Lexical Analyzer for the C Language

National Institute of Technology Karnataka Surathkal

Date:
26th August 2020

Submitted to:
Prof. P. Santhi Thilagam

Group Members:
Niranjan S Yadiyala (181CO136)
Rajath C Aralikatti (181CO241)
Shruthan R (181CO250)
Varun NR (181CO134)

Abstract

Objective:
Build a lexical analyzer for the C language using LEX.

Summary:
The lexical analyzer can achieve the following:

- Identify and differentiate between keywords, identifiers, constants, operators (arithmetic, comparison and bitwise), strings, comments and other symbols (like brackets - square, round and curly).
- Detect looping constructs like for loops and while loops and conditional statements (if - else if - else).
- Detect single line and multiline comments.
- Detect extended data types like signed, unsigned, long, short for integers and characters.
- Detect arrays with specified data types.
- Detect errors like unclosed comments, brackets and strings.
- Build a symbol table by implementing hashing.

The above-mentioned have been implemented in two files:

- `code.c:` Has code related to the implementation of the hashing operations used to build the symbol table.
- `scanner.l :` Parse through the C program to identify the tokens.

The object files of these are to be run together. This has been implemented in `runfile.sh`. To run the files, use `./runfile.sh` (after giving execution privileges using chmod).

Requirements:

- GCC, the GNU Compiler Collection
- Flex (Fast Lexical Analyzer Generator)

Table of Contents

1. Introduction

1.1. Lexical Analyzer

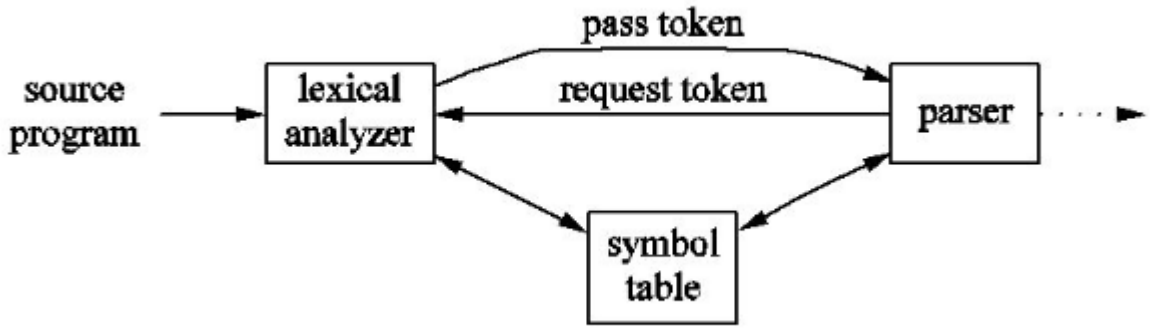
Lexical analysis is the first phase of a compiler which takes a high-level input program as input and converts it into a sequence of tokens. A lexical analyzer has the following functions:

- Tokenization
- Removing whitespace
- Removing comments
- Detecting errors and the line number in which the error occurred.

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions. A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages. Some examples include keywords (like `int, if, for),` symbols (like `+, -, *, %, /, &, etc`)

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 on demand.

A diagrammatic representation of the lexical analyzer working in tandem with the parser is shown below:



1.2 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 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.

Splitting of the Rules section and C code section:

In our project, we have separated the rule section and the C code section to adhere to the “Separation of Concerns” principle of software engineering. The two files are `code.c and scanner.l` in our project. In order to use them together, the object file of the scanner is first generated pass passing the `-c` flag while passing `lex.yy.c` to gcc to generate `lex.yy.o.` Then, `lex.yy.o` and `code.c` are together passed to gcc to generate `a.out.` All these are implemented in `runfile.sh.`

2. Implementation

2.1. Code

The lexical analyzer is implemented in two parts:

- `code.c` : A C program to read the input C program and implement the symbol table. The code for it is given below:

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

struct Table{
    char name[100];
    char type[100];
    int len;
} symbolTable[1001];

int hashFunction(char *s){
    int mod = 1001;
    int l = strlen(s), val = 0, i;
    for(i = 0; i < l; i++){
        val = val * 10 + (s[i]-'A');
        val = val % mod;
        while(val < 0){
            val += mod;
        }
    }
    return val;
}

void insertToken(char *token, char *tokenType){

    int l1 = strlen(token);
    int l2 = strlen(tokenType);
    int v = hashFunction(token);
    if(symbolTable[v].len == 0){
        strcpy(symbolTable[v].name, token);
        strcpy(symbolTable[v].type, tokenType);

        symbolTable[v].len = strlen(token);
        return;
    }

    if (strcmp(symbolTable[v].name,token) == 0)
        return;

    int i, pos = 0;

    for (i = 1; i < 1001; i++){
        int x = (i+v)%1001;
        if (strcmp(symbolTable[x].name,token) == 0)
```

```
            return;
        if(symbolTable[x].len == 0){

            pos = x;
            break;
        }
    }

    strcpy(symbolTable[pos].name, token);
    strcpy(symbolTable[pos].type, tokenType);
    symbolTable[pos].len = strlen(token);

}

void print(){
    int i;
    for(i = 0;i < 1001; i++){
        if(symbolTable[i].len == 0){
            continue;
        }
        printf("%15s \t
%40s\n",symbolTable[i].name,symbolTable[i].type);
    }
}


extern FILE* yyin;
int main(){

    int i;
    for (i = 0; i < 1001; i++){
        strcpy(symbolTable[i].name, "");
        strcpy(symbolTable[i].type, "");
        symbolTable[i].len = 0;
    }
    yyin = fopen("test.c","r");

    yylex();
    printf("\n\n-----------------------------------------------------
-------------------------\n\t\t\t\tSYMBOL TABLE\n----------------
-------------------------------------------------------------\n");
      printf("\tToken \t\t\t\t\tToken Type\n");
      printf("-----------------------------------------------------
-----------------------\n");
    print();
}
```

Symbol Table

Implemented as a hash-table, it is maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search for the

identifier record and retrieve it. The `insertToken` function inserts the new token into the symbol table at the location given by the hash function implemented in `hashFunction`. Linear probing is used to resolve collisions in the hash function.
The input C file is stored in `yyin` which is an `extern` variable. It is used in the scanner as shown below.

- `scanner.l :` Has the lex code to detect the various parts of the grammar of the C programming language.

```
%{
    #include<stdio.h>
    #include<string.h>
      #include <stdlib.h>
    #include "pretty_print.h"

    int stackTop = 0;
      int nestedCommentStack = 0;
      int line = 0;
      char parenthesisStack[100];
     int comment_nesting = 0;


%}

%x COMMENT
%x SC_COMMENT

Preprocessor
#(include<.*>|define.*|ifdef|endif|if|else|ifndef|undef|pragma)
ArithmeticOperator \+\+|\-\-|\+|\-|\*|\/|=
ComparisionOperator <=|>=|<|>
BitwiseOperator \^|\%|\&|\||\~
Identifier [a-zA-Z_]([a-zA-Z0-9_])*
NumericConstant [1-9][0-9]*(\.[0-9]+)?|0(\.[0-9]+)?
String \".*\"|\'.*\'
InvalidString \"[^"\n]*|\'[^'\n]*
SingleLineComment \/\/.*
MultiLineComment "/*"([^*]|\*+[^*/])*\*+"/"
Keyword
auto|const|default|enum|extern|register|return|sizeof|static|stru
ct|typedef|union|volatile|break|continue|goto|else|switch|if|case
|default|for|do|while|char|double|float|int|long|short|signed|uns
igned|void
InvalidID [^\n\t ]
InvalidIdentifier ([0-9\*\-\+\%\/]+[a-zA-Z][a-zA-Z0-9\*\-
\+\%\/]*)

%%
\n line++;
```

```
[\t ] ;
; {printf("%s \t---- Semicolon Delimiter\n", yytext);}
, {printf("%s \t---- Comma Delimiter\n", yytext);}

\{ {
    printf("%s \t---- Parenthesis\n", yytext);
    parenthesisStack[stackTop]='{';
      stackTop++;

}
\} {
    printf("%s \t---- Parenthesis\n", yytext);
      if(stackTop == 0 || parenthesisStack[stackTop-1] != '{')
            printf("ERROR: Unbalanced parenthesis at line number:
%d\n",line);
      stackTop--;
}
\( {
    printf("%s \t---- Parenthesis\n", yytext);
      parenthesisStack[stackTop]='(';
    stackTop++;
}
\) {
    printf("%s \t---- Parenthesis\n", yytext);
      if(stackTop == 0 || parenthesisStack[stackTop-1] != '(')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      stackTop--;
}

\[ {
    printf("%s \t---- Parenthesis\n", yytext);
      parenthesisStack[stackTop]='[';
    stackTop++;
}

\] {
    printf("%s \t---- Parenthesis\n", yytext);
      if (stackTop == 0 || parenthesisStack[stackTop-1] != '[')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      stackTop--;
}
\\ {
    printf("%s \t- Backward Slash\n", yytext);
}
\. {
    printf("%s \t- Dot Delimiter\n", yytext);
}


(main\(\)) {
    printf("%s \t- Main Function\n", yytext);
```

```
}

(main\(void\)) {
    printf("%s \t- Main Function\n", yytext);
}

(main\(int[ ]+{Identifier}\,[ ]+char\*\*[ ]+{Identifier}\)) {
    printf("%s \t- Main Function\n", yytext);
}


"/*" {
    BEGIN COMMENT;
    nestedCommentStack = 1;
    yymore();
}

<COMMENT>{
    "/*" {
        nestedCommentStack++;

        yymore();
    }

    "*/" {
        nestedCommentStack--;
        if (nestedCommentStack<0)
        {
            printf("\n \"%s\"\t---- ERROR: Unbalanced COMMENT at
line: %d.", yytext, line);
            yyterminate();
        }
        else if (nestedCommentStack==0)
        {

            BEGIN(INITIAL);
        }
        else
            yymore();
    }
    <<EOF>> {
        printf("\nERROR: Multiline Comment: \"");
        yyless(yyleng-2);
        ECHO;
        printf("\", Doesn't terminate at line: %d",line);
        yyterminate();
    }
    . {
        yymore();
    }
    \n {
        yymore();
        line++;
```

```
    }
}

"*/"    {

    printf("%s \t---- ERROR: Unintialized COMMENT at line: %d\n",
yytext,line);
    yyterminate();
}

//".*  {
    printf("%s \t---- Single line COMMENT\n", yytext);
}

{Preprocessor} {
    printf("%s \t---- Preprocessor Directive\n", yytext);
}
{String} {
    printf("%s \t---- String \n", yytext);
    insertToken(yytext,"String Constant");
}
{MultiLineComment} {
    printf("%s \t---- Multi-Line Comment\n", yytext);
}
{Keyword} {
    printf("%s \t---- Keyword\n", yytext);
    insertToken(yytext, "Keyword");
}
{Identifier} {
    printf("%s \t---- Identifier\n", yytext);
    insertToken(yytext, "Identifier");
}
{InvalidIdentifier} {
    printf("%s \t---- ERROR: Invalid Identifier\n", yytext);
}
{NumericConstant} {
    printf("%s \t---- Numeric Constant\n", yytext);
    insertToken(yytext, "Numeric Constant");
}
{ArithmeticOperator} {
    printf("%s \t---- Arithmetic\n", yytext);
}
{BitwiseOperator} {
    printf("%s \t---- Bitwise Operator\n", yytext);
}
{ComparisionOperator} {
    printf("%s \t---- Comparision Operator\n", yytext);
}
{InvalidString} {
    printf("%s \t---- ERROR: Unterminated string at line number:
%d\n", yytext,line);
}
{InvalidID} {
```

```
    printf("%s \t---- ERROR: Invalid identifier at line number:
%d\n", yytext,line);
}
%%

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

The following class of tokens can be detected:

● **Preprocessor directives:**

Statements processed : `#include, #define var1 var2`,

Token generated : Preprocessor Directive

● **Operators**

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

Tokens generated : Operators

● **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`.

Tokens generated: Keyword

● **Identifiers**

Statements processed : `printf, i, varName` etc.

Tokens generated : Identifier

● **Single-line comments**:

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

● **Multi-line comments:**

Statements processed : `/*..........*/, /*.../*...*/*...*/`

● **Brackets:**

Statements processed : `(..), {..}, [..]` (without errors)

`(..)..), {..}..}, [..]..], (..., [...` (with errors)

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

● **User defined functions:**

Statements processed : User defined functions with parameters.

● **Errors for incomplete strings**

Statements processed : `char a[]= ”abcd`

Error generated: `Error Incomplete string and line number`

● **Errors for nested comments**

Statements processed : `/*....../*....*/....`

Error generated: `Error with line number`

● **Errors for unmatched comments**

Statements processed : /*.....

Error generated: `Error with line number`

2.2 Some details

Regular expressions to identify the different tokens are given below:

- Preprocessor Directives:
  `#(include<.*>|define.*|ifdef|endif|if|else|ifndef|undef|pragma)`
- Arithmetic operators:
  `\+\+|\-\-|\+|\-|\*|\/|=`
- Comparison Operators:
  `<=|>=|<|>`
- Bitwise Operator
  `\^|\%|\&|\||\~`
- Identifier
  `[a-zA-Z_]([a-zA-Z0-9_])*`
- Numeric Constant
  `[1-9][0-9]*(\.[0-9]+)?|0(\.[0-9]+)?`
- String
  `\".*\"|\'.*\'`
- Invalid String
  `\"[^"\n]*|\'[^'\n]*`
- Single Line Comment
  `\/\/.*`
- Multi line Comment
  `"/*"([^*]|\*+[^*/])*\*+"/"`
- InvalidID
  `[^\n\t ]`
- Invalid Identifier
  `([0-9\*\-\+\%\/]+[a-zA-Z][a-zA-Z0-9\*\-\+\%\/]*)`

After all tokens have been detected, they are printed to the terminal. As successive tokens are encountered, their values are updated in the symbol table and the symbol table is displayed at the end.

3. Testing the lexical analyzer:

Given below are a few test programs and their outputs:

Test 1

A program with no errors.

```
#include<stdio.h>
int main()
{
    int a,b,c;
    a = b + c;
    printf("Sum is %d",a);
    return 0;
}
shruthan@DESKTOP-QMQLGCS:/mnt/c/Users/shrut/Compiler-Design/Lexical
Analyzer$ ./runfile.sh
#include<stdio.h>       ---- Preprocessor Directive
int     ---- Keyword
```

```
main()  - Main Function
{       ---- Parenthesis
int     ---- Keyword
a       ---- Identifier
,       ---- Comma Delimiter
b       ---- Identifier
,       ---- Comma Delimiter
c       ---- Identifier
;       ---- Semicolon Delimiter
a       ---- Identifier
=       ---- Arithmetic
b       ---- Identifier
+       ---- Arithmetic
c       ---- Identifier
;       ---- Semicolon Delimiter
printf  ---- Identifier
(       ---- Parenthesis
"Sum is %d"     ---- String
,       ---- Comma Delimiter
a       ---- Identifier
)       ---- Parenthesis
;       ---- Semicolon Delimiter
return  ---- Keyword
0       ---- Numeric Constant
;       ---- Semicolon Delimiter
}       ---- Parenthesis


--------------------------------------------------------------------------
------
                                SYMBOL TABLE
--------------------------------------------------------------------------
------
        Token                                           Token Type
--------------------------------------------------------------------------
------
             a                                        Identifier
             b                                        Identifier
             c                                        Identifier
         return                                          Keyword
            int                                          Keyword
         printf                                      Identifier
    "Sum is %d"                                String Constant
```

Test 2

A program with comment imbalance.

```c
#include<stdio.h>
int main()
{
    int a[5], b[5], c[5];
    int i;
    for (i = 0; i < 5; i++) {

        a[i] = 1;
        b[i] = i;
```

```
    }
    i=0;
    while(i < 5)
    {
        c[i] = a[i] + b[i];
        i++;
    }
     /*
    This File Contains Test cases about Comments and Parenthesis imbala
nce
}
shruthan@DESKTOP-QMQLGCS:/mnt/c/Users/shrut/Compiler-Design/Lexical
Analyzer$ ./runfile.sh
#include<stdio.h>       ---- Preprocessor Directive
int     ---- Keyword
main()  - Main Function
{       ---- Parenthesis
int     ---- Keyword
a       ---- Identifier
[       ---- Parenthesis
5       ---- Numeric Constant
]       ---- Parenthesis
,       ---- Comma Delimiter
b       ---- Identifier
[       ---- Parenthesis
5       ---- Numeric Constant
]       ---- Parenthesis
,       ---- Comma Delimiter
c       ---- Identifier
[       ---- Parenthesis
5       ---- Numeric Constant
]       ---- Parenthesis
;       ---- Semicolon Delimiter
int     ---- Keyword
i       ---- Identifier
;       ---- Semicolon Delimiter
for     ---- Keyword
(       ---- Parenthesis
i       ---- Identifier
=       ---- Arithmetic
0       ---- Numeric Constant
;       ---- Semicolon Delimiter
i       ---- Identifier
<       ---- Comparision Operator
5       ---- Numeric Constant
;       ---- Semicolon Delimiter
i       ---- Identifier
++      ---- Arithmetic
)       ---- Parenthesis
{       ---- Parenthesis
a       ---- Identifier
[       ---- Parenthesis
i       ---- Identifier
]       ---- Parenthesis
=       ---- Arithmetic
1       ---- Numeric Constant
;       ---- Semicolon Delimiter
```

```
b       ---- Identifier
[       ---- Parenthesis
i       ---- Identifier
]       ---- Parenthesis
=       ---- Arithmetic
i       ---- Identifier
;       ---- Semicolon Delimiter
}       ---- Parenthesis
i       ---- Identifier
=       ---- Arithmetic
0       ---- Numeric Constant
;       ---- Semicolon Delimiter
while   ---- Keyword
(       ---- Parenthesis
i       ---- Identifier
<       ---- Comparision Operator
5       ---- Numeric Constant
)       ---- Parenthesis
{       ---- Parenthesis
c       ---- Identifier
[       ---- Parenthesis
i       ---- Identifier
]       ---- Parenthesis
=       ---- Arithmetic
a       ---- Identifier
[       ---- Parenthesis
i       ---- Identifier
]       ---- Parenthesis
+       ---- Arithmetic
b       ---- Identifier
[       ---- Parenthesis
i       ---- Identifier
]       ---- Parenthesis
;       ---- Semicolon Delimiter
i       ---- Identifier
++      ---- Arithmetic
;       ---- Semicolon Delimiter
}       ---- Parenthesis

ERROR: Multiline Comment: " This File Contains Test cases about
Comments and Parenthesis imbalance
", Doesn't terminate at line: 17

------------------------------------------------------------------------
------
                              SYMBOL TABLE
------------------------------------------------------------------------
------
        Token                                            Token Type
------------------------------------------------------------------------
------
              a                                        Identifier
              b                                        Identifier
              c                                        Identifier
              i                                        Identifier
            for                                           Keyword
            int                                           Keyword
          while                                           Keyword
              0                                 Numeric Constant
```

```
                1                                       Numeric Constant
                5                                       Numeric Constant
```

Test 3

A program with an unclosed string

```
#include<stdio.h>

int main(void)
{
    char string[10];
    string = "Hello World!;
}
shruthan@DESKTOP-QMQLGCS:/mnt/c/Users/shrut/Compiler-Design/Lexical
Analyzer$ ./runfile.sh
#include<stdio.h>       ---- Preprocessor Directive
int     ---- Keyword
main(void)      - Main Function
{       ---- Parenthesis
char    ---- Keyword
string  ---- Identifier
[       ---- Parenthesis
10      ---- Numeric Constant
]       ---- Parenthesis
;       ---- Semicolon Delimiter
string  ---- Identifier
=       ---- Arithmetic
"Hello World!;  ---- ERROR: Unterminated string at line number: 5
}       ---- Parenthesis


------------------------------------------------------------------------
------
                                SYMBOL TABLE
------------------------------------------------------------------------
------
        Token                                           Token Type
------------------------------------------------------------------------
------
          char                                            Keyword
           int                                            Keyword
            10                                  Numeric Constant
        string                                        Identifier
```

Test 4

A program with no errors, many comments and #define.

```
/* Program to multiply by 10
*/
// Header Files
#include <stdio.h>

#define ten 10

```

```
int main(void) {
    // Prompt input
    printf("Enter the number to be multiplied by 10\n");
    int n;
    // Take input
    scanf("%d", &n);
    return ten*n;
}
```

```
shruthan@DESKTOP-QMQLGCS:/mnt/c/Users/shrut/Compiler-Design/Lexical
Analyzer$ ./runfile.sh
/* Program to  multiply by 10 */        ---- Multi-Line Comment
// Header Files         ---- Single line COMMENT
#include<stdio.h>       ---- Preprocessor Directive
#define ten 10  ---- Preprocessor Directive
int     ---- Keyword
main(void)    - Main Function
{       ---- Parenthesis
// Prompt input         ---- Single line COMMENT
printf  ---- Identifier
(       ---- Parenthesis
"Enter the number to be multiplied by 10\n"     ---- String
)       ---- Parenthesis
;       ---- Semicolon Delimiter
int     ---- Keyword
n       ---- Identifier
;       ---- Semicolon Delimiter
// Take input   ---- Single line COMMENT
scanf   ---- Identifier
(       ---- Parenthesis
"%d"    ---- String
,       ---- Comma Delimiter
&       ---- Bitwise Operator
n       ---- Identifier
)       ---- Parenthesis
;       ---- Semicolon Delimiter
return  ---- Keyword
ten     ---- Identifier
*n      ---- ERROR: Invalid Identifier
;       ---- Semicolon Delimiter
}       ---- Parenthesis


------------------------------------------------------------------------
------
                              SYMBOL TABLE
------------------------------------------------------------------------
------
        Token                                           Token Type
------------------------------------------------------------------------
------
              n                                       Identifier
          scanf                                       Identifier
         return                                          Keyword
            int                                          Keyword
            ten                                       Identifier
           "%d"                                 String Constant
           main                                       Identifier
"Enter the number to be multiplied by 10\n"
```

```
   String Constant
            printf                                          Identifier
              void                                             Keyword
```

Test 5

A program with an invalid assignment.

```c
#include<stdio.h>

int main(void) {
    // Prompt input
    printf("Enter the number to be multiplied by 10\n");
    int n;
    int ten = 1xabc;
    // Take input
    scanf("%d", &n);
    return ten * n;
}
```

```
shruthan@DESKTOP-QMQLGCS:/mnt/c/Users/shrut/Compiler-Design/Lexical Analyzer$
./runfile.sh
#include<stdio.h>       ---- Preprocessor Directive
int     ---- Keyword
main(void)    - Main Function
{       ---- Parenthesis
// Prompt input         ---- Single line COMMENT
printf  ---- Identifier
(       ---- Parenthesis
"Enter the number to be multiplied by 10\n"     ---- String
)       ---- Parenthesis
;       ---- Semicolon Delimiter
int     ---- Keyword
n       ---- Identifier
;       ---- Semicolon Delimiter
int     ---- Keyword
ten     ---- Identifier
=       ---- Arithmetic
1xabc   ---- ERROR: Invalid Identifier
;       ---- Semicolon Delimiter
// Take input   ---- Single line COMMENT
scanf   ---- Identifier
(       ---- Parenthesis
"%d"    ---- String
,       ---- Comma Delimiter
&       ---- Bitwise Operator
n       ---- Identifier
)       ---- Parenthesis
;       ---- Semicolon Delimiter
return  ---- Keyword
ten     ---- Identifier
*       ---- Arithmetic
n       ---- Identifier
;       ---- Semicolon Delimiter
}       ---- Parenthesis


--------------------------------------------------------------------------------
                                SYMBOL TABLE
--------------------------------------------------------------------------------
        Token                                           Token Type
```

```
------------------------------------------------------------------------
              n                                          Identifier
          scanf                                          Identifier
         return                                             Keyword
            int                                             Keyword
            ten                                          Identifier
           "%d"                                    String Constant
           main                                          Identifier
"Enter the number to be multiplied by 10\n"
String Constant
         printf                                          Identifier
           void                                             Keyword
```

Test 6

Missing parenthesis. Main function also takes command line arguments, which are identified as a part of the main function.

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

int main(int argc, char** argv) {
    int l = strlen(argv;
    return 0;
}
shruthan@DESKTOP-QMQLGCS:/mnt/c/Users/shrut/Compiler-Design/Lexical Analyzer$
./runfile.sh
#include<stdio.h>       ---- Preprocessor Directive
#include<string.h>      ---- Preprocessor Directive
int     ---- Keyword
main(int argc, char** argv)     - Main Function
{       ---- Parenthesis
int     ---- Keyword
l       ---- Identifier
=       ---- Arithmetic
strlen  ---- Identifier
(       ---- Parenthesis
argv    ---- Identifier
;       ---- Semicolon Delimiter
return  ---- Keyword
0       ---- Numeric Constant
;       ---- Semicolon Delimiter
}       ---- Parenthesis
ERROR: Unbalanced parenthesis at line number: 6


------------------------------------------------------------------------
                              SYMBOL TABLE
------------------------------------------------------------------------
        Token                                          Token Type
------------------------------------------------------------------------
              l                                          Identifier
         strlen                                          Identifier
           argv                                          Identifier
         return                                             Keyword
```

```
            int                                          Keyword
              0                                  Numeric Constant
```

4. Some Implementational Details:

Multiline comments: This has been supported by checking the occurence of '/*' and '*/' in the code. The statements between them has been excluded. Errors for unmatched and nested comments have also been displayed.
Error Handling for Incomplete String: Open and close quote missing, both kind of errors have been handled in the rules written in the script.
Error Handling for Nested Comments: This use-case has been handled by checking for occurrence of multiple successive '/*' or '*/' in the C code, and by omitting the text in between them.
Identification of Main Function: The main function and with arguments of `void` or command line arguments passed to it are identified using the following three regular expressions:

```
(main\(\))
(main\(void\))
(main\(int[ ]+{Identifier}\,[ ]+char\*\*[ ]+{Identifier}\))
```

5. Future Work

The lexical analyser that was created in this project helps in breaking source program into tokens define by the C programming language.

In the next phase, the syntactic analyzer will be designed which will call upon the Flex program to give it tokens and the lexical analyzer will return to the parser the integer value associated with the tokens as and when required by the parser. Together with the symbol table, the parser will prepare a syntax tree with the help of a grammar that we provide it with.

The syntactic analyzer can then logically group the tokens to form meaningful statements and can detect C programming constructs such as arrays, loops, and functions. The syntactic analyzer will also help us identify errors that could not be detected in the lexical analysis phase such as unbalanced parentheses, unterminated statements, missing operators, two operators in a row, etc

6. References

- https://cs.nyu.edu/courses/spring11/G22.2130-001/lecture4.pdf
- https://en.wikipedia.org/wiki/Lexical_analysis
- https://silcnitc.github.io/lex.html
- https://www.d.umn.edu/~rmaclin/cs5641/Notes/L15_SymbolTable.pdf