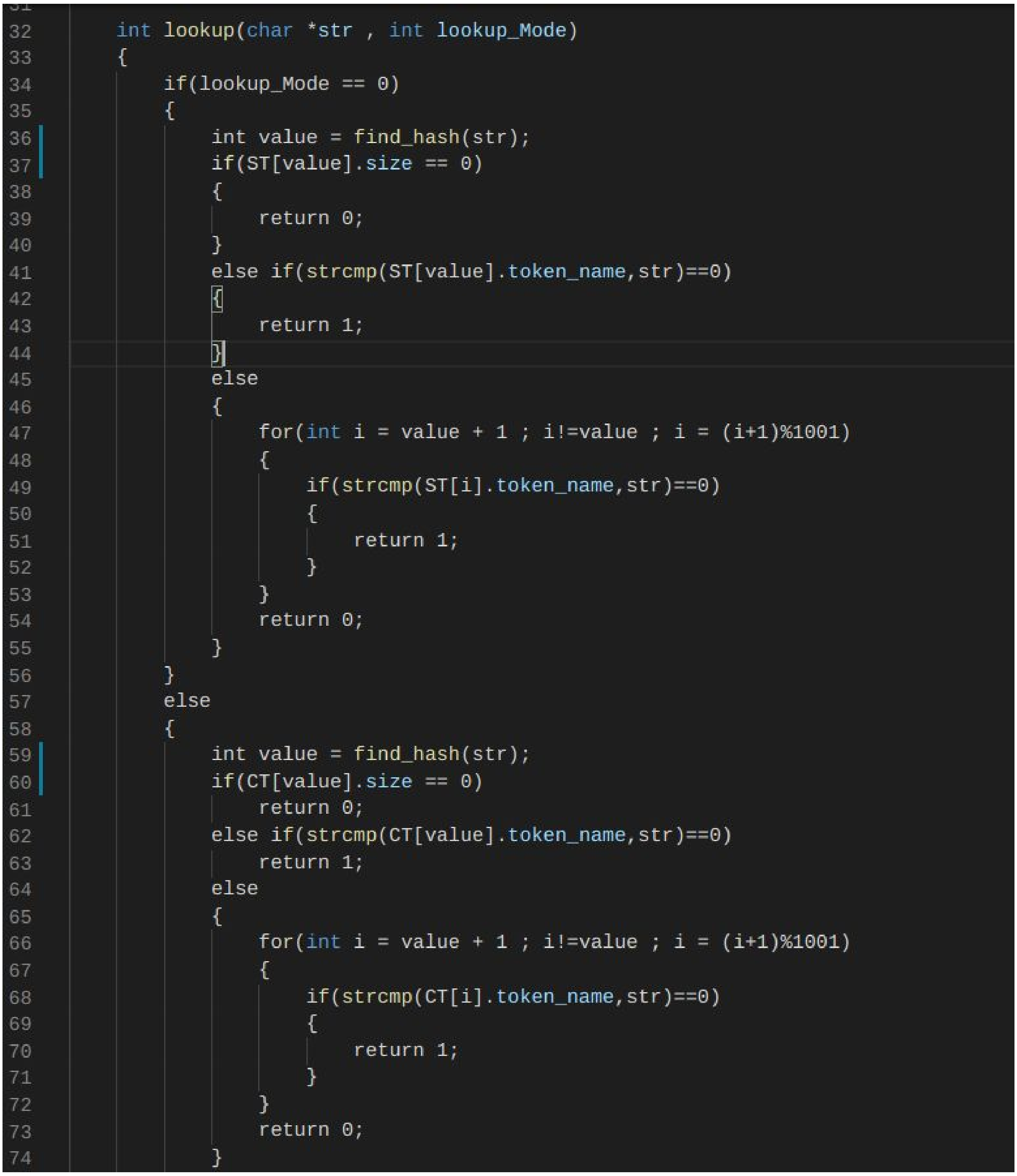
```
%{
    #include <stdio.h>
    #include <string.h>

    struct Constant_Table
    {
        char token_name[100];
        char token_type[100];
        int size;
    }CT[1001];

    struct Symbol_Table
    {
        char token_name[100];
        char token_type[100];
        int size;
    }ST[1001];
```

Hash Function



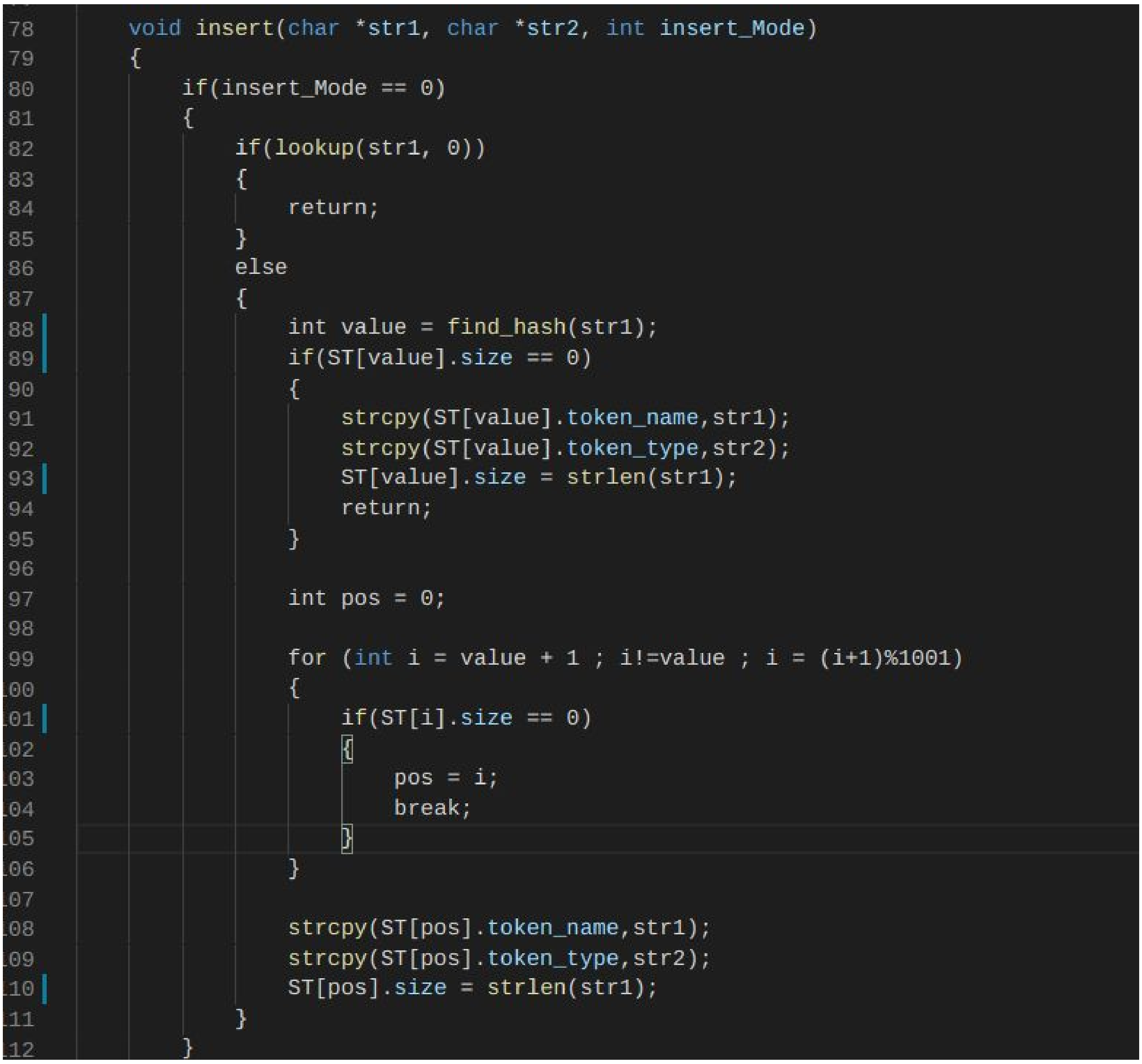
```

    int find_hash(char *str)
    {
        int value = 0;
        for(int i = 0 ; i < strlen(str) ; i++)
        {
            value = 10*value + (str[i] - 'A');
            value = value % 1001;
            while(value < 0)
                value = value + 1001;
        }
        return value;
    }
```

Look Up Function

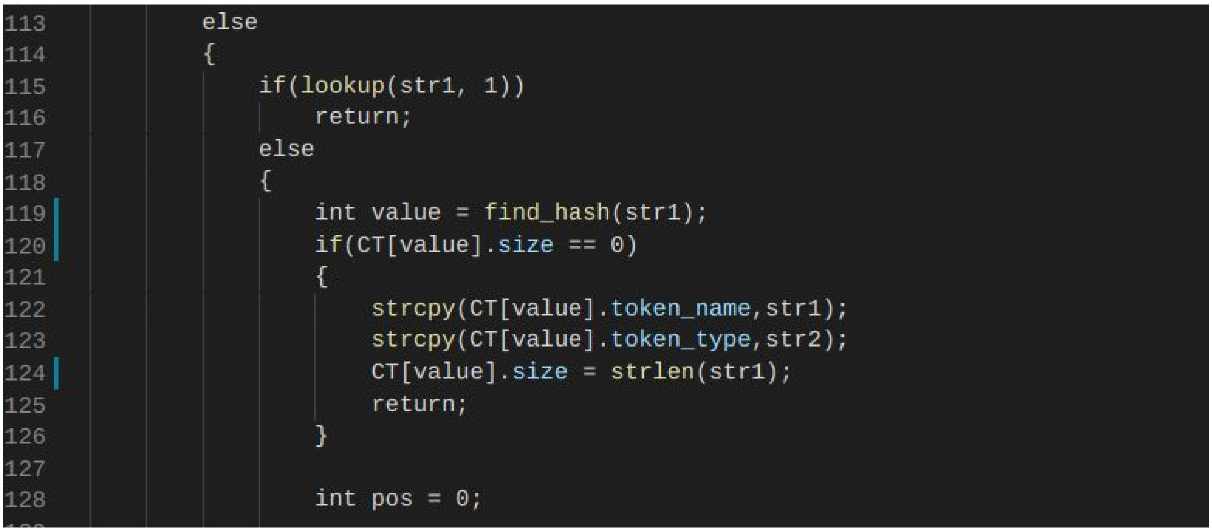


```
    int lookup(char *str , int lookup_Mode)
    {
        if(lookup_Mode == 0)
        {
            int value = find_hash(str);
            if(ST[value].size == 0)
            {
                return 0;
            }
            else if(strcmp(ST[value].token_name,str)==0)
            {
                return 1;
            }
            else
            {
                for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
                {
                    if(strcmp(ST[i].token_name,str)==0)
                    {
                        return 1;
                    }
                }
                return 0;
            }
        }
        else
        {
            int value = find_hash(str);
            if(CT[value].size == 0)
                return 0;
            else if(strcmp(CT[value].token_name,str)==0)
                return 1;
            else
            {
                for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
                {
                    if(strcmp(CT[i].token_name,str)==0)
                    {
                        return 1;
                    }
                }
                return 0;
            }
```

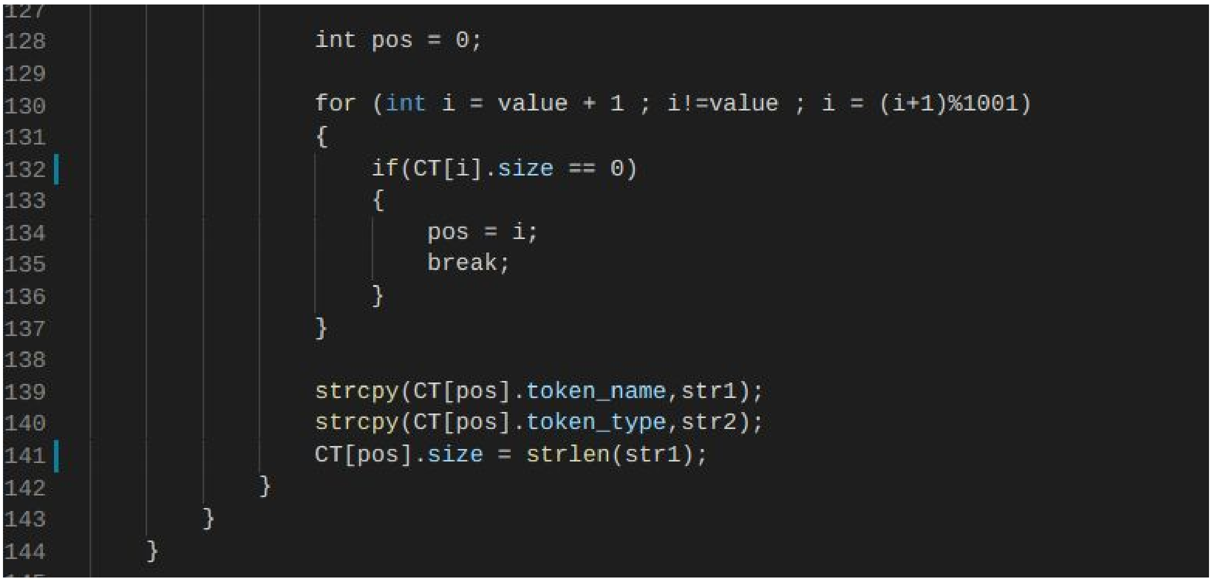
Insert Function



```
void insert(char *str1, char *str2, int insert_Mode)
{
    if(insert_Mode == 0)
    {
        if(lookup(str1, 0))
        {
            return;
        }
        else
        {
            int value = find_hash(str1);
            if(ST[value].size == 0)
            {
                strcpy(ST[value].token_name,str1);
                strcpy(ST[value].token_type,str2);
                ST[value].size = strlen(str1);
                return;
            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
            {
                if(ST[i].size == 0)
                {
                    pos = i;
                    break;
                }
            }

            strcpy(ST[pos].token_name,str1);
            strcpy(ST[pos].token_type,str2);
            ST[pos].size = strlen(str1);
        }
    }
```

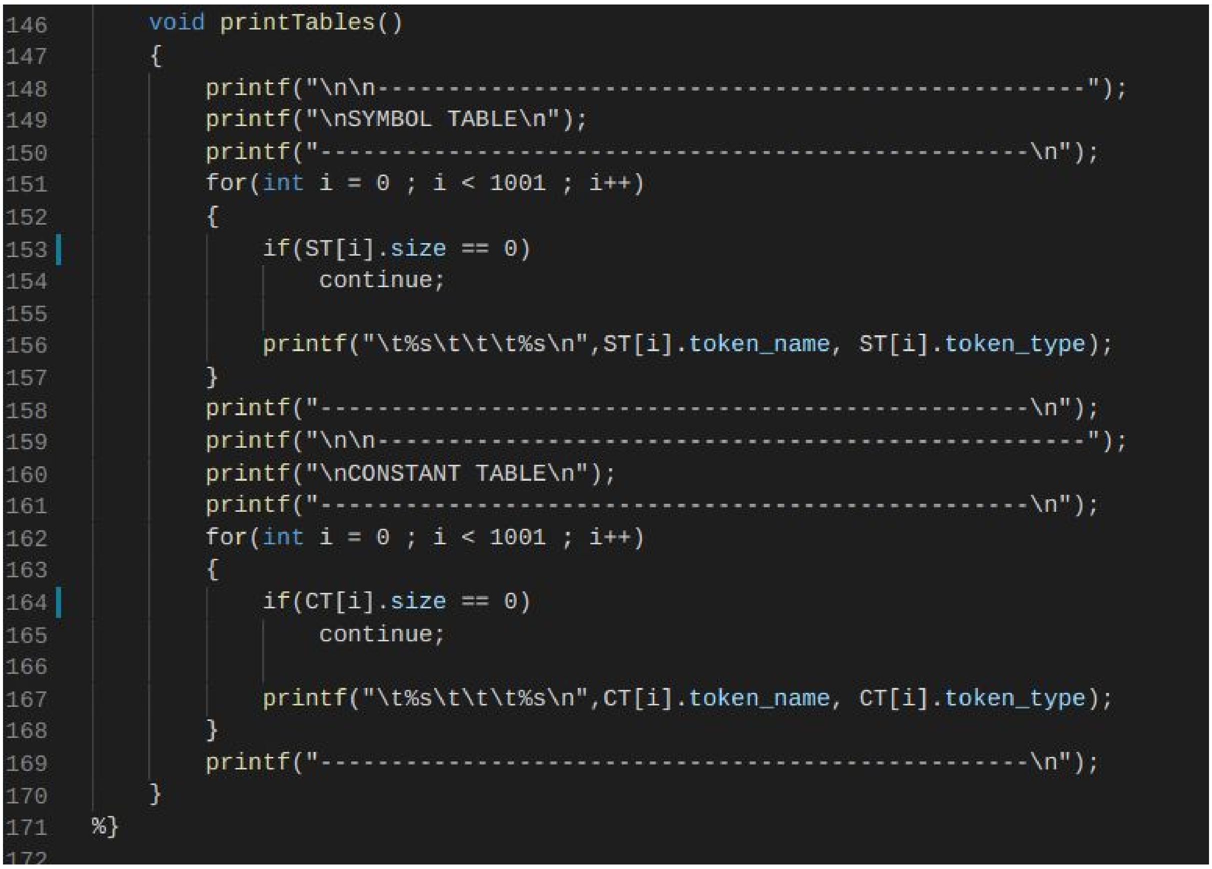


```
    else
    {
        if(lookup(str1, 1))
            return;
        else
        {
            int value = find_hash(str1);
            if(CT[value].size == 0)
            {
                strcpy(CT[value].token_name,str1);
                strcpy(CT[value].token_type,str2);
                CT[value].size = strlen(str1);
                return;
            }

            int pos = 0;
```



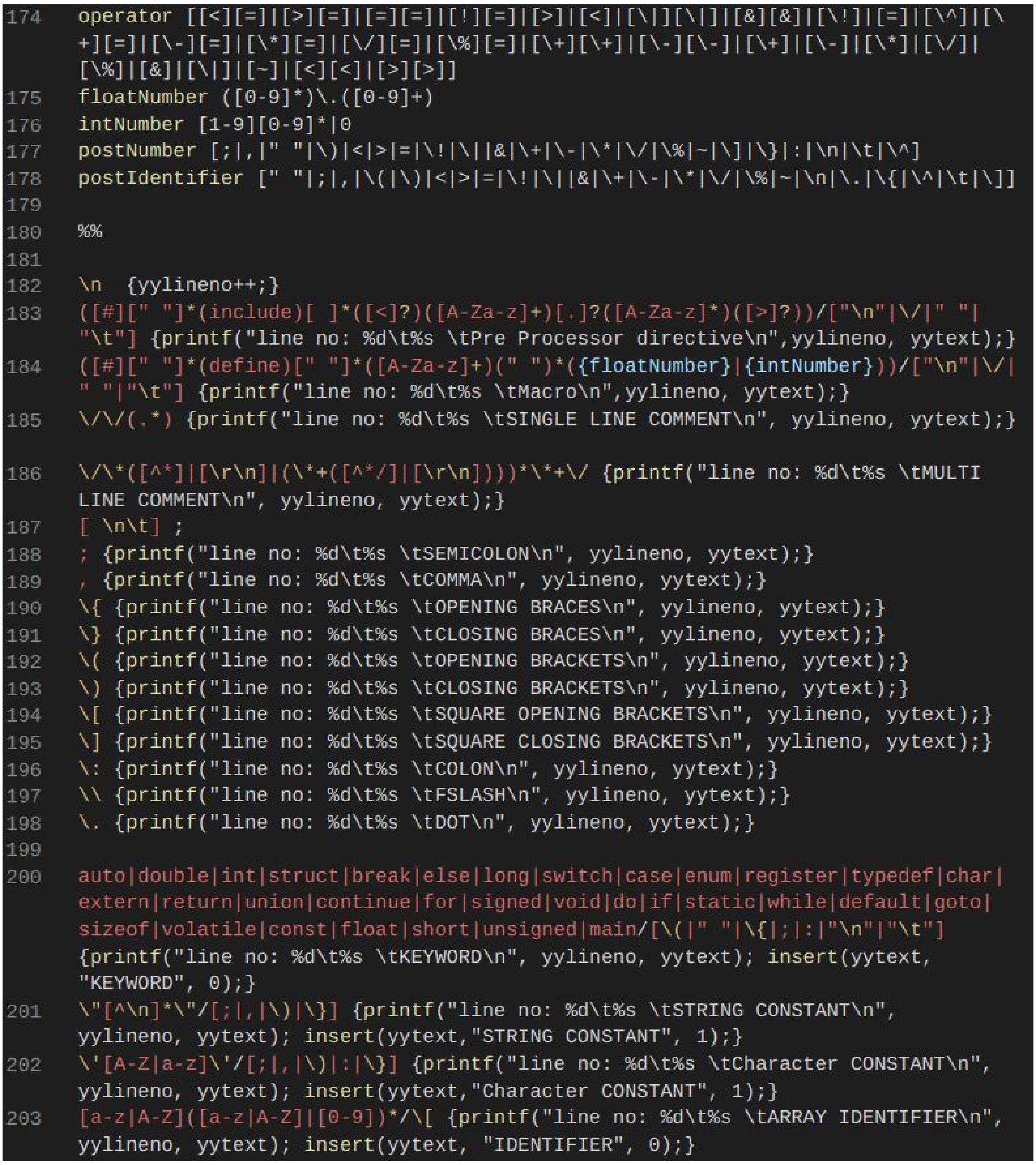
```
                int pos = 0;

                for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
                {
                    if(CT[i].size == 0)
                    {
                        pos = i;
                        break;
                    }
                }

                strcpy(CT[pos].token_name,str1);
                strcpy(CT[pos].token_type,str2);
                CT[pos].size = strlen(str1);
            }
        }
    }
```

Printing Tables

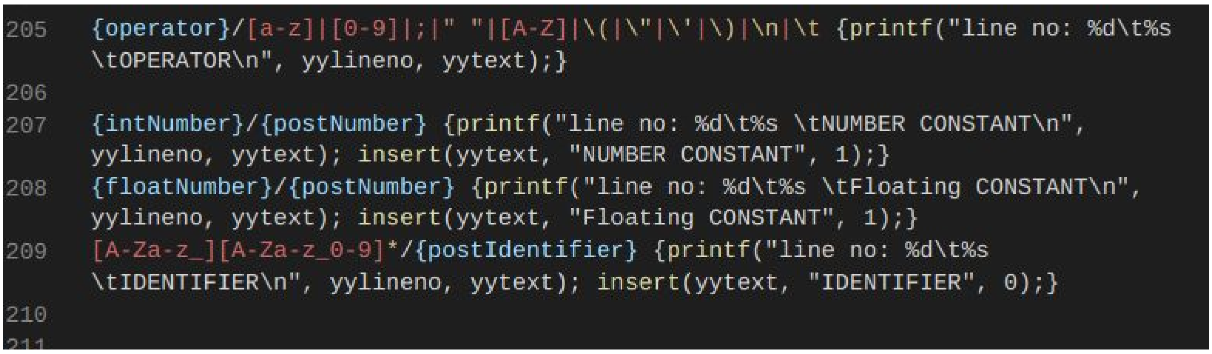


```
    void printTables()
    {
        printf("\n\n---------------------------------------------------");
        printf("\nSYMBOL TABLE\n");
        printf("---------------------------------------------------\n");
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(ST[i].size == 0)
                continue;

            printf("\t%s\t\t\t%s\n",ST[i].token_name, ST[i].token_type);
        }
        printf("---------------------------------------------------\n");
        printf("\n\n---------------------------------------------------");
        printf("\nCONSTANT TABLE\n");
        printf("---------------------------------------------------\n");
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(CT[i].size == 0)
                continue;

            printf("\t%s\t\t\t%s\n",CT[i].token_name, CT[i].token_type);
        }
        printf("---------------------------------------------------\n");
    }
%}
```

Rules

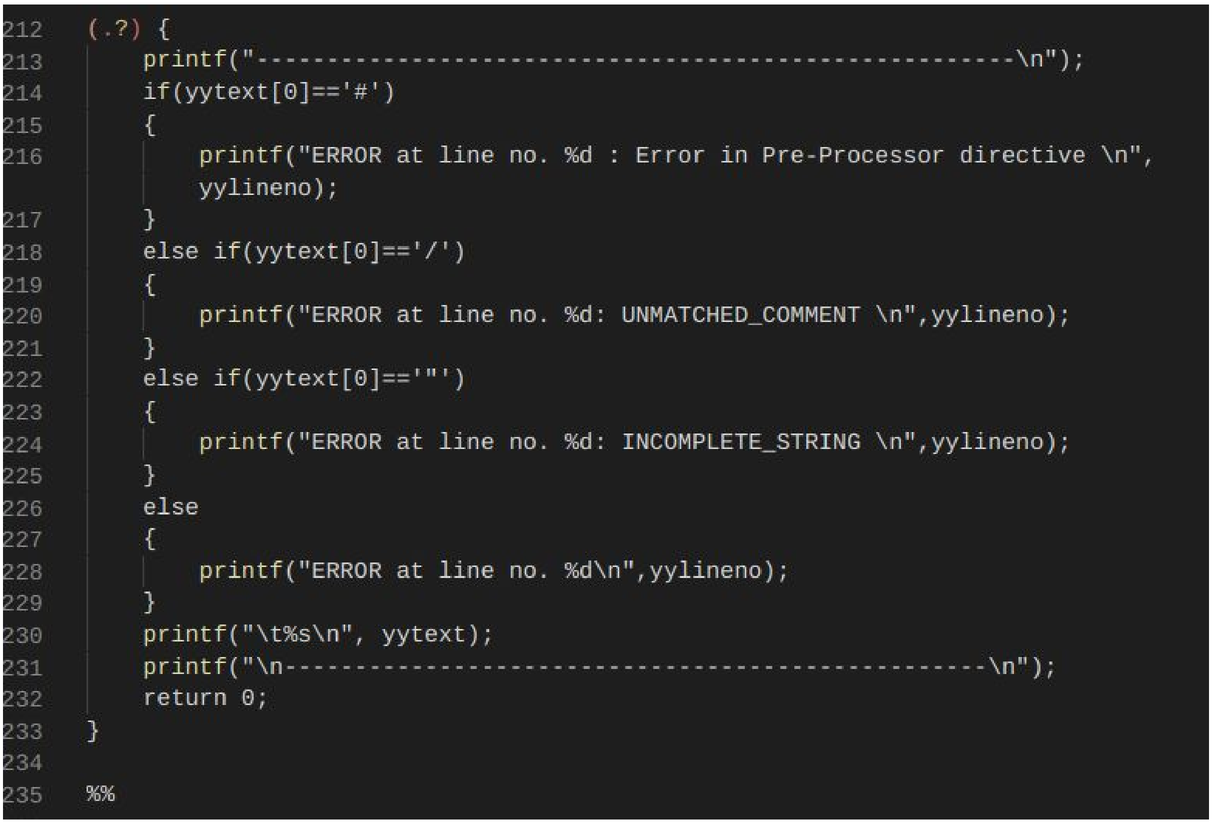


```
operator [[<][=]|[>][=]|[=][=]|[!][=]|[>]|[<]|[\|][\|]|[&][&]|[\!]|[=]|[\^]|[\+][=]|[\-][=]|[\*][=]|[\/][=]|[\%][=]|[\+][\+]|[\-][\-]|[\+]|[\-]|[\*]|[\/]|[\%]|[&]|[\|]|[~]|[<][<]|[>][>]]
floatNumber ([0-9]*)\.([0-9]+)
intNumber [1-9][0-9]*|0
postNumber [;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
postIdentifier [" "|;|,|\(|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\.|\{|\^|\t|\]]

%%

\n  {yylineno++;}
([#][" "]*(include)[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] {printf("line no: %d\t%s \tPre Processor directive\n",yylineno, yytext);}
([#][" "]*(define)[" "]*([A-Za-z]+)(" ")*({floatNumber}|{intNumber}))/["\n"|\/|" "|"\t"] {printf("line no: %d\t%s \tMacro\n",yylineno, yytext);}
\/\/(.*) {printf("line no: %d\t%s \tSINGLE LINE COMMENT\n", yylineno, yytext);}

\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/ {printf("line no: %d\t%s \tMULTI LINE COMMENT\n", yylineno, yytext);}
[ \n\t] ;
; {printf("line no: %d\t%s \tSEMICOLON\n", yylineno, yytext);}
, {printf("line no: %d\t%s \tCOMMA\n", yylineno, yytext);}
\{ {printf("line no: %d\t%s \tOPENING BRACES\n", yylineno, yytext);}
\} {printf("line no: %d\t%s \tCLOSING BRACES\n", yylineno, yytext);}
\( {printf("line no: %d\t%s \tOPENING BRACKETS\n", yylineno, yytext);}
\) {printf("line no: %d\t%s \tCLOSING BRACKETS\n", yylineno, yytext);}
\[ {printf("line no: %d\t%s \tSQUARE OPENING BRACKETS\n", yylineno, yytext);}
\] {printf("line no: %d\t%s \tSQUARE CLOSING BRACKETS\n", yylineno, yytext);}
\: {printf("line no: %d\t%s \tCOLON\n", yylineno, yytext);}
\\ {printf("line no: %d\t%s \tFSLASH\n", yylineno, yytext);}
\. {printf("line no: %d\t%s \tDOT\n", yylineno, yytext);}

auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern|return|union|continue|for|signed|void|do|if|static|while|default|goto|sizeof|volatile|const|float|short|unsigned|main/[\(|" "|\{|;|:|"\n"|"\t"] {printf("line no: %d\t%s \tKEYWORD\n", yylineno, yytext); insert(yytext, "KEYWORD", 0);}
\"[^\n]*\"/[;|,|\)|\}] {printf("line no: %d\t%s \tSTRING CONSTANT\n", yylineno, yytext); insert(yytext,"STRING CONSTANT", 1);}
\'[A-Z|a-z]\'/[;|,|\)|:|\}] {printf("line no: %d\t%s \tCharacter CONSTANT\n", yylineno, yytext); insert(yytext,"Character CONSTANT", 1);}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[ {printf("line no: %d\t%s \tARRAY IDENTIFIER\n", yylineno, yytext); insert(yytext, "IDENTIFIER", 0);}
```



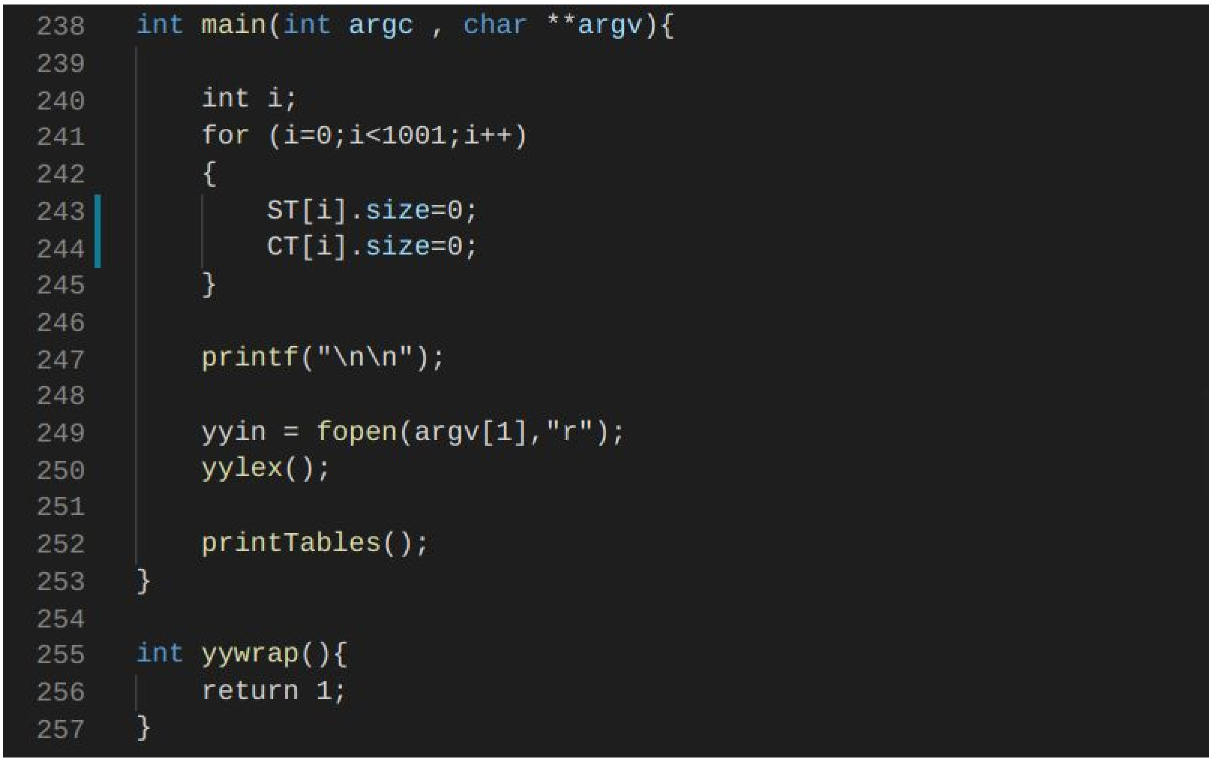
```
{operator}/[a-z]|[0-9]|;|" "|[A-Z]|\(|\"|\'|\)|\n|\t {printf("line no: %d\t%s \tOPERATOR\n", yylineno, yytext);}

{intNumber}/{postNumber} {printf("line no: %d\t%s \tNUMBER CONSTANT\n", yylineno, yytext); insert(yytext, "NUMBER CONSTANT", 1);}
{floatNumber}/{postNumber} {printf("line no: %d\t%s \tFloating CONSTANT\n", yylineno, yytext); insert(yytext, "Floating CONSTANT", 1);}
[A-Za-z_][A-Za-z_0-9]*/{postIdentifier} {printf("line no: %d\t%s \tIDENTIFIER\n", yylineno, yytext); insert(yytext, "IDENTIFIER", 0);}

```

Identifying Lexical Errors



```
(.?) {
    printf("------------------------------------------------------\n");
    if(yytext[0]=='#')
    {
        printf("ERROR at line no. %d : Error in Pre-Processor directive \n", yylineno);
    }
    else if(yytext[0]=='/')
    {
        printf("ERROR at line no. %d: UNMATCHED_COMMENT \n",yylineno);
    }
    else if(yytext[0]=='"')
    {
        printf("ERROR at line no. %d: INCOMPLETE_STRING \n",yylineno);
    }
    else
    {
        printf("ERROR at line no. %d\n",yylineno);
    }
    printf("\t%s\n", yytext);
    printf("\n------------------------------------------------------\n");
    return 0;
}

%%
```

Main Function



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

    int i;
    for (i=0;i<1001;i++)
    {
        ST[i].size=0;
        CT[i].size=0;
    }

    printf("\n\n");

    yyin = fopen(argv[1],"r");
    yylex();

    printTables();
}

int yywrap(){
    return 1;
}
```

Explanation

Definition Section

- All header files are included in the definition section.
- It also contains the structure of the `Symbol table` and the `Constant table` along with the various functions supporting both the tables lookup, insert, and hash function.
- The `find_hash` function takes a string input and returns an integer hash of that string.
- The `lookup` function checks if entry is already present in the table or not.
- The `insert` function inserts an entry into the corresponding table, linear probing hashing technique is used to handle collisions.
- The `print_table` function is used to neatly print both the tables.

Rules Section

- The rules section includes all the necessary rules in the form of regular expressions.
- Regular expressions are included to identify the following:
    - Preprocessor Directive
    - Keywords
    - Identifiers
    - Operators
    - Single and Multiline comments
    - Parenthesis
    - ERRORS in
        - Preprocessor directives
        - Unmatched comments
        - Incomplete string

C Section

- In the C section, both the symbol and constant table are initialized and the `print_table` function is called to show the results
- yylex() function is called to run the program on the given input file.

Test Cases

Without Errors

Table 1

Serial No.	**Test Case**	**Expected Output**	**Result**
1	#include <stdio.h>	Preprocessor directive	PASS
2	/* Comment */	Comment	PASS
3	int n = 10;	int: Keyword n Identifier = Operator 10 Number Constant	PASS
4	while()	While Keyword (Opening Bracket) Closing Bracket	PASS

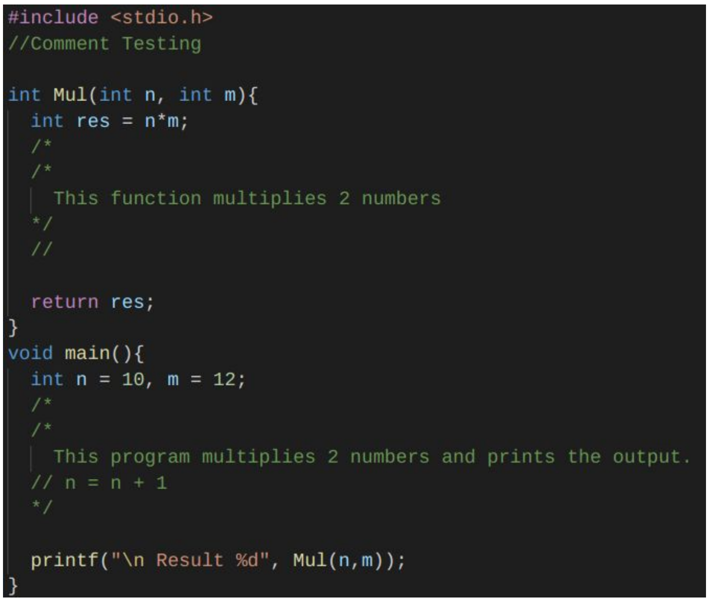
With Errors

Table 2

Serial No.	**Test Case**	**Expected Output**	**Result**
1	if (	If Keyword (Opening Bracket Error Missing Bracket	PASS
2	/* Comment	Error Unmatched Comment	PASS
3	string a = “ac	string: Keyword a Identifier = Operator Error Unmatched Comment	PASS
4	# inc <stdio.>	Error in preprocessor directive	PASS

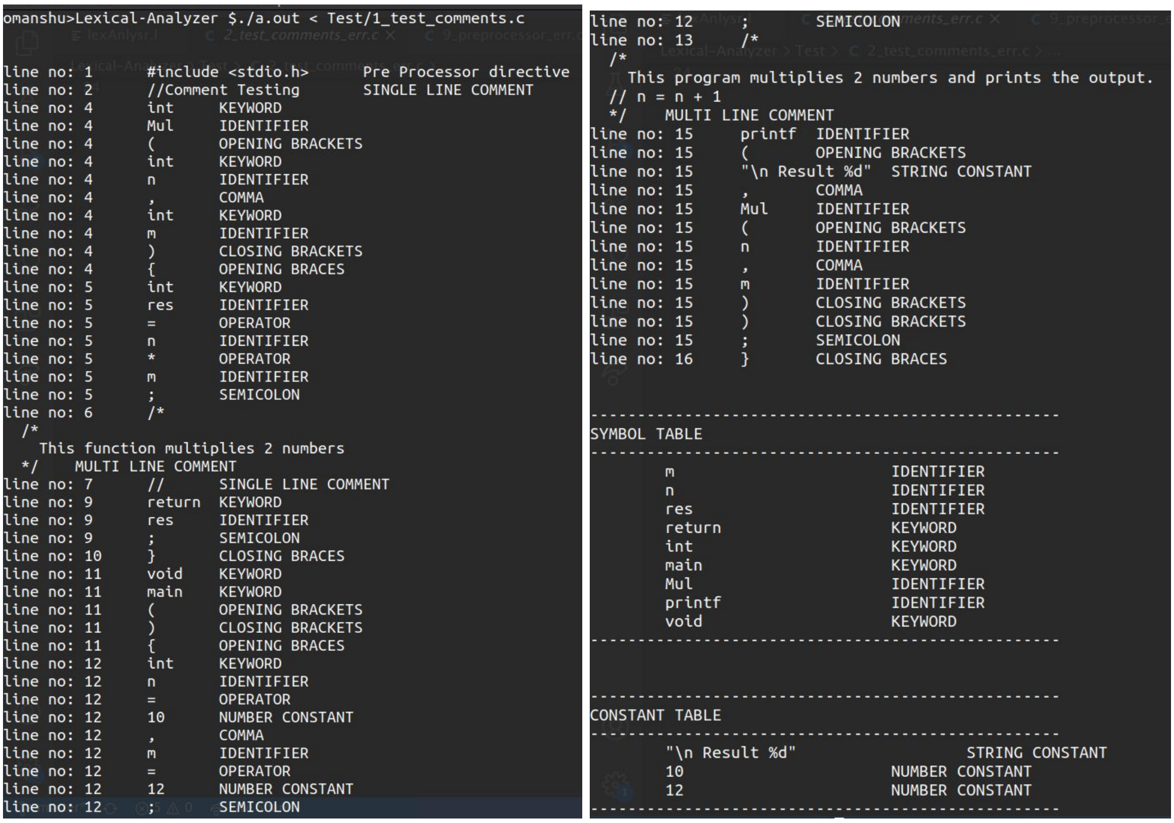
Identifying Comments

Figure 1



```
#include <stdio.h>
//Comment Testing

int Mul(int n, int m){
  int res = n*m;
  /*
  /*
    This function multiplies 2 numbers
  */
  //

  return res;
}
void main(){
  int n = 10, m = 12;
  /*
  /*
    This program multiplies 2 numbers and prints the output.
  // n = n + 1
  */

  printf("\n Result %d", Mul(n,m));
}
```

Figure 2

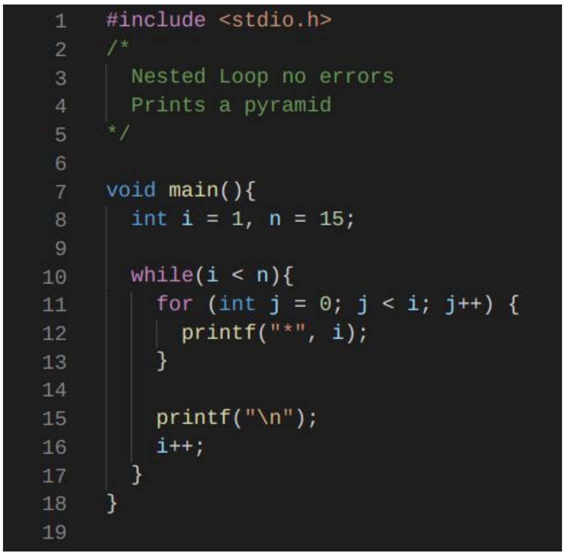


```
omanshu>Lexical-Analyzer $./a.out < Test/1_test_comments.c

line no: 1      #include <stdio.h>      Pre Processor directive
line no: 2      //Comment Testing       SINGLE LINE COMMENT
line no: 4      int     KEYWORD
line no: 4      Mul     IDENTIFIER
line no: 4      (       OPENING BRACKETS
line no: 4      int     KEYWORD
line no: 4      n       IDENTIFIER
line no: 4      ,       COMMA
line no: 4      int     KEYWORD
line no: 4      m       IDENTIFIER
line no: 4      )       CLOSING BRACKETS
line no: 4      {       OPENING BRACES
line no: 5      int     KEYWORD
line no: 5      res     IDENTIFIER
line no: 5      =       OPERATOR
line no: 5      n       IDENTIFIER
line no: 5      *       OPERATOR
line no: 5      m       IDENTIFIER
line no: 5      ;       SEMICOLON
line no: 6      /*
  /*
    This function multiplies 2 numbers
  */    MULTI LINE COMMENT
line no: 7      //      SINGLE LINE COMMENT
line no: 9      return  KEYWORD
line no: 9      res     IDENTIFIER
line no: 9      ;       SEMICOLON
line no: 10     }       CLOSING BRACES
line no: 11     void    KEYWORD
line no: 11     main    KEYWORD
line no: 11     (       OPENING BRACKETS
line no: 11     )       CLOSING BRACKETS
line no: 11     {       OPENING BRACES
line no: 12     int     KEYWORD
line no: 12     n       IDENTIFIER
line no: 12     =       OPERATOR
line no: 12     10      NUMBER CONSTANT
line no: 12     ,       COMMA
line no: 12     m       IDENTIFIER
line no: 12     =       OPERATOR
line no: 12     12      NUMBER CONSTANT
line no: 12     ;       SEMICOLON
```

```
line no: 12     ;       SEMICOLON
line no: 13     /*
  /*
   This program multiplies 2 numbers and prints the output.
  // n = n + 1
  */    MULTI LINE COMMENT
line no: 15     printf  IDENTIFIER
line no: 15     (       OPENING BRACKETS
line no: 15     "\n Result %d"  STRING CONSTANT
line no: 15     ,       COMMA
line no: 15     Mul     IDENTIFIER
line no: 15     (       OPENING BRACKETS
line no: 15     n       IDENTIFIER
line no: 15     ,       COMMA
line no: 15     m       IDENTIFIER
line no: 15     )       CLOSING BRACKETS
line no: 15     )       CLOSING BRACKETS
line no: 15     ;       SEMICOLON
line no: 16     }       CLOSING BRACES


--------------------------------------------------
SYMBOL TABLE
--------------------------------------------------
        m                       IDENTIFIER
        n                       IDENTIFIER
        res                     IDENTIFIER
        return                  KEYWORD
        int                     KEYWORD
        main                    KEYWORD
        Mul                     IDENTIFIER
        printf                  IDENTIFIER
        void                    KEYWORD
--------------------------------------------------


--------------------------------------------------
CONSTANT TABLE
--------------------------------------------------
        "\n Result %d"                  STRING CONSTANT
        10                      NUMBER CONSTANT
        12                      NUMBER CONSTANT
--------------------------------------------------
```

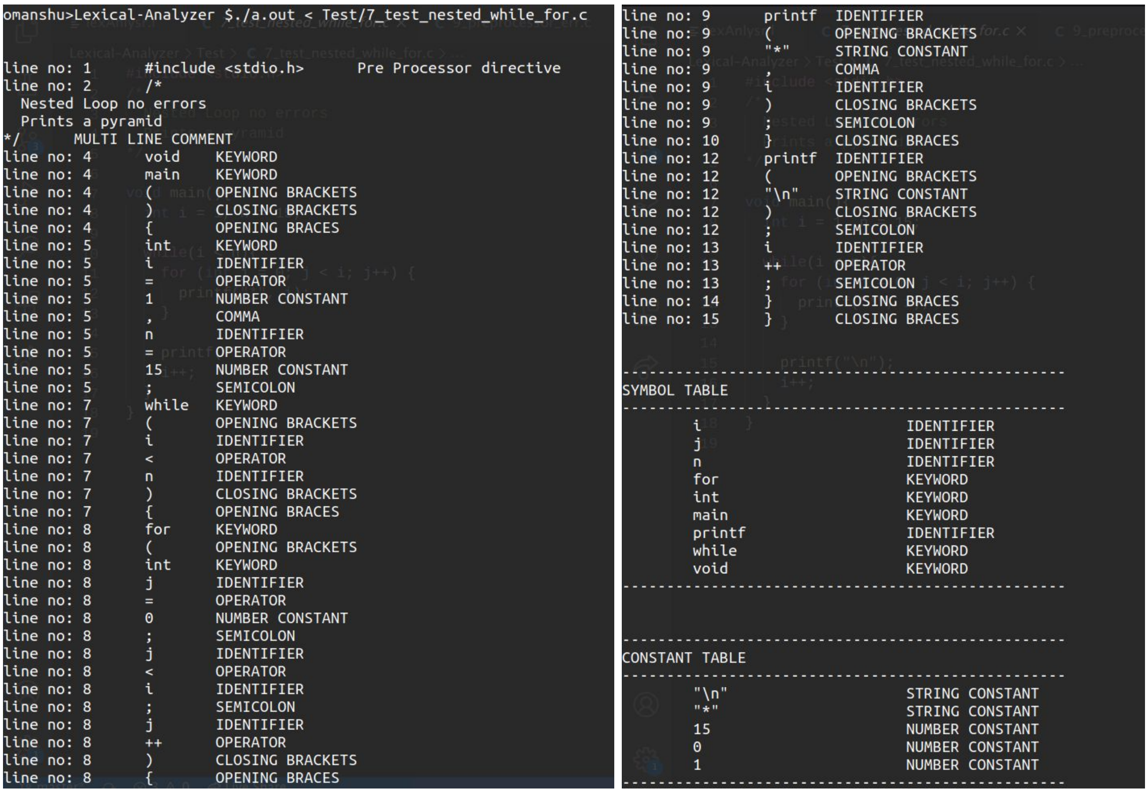
Nested Loop

Figure 3



```c
#include <stdio.h>
/*
  Nested Loop no errors
  Prints a pyramid
*/

void main(){
  int i = 1, n = 15;

  while(i < n){
    for (int j = 0; j < i; j++) {
      printf("*", i);
    }

    printf("\n");
    i++;
  }
}

```

Figure 4

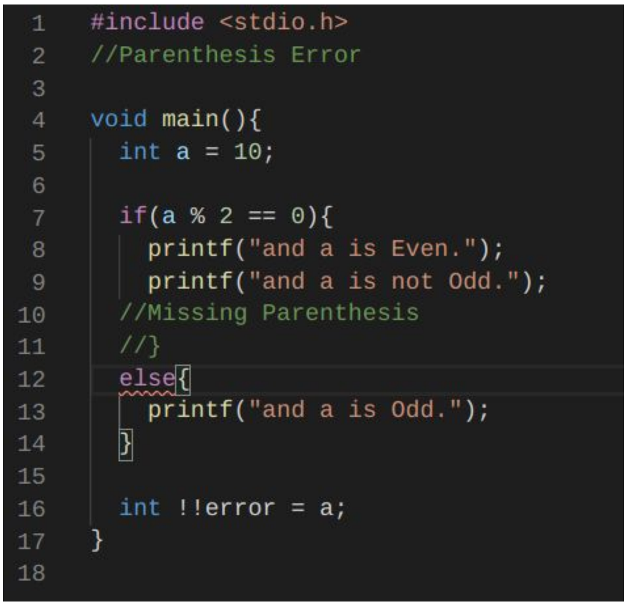


```
omanshu>Lexical-Analyzer $./a.out < Test/7_test_nested_while_for.c

line no: 1      #include <stdio.h>      Pre Processor directive
line no: 2      /*
  Nested Loop no errors
  Prints a pyramid
*/      MULTI LINE COMMENT
line no: 4      void    KEYWORD
line no: 4      main    KEYWORD
line no: 4      (       OPENING BRACKETS
line no: 4      )       CLOSING BRACKETS
line no: 4      {       OPENING BRACES
line no: 5      int     KEYWORD
line no: 5      i       IDENTIFIER
line no: 5      =       OPERATOR
line no: 5      1       NUMBER CONSTANT
line no: 5      ,       COMMA
line no: 5      n       IDENTIFIER
line no: 5      =       OPERATOR
line no: 5      15      NUMBER CONSTANT
line no: 5      ;       SEMICOLON
line no: 7      while   KEYWORD
line no: 7      (       OPENING BRACKETS
line no: 7      i       IDENTIFIER
line no: 7      <       OPERATOR
line no: 7      n       IDENTIFIER
line no: 7      )       CLOSING BRACKETS
line no: 7      {       OPENING BRACES
line no: 8      for     KEYWORD
line no: 8      (       OPENING BRACKETS
line no: 8      int     KEYWORD
line no: 8      j       IDENTIFIER
line no: 8      =       OPERATOR
line no: 8      0       NUMBER CONSTANT
line no: 8      ;       SEMICOLON
line no: 8      j       IDENTIFIER
line no: 8      <       OPERATOR
line no: 8      i       IDENTIFIER
line no: 8      ;       SEMICOLON
line no: 8      j       IDENTIFIER
line no: 8      ++      OPERATOR
line no: 8      )       CLOSING BRACKETS
line no: 8      {       OPENING BRACES
```

```
line no: 9      printf  IDENTIFIER
line no: 9      (       OPENING BRACKETS
line no: 9      "*"     STRING CONSTANT
line no: 9      ,       COMMA
line no: 9      i       IDENTIFIER
line no: 9      )       CLOSING BRACKETS
line no: 9      ;       SEMICOLON
line no: 10     }       CLOSING BRACES
line no: 12     printf  IDENTIFIER
line no: 12     (       OPENING BRACKETS
line no: 12     "\n"    STRING CONSTANT
line no: 12     )       CLOSING BRACKETS
line no: 12     ;       SEMICOLON
line no: 13     i       IDENTIFIER
line no: 13     ++      OPERATOR
line no: 13     ;       SEMICOLON
line no: 14     }       CLOSING BRACES
line no: 15     }       CLOSING BRACES


--------------------------------------------------
SYMBOL TABLE
--------------------------------------------------
        i                       IDENTIFIER
        j                       IDENTIFIER
        n                       IDENTIFIER
        for                     KEYWORD
        int                     KEYWORD
        main                    KEYWORD
        printf                  IDENTIFIER
        while                   KEYWORD
        void                    KEYWORD
--------------------------------------------------


--------------------------------------------------
CONSTANT TABLE
--------------------------------------------------
        "\n"                    STRING CONSTANT
        "*"                     STRING CONSTANT
        15                      NUMBER CONSTANT
        0                       NUMBER CONSTANT
        1                       NUMBER CONSTANT
--------------------------------------------------
```

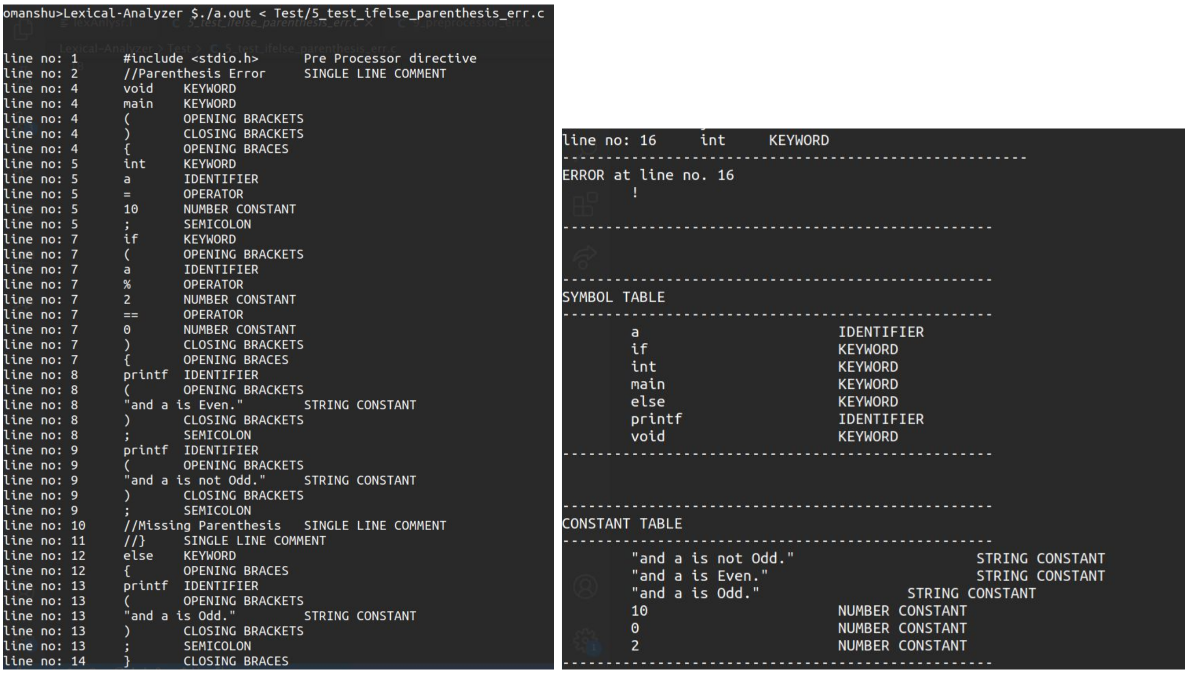
Missing parenthesis

Figure 5



```c
#include <stdio.h>
//Parenthesis Error

void main(){
  int a = 10;

  if(a % 2 == 0){
    printf("and a is Even.");
    printf("and a is not Odd.");
  //Missing Parenthesis
  //}
  else{
    printf("and a is Odd.");
  }

  int !!error = a;
}

```

Figure 6

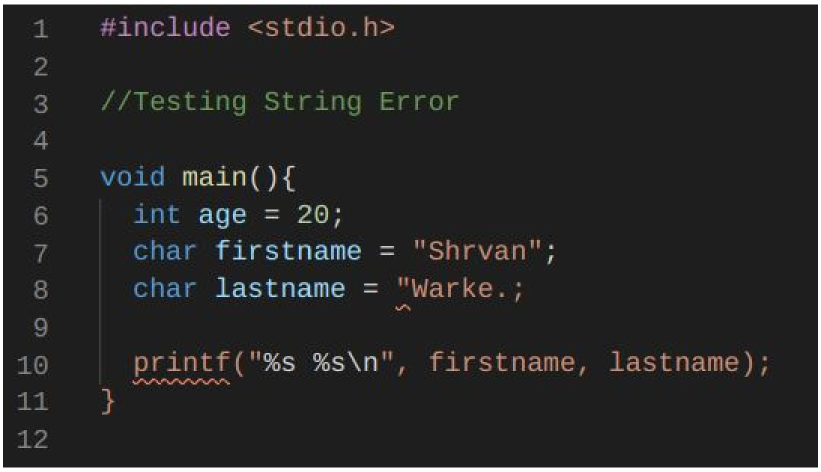


```
omanshu>Lexical-Analyzer $./a.out < Test/5_test_ifelse_parenthesis_err.c

line no: 1      #include <stdio.h>      Pre Processor directive
line no: 2      //Parenthesis Error     SINGLE LINE COMMENT
line no: 4      void    KEYWORD
line no: 4      main    KEYWORD
line no: 4      (       OPENING BRACKETS
line no: 4      )       CLOSING BRACKETS
line no: 4      {       OPENING BRACES
line no: 5      int     KEYWORD
line no: 5      a       IDENTIFIER
line no: 5      =       OPERATOR
line no: 5      10      NUMBER CONSTANT
line no: 5      ;       SEMICOLON
line no: 7      if      KEYWORD
line no: 7      (       OPENING BRACKETS
line no: 7      a       IDENTIFIER
line no: 7      %       OPERATOR
line no: 7      2       NUMBER CONSTANT
line no: 7      ==      OPERATOR
line no: 7      0       NUMBER CONSTANT
line no: 7      )       CLOSING BRACKETS
line no: 7      {       OPENING BRACES
line no: 8      printf  IDENTIFIER
line no: 8      (       OPENING BRACKETS
line no: 8      "and a is Even."        STRING CONSTANT
line no: 8      )       CLOSING BRACKETS
line no: 8      ;       SEMICOLON
line no: 9      printf  IDENTIFIER
line no: 9      (       OPENING BRACKETS
line no: 9      "and a is not Odd."     STRING CONSTANT
line no: 9      )       CLOSING BRACKETS
line no: 9      ;       SEMICOLON
line no: 10     //Missing Parenthesis   SINGLE LINE COMMENT
line no: 11     //}     SINGLE LINE COMMENT
line no: 12     else    KEYWORD
line no: 12     {       OPENING BRACES
line no: 13     printf  IDENTIFIER
line no: 13     (       OPENING BRACKETS
line no: 13     "and a is Odd."         STRING CONSTANT
line no: 13     )       CLOSING BRACKETS
line no: 13     ;       SEMICOLON
line no: 14     }       CLOSING BRACES
```

```
line no: 16     int     KEYWORD
-----------------------------------------------------
ERROR at line no. 16
        !

-------------------------------------------------


-------------------------------------------------
SYMBOL TABLE
-------------------------------------------------
        a                       IDENTIFIER
        if                      KEYWORD
        int                     KEYWORD
        main                    KEYWORD
        else                    KEYWORD
        printf                  IDENTIFIER
        void                    KEYWORD
-------------------------------------------------


-------------------------------------------------
CONSTANT TABLE
-------------------------------------------------
        "and a is not Odd."                     STRING CONSTANT
        "and a is Even."                        STRING CONSTANT
        "and a is Odd."                 STRING CONSTANT
        10                      NUMBER CONSTANT
        0                       NUMBER CONSTANT
        2                       NUMBER CONSTANT
-------------------------------------------------
```

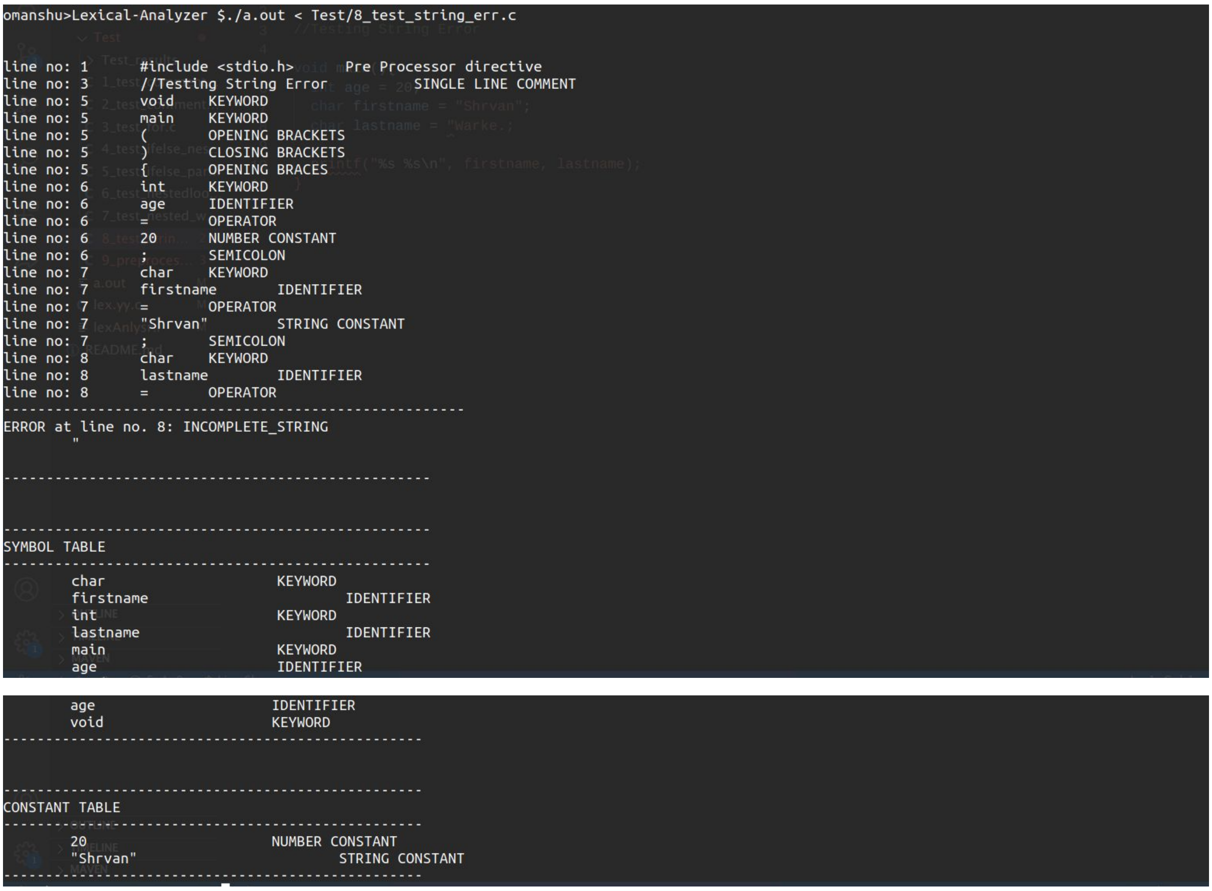
Incomplete String

Figure 7



```c
#include <stdio.h>

//Testing String Error

void main(){
    int age = 20;
    char firstname = "Shrvan";
    char lastname = "Warke.;

    printf("%s %s\n", firstname, lastname);
}

```

Figure 8

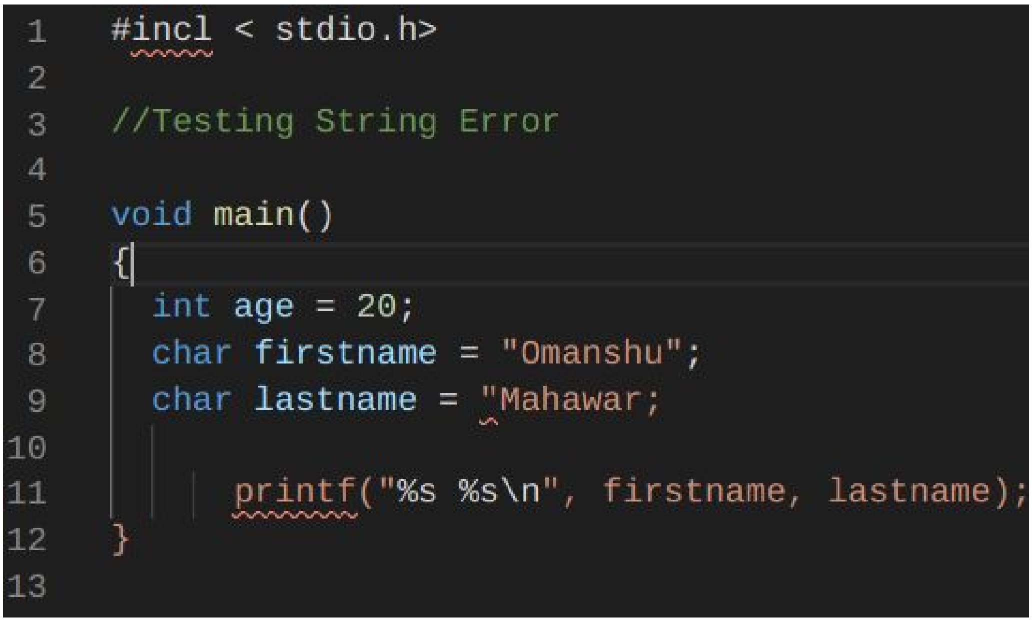


```
omanshu>Lexical-Analyzer $./a.out < Test/8_test_string_err.c

line no: 1      #include <stdio.h>      Pre Processor directive
line no: 3      //Testing String Error  SINGLE LINE COMMENT
line no: 5      void    KEYWORD
line no: 5      main    KEYWORD
line no: 5      (       OPENING BRACKETS
line no: 5      )       CLOSING BRACKETS
line no: 5      {       OPENING BRACES
line no: 6      int     KEYWORD
line no: 6      age     IDENTIFIER
line no: 6      =       OPERATOR
line no: 6      20      NUMBER CONSTANT
line no: 6      ;       SEMICOLON
line no: 7      char    KEYWORD
line no: 7      firstname       IDENTIFIER
line no: 7      =       OPERATOR
line no: 7      "Shrvan"        STRING CONSTANT
line no: 7      ;       SEMICOLON
line no: 8      char    KEYWORD
line no: 8      lastname        IDENTIFIER
line no: 8      =       OPERATOR
------------------------------------------------------
ERROR at line no. 8: INCOMPLETE_STRING
        "

--------------------------------------------------


--------------------------------------------------
SYMBOL TABLE
--------------------------------------------------
        char                    KEYWORD
        firstname                       IDENTIFIER
        int                     KEYWORD
        lastname                        IDENTIFIER
        main                    KEYWORD
        age                     IDENTIFIER
```

```
        age                     IDENTIFIER
        void                    KEYWORD
-------------------------------------------------


-------------------------------------------------
CONSTANT TABLE
-------------------------------------------------
        20                      NUMBER CONSTANT
        "Shrvan"                        STRING CONSTANT
-------------------------------------------------
```

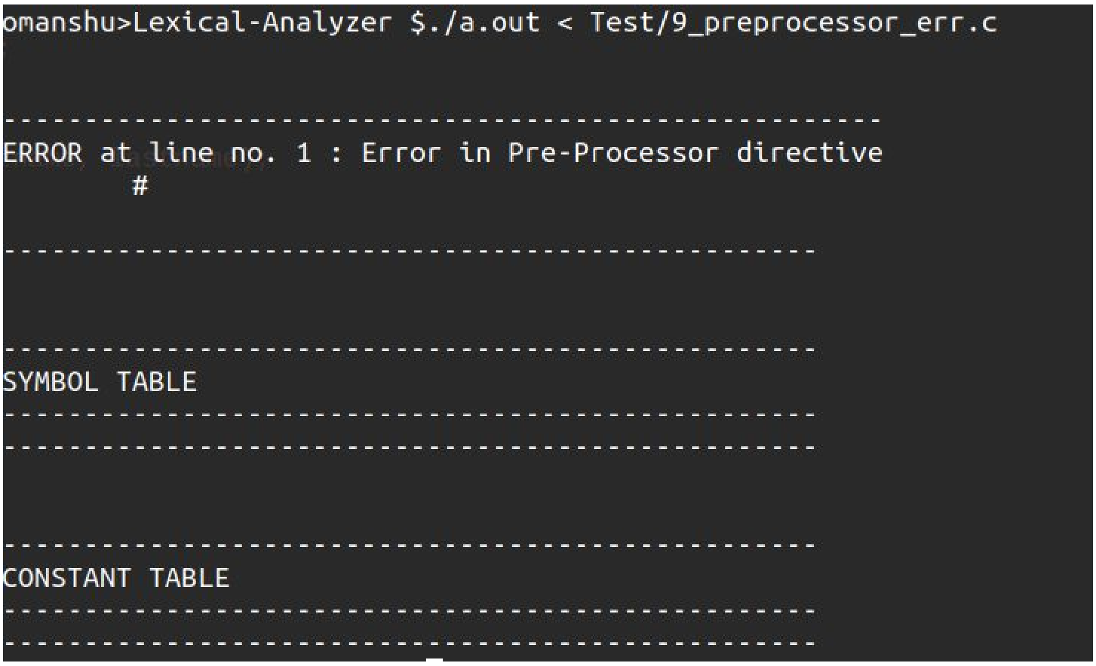
Error in Preprocessor directive

Figure 9



```c
#incl < stdio.h>

//Testing String Error

void main()
{
    int age = 20;
    char firstname = "Omanshu";
    char lastname = "Mahawar;

            printf("%s %s\n", firstname, lastname);
}

```

Figure 10



```
omanshu>Lexical-Analyzer $./a.out < Test/9_preprocessor_err.c


-----------------------------------------------------
ERROR at line no. 1 : Error in Pre-Processor directive
        #

-------------------------------------------------



-------------------------------------------------
SYMBOL TABLE
-------------------------------------------------
-------------------------------------------------



-------------------------------------------------
CONSTANT TABLE
-------------------------------------------------
-------------------------------------------------
```

Implementation

The Regular Expressions used for each different segment of the C programming language are listed below :

- **Preprocessor directives**

  Statements processed : #include<stdio.h>, #define

  Identified using:

  ```
  [#][" "]*(include)[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"]
  ```

- **Keywords:**

  Statements processed: auto, const, default, enum, extern, register, return, sizeof, static, struct, typedef, union, volatile, break, continue, goto, else, switch, if, case, for, do, while, char, double, float, int, long, short, signed, unsigned, void and so on.

  Tokens generated: Keyword

  Identified using:

  ```
  auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern|return|
  union|continue|for|signed|void|do|if|static|while|default|goto|sizeof|volatile|const|float|short|
  unsigned|main/[\(|" "|\{|;|:|"\n"|"\t"]
  ```

- **Identifiers:**

  Statements processed : a, abc, a_b, a12b4

  Tokens generated: Identifier

  Identified using:

  ```
  [a-z|A-Z]([a-z|A-Z]|[0-9])*/\
  ```

- **Operators:**

  Statements processed : +, -, *, /, %

  Tokens generated: Operators

  Identified using

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

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

- **Single-line comments:**

  Statements processed : //..........

  Identified using: \/\/(.*)

- **Multi-line comments:**
  Statements processed : /*..........*/, /*.../*...*/ .... */
  Identified using: \/*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))**+\/

- **Parentheses (all types):**
  Statements processed : (..), {..}, [..] (without errors) (..)..), {..}..}, [..]..], (...., [... (with errors)

  Tokens generated: Parenthesis (without error) / Error with line number (with error)

There can be various Errors in the C program that should be displayed and some of them are handled here as follows:

- **Errors for Pre Processor directives:**
  Identified using: (yytext[0]=='#')

- **Errors for incomplete strings:**
  Statements processed: char a[]= "abcd Error generated: Error Incomplete string and line number
  Identified using: (yytext[0]=='"')

- **Errors for nested comments:**
  Statements processed : /*....../*....*/....
  Errors generated: Error with line number.

- **Errors for unmatched comments:**
  Statements processed : /*.........
  Identified using: (yytext[0]=='/')

After recognizing all tokens, the lexer analyzes and prints the different identifiers, literal, and constants with the line number. The following is used for this :

- Two main structures are used to form the Symbol Table and the Constant Table which contains the identifiers and the constants.
- The lookup() function is used to check if the given identifier or constant is already present in the respective Symbol table or Constant table. The insert() function then adds the identifier or constant if it is not present.
- For every Identifier/Constant the lookup() and the insert() functions are called and then put them in their respective tables.
- At the end of the main() function, we call the printTables() function which prints the Symbol Table and the Constant Table.

Result

The above lexical analyzer generates the following output:

```
Token Token_Type

Symbol Table
Token | Token_Type

Constant Table
Token | Token_Type
```

Future Work

The Lexical Analyser helps to break down the source c program in tokens that are defined by the C Programming Language.
The flex script presented in this report takes care of all the rules of C language but is not fully exhaustive. Our future work would include making the script even more robust to handle all aspects of C language and making it more efficient
For the next phase, a parser will be designed, which will call the Flex program to give it tokens. The lexical analyzer cannot find the syntactical errors or find unmatched parenthesis, to do this the parser to be designed in the next phase is used.

References

- Compilers – Principles, Techniques, and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.
- https://www.geeksforgeeks.org/cc-tokens/
- http://www.isi.edu/~pedro/Teaching/CSCI565-Spring11/Practice/SDT-Sample.pdf
- StackOverflow for regex
- http://dinosaur.compilertools.net/lex/index.html