Lexical Analyzer for the C Language

National Institute of Technology Karnataka,Surathkal

Date: 17th Jan 2019

Submitted To

Dr. P.Santhi Thilagam

Dept. of Computer Science

NITK Surathkal

Group Members:

1. Hardik Rana	(16CO138)
2. Shushant Kumar	(16CO143)
3. Anmol Horo	(16CO206)

Abstract

Aim

Designing a mini C compiler for the analysis phase (frontend of compiler) of compiler. The compiler will be able to perform lexical analysis, parsing, semantic analysis and intermediate code generation for identifiers, constants, escape sequences and certain keywords of C language.

Features

1. **Keywords** - int, char, float, void, long, short, unsigned, for, while, break, continue, if, else, return.
2. **Data types** - int, char, float, void, unsigned int, short, long, long long.

   <data-type><identifier> OR <data-type><identifier> = <value>

   Ex. float a; float a = 0.5;

3. **Arrays** - int, char, float, long, long long

   <data-type><array-name> [size];

   Ex. float a[5];

4. **Punctuators** - ‘[‘, ‘]’, ‘{‘, ‘}’, ‘(‘, ‘)’, ‘=’, ‘,’ , ‘;’ , ‘*’, ‘ “ ” ’
5. **Operators** - +, -, *, /, %, ++, --, !=, &&, ||, >, <, >=, <=, ==, +=, -=, *=, /=, %=
6. **Comments** - // Single line comment, /* Multi line comment */
7. **Loops (with syntax)** - for, while

   Ex. for(initialization; condition; increment) {}

8. **Condition** - if, if else, if else ladder
9. **Function** - simple functions, nested functions

   <return-type><function-name>(<data-type><identifier>){}

10. All the errors will be displayed with the line number.

Contents

1. Introduction

Lexical Analysis:

A compiler is a computer program that transforms source code written in a programming language (the source language) into a machine language (the target language), with the latter often having a binary form known as object code. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

As the first phase of a compiler, the main task of the lexical analyzer being implemented 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. This lexical analyzer maintains a data structure called as the symbol table and constant table. When the lexical analyzer will discover a lexeme constituting an identifier, it enters that lexeme into the symbol table. When it will discover constants then it will be inserted into constant table. The lexical analyzer performs certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace. Another task is correlating error messages generated by the compiler with the source program.

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 scanner 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:

%{ .....   %}

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.

Rules section:

%% ......   %%

The rules section associates regular expression patterns with C statements. When the scanner sees text in the input matching a given pattern, it will execute the associated C code.

C Code section:

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.Lexical analysis only takes care of parsing the tokens and identifying their type. For this reason, we have assumed the C program to be syntactically correct and we generate the stream of tokens as well as the symbol table from it.

2. Design Of Programs

Code:

The entire code for lexical analysis is broken down into 3 files: scanner.l, token_number.h and table.h.

File	Contents
scanner.l	A lex file containing the lex specification of regular expressions.
token_number.h	Contains enumerated constants for keywords, operator, special symbols, constants and identifiers
table.h	Contains the definition of the symbol table and the constants table and also defines functions for inserting into the hash table and displaying its contents.

Lex Code(Scanner.l file)

```
%{

#include <stdio.h>
#include "table.h"
#include "token_number.h"

entry_into_table** stable;
entry_into_table** ctable;
int cmnt_strt = 0;
%}

letter [a-zA-Z]
digit [0-9]
sp  [ \t\r\f\v]+
und [_]
identifier (_|{letter})({letter}|{digit}|_){0,31}
hex [0-9a-f]

 /* Exclusive states */
%x COMMENT
%x PREIN
%x PREDEF

%%
  /* Keywords*/


"main(void)"                     {printf("\t%-10s : %2d- MAINFUNC\n",yytext,MAINFUNC);}
"main()"                         {printf("\t%-10s : %2d- MAINFUNC\n",yytext,MAINFUNC);}
"main(int argc, char **argv)"    {printf("\t%-10s : %2d- MAINFUNC\n",yytext,MAINFUNC);}
"main(int argc, char *argv[])"   {printf("\t%-10s : %2d- MAINFUNC\n",yytext,MAINFUNC);}
"int"                            {printf("\t%-10s : %2d- INT\n",yytext,INT);}
```

```
"char"                          {printf("\t%-10s : %2d- CHAR\n",yytext,CHAR);}
"float"                         {printf("\t%-10s : %2d- FLOAT\n",yytext,FLOAT);}
"void"                          {printf("\t%-10s : %2d- VOID\n",yytext,VOID);}
"long"                          {printf("\t%-10s : %2d- LONG\n",yytext,LONG);}
"long long"                     {printf("\t%-10s : %2d- LONG_LONG\n",yytext,LONG_LONG);}
"short"                         {printf("\t%-10s : %2d- SHORT\n",yytext,SHORT);}
"signed"                        {printf("\t%-10s : %2d- SIGNED\n",yytext,SIGNED);}
"unsigned"                      {printf("\t%-10s : %2d- UNSIGNED\n",yytext,UNSIGNED);}
"for"                           {printf("\t%-10s : %2d- FOR\n",yytext,FOR);}
"while"                         {printf("\t%-10s : %2d- WHILE\n",yytext,WHILE);}
"break"                         {printf("\t%-10s : %2d- BREAK\n",yytext,BREAK);}
"continue"                      {printf("\t%-10s : %2d- CONTINUE\n",yytext,CONTINUE);}
"if"                            {printf("\t%-10s : %2d- IF\n",yytext,IF);}
"else"                          {printf("\t%-10s : %2d- ELSE\n",yytext,ELSE);}
"return"                        {printf("\t%-10s : %2d- RETURN\n",yytext,RETURN);}

{identifier}                    {printf("\t%-10s : %2d- IDENTIFIER\n", yytext,IDENTIFIER);
                                 insert( stable,yytext,IDENTIFIER );}
{sp}                             ;
[+\-]?[0][x|X]{hex}+[lLuU]?               {printf("\t%-10s  :  %2d-  HEXADECIMAL_CONSTANT\n",
yytext,HEX_CONSTANT);
                                insert( ctable,yytext,HEX_CONSTANT);}
[+\-]?{digit}+[lLuU]?       { printf("\t%-10s : %2d- DECIMAL_CONSTANT\n", yytext,DEC_CONSTANT);
                              insert( ctable,yytext,DEC_CONSTANT);}

{letter}({letter}|{digit}|{und})*\[{digit}*\] {printf("\t%-10s : ARRAY\n",yytext);}
"#ifdef"                    {printf("\t%-10s : %2d- IFDEF\n",yytext,IFDEF);}
"#ifndef"                   {printf("\t%-10s : %2d- IFNDEF\n",yytext,IFNDEF);}
"#if"                       {printf("\t%-10s : %2d- IFF\n",yytext,IFF);}
"#else"                     {printf("\t%-10s : %2d- IELSE\n",yytext,IELSE);}
"#elif"                     {printf("\t%-10s : %2d- IELIF\n",yytext,IELIF);}
"#endif"                    {printf("\t%-10s : %2d- IENDIF\n",yytext,IENDIF);}
"#error"                    {printf("\t%-10s : %2d- ERROR\n",yytext,ERROR);}
"#pragma"                   {printf("\t%-10s : %2d- PRAGMA\n",yytext,PRAGMA);}

"/*"                        {cmnt_strt = yylineno; BEGIN COMMENT;}
<COMMENT>.|{sp}                  ;
<COMMENT>\n                     {yylineno++;}
<COMMENT>"*/"                   {BEGIN INITIAL;}
<COMMENT>"/*"                   {printf("Line %2d: Nested comments are not valid!\n",yylineno);}
<COMMENT><<EOF>>                        {printf("Line  %2d:  Unterminated  comment\n",  cmnt_strt);
yyterminate();}
^"#include"                     {BEGIN PREIN;}
<PREIN>"<"[^<>\n]+">"           {printf("\t%-10s : %2d- HEADER_FILE\n",yytext,HEADER_FILE);}
<PREIN>{sp}                     ;
<PREIN>\"[^"\n]+\"              {printf("\t%-10s : %2d- HEADER_FILE\n",yytext,HEADER_FILE);}
<PREIN>\n                       {yylineno++; BEGIN INITIAL;}
<PREIN>.                        {printf("Line %2d: Illegal header file format \n",yylineno);}

^"#define"                  {BEGIN PREDEF;}
<PREDEF>{sp}+{letter}({letter}|{digit}|{und})*{sp}+{digit}+         {printf("\t%-10s         :         %2d-
DEFINE_FILE\n",yytext,DEFINE_FILE);}
<PREDEF>{sp}+{letter}({letter}|{digit}|{und})*{sp}+({digit}+)\.({digit}+)     {printf("\t%-10s    :    %2d-
DEFINE_FILE\n",yytext,DEFINE_FILE);}
<PREDEF>{sp}+{letter}({letter}|{digit}|{und})*{sp}+{letter}({letter}|{digit}|{und})* {printf("\t%-10s :
%2d- DEFINE_FILE\n",yytext,DEFINE_FILE);}
<PREDEF>\n                    {yylineno++; BEGIN INITIAL;}
<PREDEF>.                     {printf("Line %2d: Illegal define file format \n",yylineno);}

"//".*                        ;

\"[^\"\n]*\"     {

  if(yytext[yyleng-2]=='\\') /* check if it was an escaped quote */
  {
    yyless(yyleng-1);       /* push the quote back if it was escaped */
    yymore();
  }
  else {
  insert( ctable,yytext,STRING);
  {printf("\t%-10s : %2d- STRING_CONSTANT\n", yytext,STRING);
  }
 }

\"[^\"\n]*$                   {printf("Line %2d: Unterminated string %s\n",yylineno,yytext);}
{digit}+({letter}|_)+         {printf("Line %2d: Illegal identifier name %s\n",yylineno,yytext);}
\n                                 {yylineno++;}
"--"                               {printf("\t%-10s : %2d- DECREMENT\n",yytext,DECREMENT);}
"++"                               {printf("\t%-10s : %2d- INCREMENT\n",yytext,INCREMENT);}
"->"                               {printf("\t%-10s : %2d- PTR_SELECT\n",yytext,PTR_SELECT);}
"&&"                               {printf("\t%-10s : %2d- LOGICAL_AND\n",yytext,LOGICAL_AND);}
"||"                               {printf("\t%-10s : %2d- LOGICAL_OR\n",yytext,LOGICAL_OR);}
"<="                               {printf("\t%-10s : %2d- LS_THAN_EQ\n",yytext,LS_THAN_EQ);}
">="                               {printf("\t%-10s : %2d- GR_THAN_EQ\n",yytext,GR_THAN_EQ);}
"=="                               {printf("\t%-10s : %2d- EQUAL_TO\n",yytext,EQ);}
"!="                               {printf("\t%-10s : %2d- NOT_EQUAL_TO\n",yytext,NOT_EQ);}
";"                                {printf("\t%-10s : %2d- DELIMITER\n",yytext,DELIMITER);}
"{"                                {printf("\t%-10s : %2d- OPEN_BRACES\n",yytext,OPEN_BRACES);}
"}"                                {printf("\t%-10s : %2d- CLOSE_BRACES\n",yytext,CLOSE_BRACES);}
","                                {printf("\t%-10s : %2d- COMMA\n",yytext,COMMA);}
"="                                                        {printf("\t%-10s   :   %2d-
ASSIGNMENT_OPERATOR\n",yytext,ASSIGN);}
"("                                                         {printf("\t%-10s   :   %2d-
OPEN_PARENTHESIS\n",yytext,OPEN_PAR);}
")"                                 {printf("\t%-10s : %2d- CLOSE_PAR\n",yytext,CLOSE_PAR);}
"["                  {printf("\t%-10s : %2d- OPEN_SQ_BRKT\n",yytext,OPEN_SQ_BRKT);}
"]"                  {printf("\t%-10s : %2d- CLOSE_SQ_BRKT\n",yytext,CLOSE_SQ_BRKT);}
"-"                                 {printf("\t%-10s : %2d- MINUS\n",yytext,MINUS);}
"+"                                 {printf("\t%-10s : %2d- PLUS\n",yytext,PLUS);}
"*"                                 {printf("\t%-10s : %2d- STAR\n",yytext,STAR);}
"/"                                 {printf("\t%-10s : %2d- DIV\n",yytext,FW_SLASH);}
"%"                                 {printf("\t%-10s : %2d- MOD\n",yytext,MODULO);}
"<"                                 {printf("\t%-10s : %2d- LS_THAN\n",yytext,LS_THAN);}
">"                                 {printf("\t%-10s : %2d- GR_THAN\n",yytext,GR_THAN);}
"+="                                {printf("\t%-10s : %2d- ADD_ASSIGN\n",yytext,ADDASS);}
"-="                                {printf("\t%-10s : %2d- SUB_ASSIGN\n",yytext,SUBASS);}
```

```
"*="                                        {printf("\t%-10s : %2d- MUL_ASSIGN\n",yytext,MULASS);}
"/="                                        {printf("\t%-10s : %2d- DIV_ASSIGN\n",yytext,DIVASS);}
"%="                                        {printf("\t%-10s : %2d- MOD_ASSIGN\n",yytext,MODASS);}
.                                           {printf("Line %2d: Illegal character %s\n",yylineno,yytext);}


%%

int main()
{
  stable=table_create();
  ctable=table_create();
  yyin=fopen("testcases/test-case-6.c","r");
  yylex();
  printf("\n\tSymbol table");
  display(stable);
  printf("\n\tConstant Table");
  display(ctable);
}
```

table.h file

```
/*
 File : table .h
 Description : This file contains functions related to a hash organised
                        symbol table.
 The functions implemented are:
  create_table (), insert (), search , display ()
*/

#include <stdio.h>

#define TABLE_SIZE 500

/* struct to hold each entry */
struct entry_into_state
{
            char* lexeme;
            int token_name;
            struct entry_into_state* successor;
};

typedef struct entry_into_state entry_into_table;


// Traverse the hash table and print all the entries
void display(entry_into_table** hash_table_ptr)
{
            int i;
```

```
        entry_into_table* traverser;
   printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
   printf("\t < lexeme , token >\n");
   printf("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");

        for( i=0; i < TABLE_SIZE; i++)
        {
                traverser = hash_table_ptr[i];

                while( traverser != NULL)
                {
                        printf("<        %-20s,        %3d        >\n",        traverser->lexeme,
traverser->token_name);
                        traverser = traverser->successor;
                }
        }

}


/* Create a new hash_table. */
entry_into_table** table_create()
{
        entry_into_table** hash_table_ptr = NULL; // declare a pointer

        /* Allocate memroy for a hashtable array of size TABLE_SIZE */
        if( ( hash_table_ptr = ( entry_into_table** ) malloc( sizeof( entry_into_table* ) * TABLE_SIZE )
) == NULL )
        return NULL;

        int i;

        // Intitialise all entries as NULL
       for( i = 0; i < TABLE_SIZE; i++ )
        {
                hash_table_ptr[i] = NULL;
        }

        return hash_table_ptr;
}

/* Generate hash from a string. Then generate an index in [0, TABLE_SIZE) */
uint32_t hash( char *lexeme )
{
        size_t i;
        uint32_t hash;

        /* Apply jenkin's hash function
        * https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time
        */
        for ( hash = i = 0; i < strlen(lexeme); ++i ) {
      hash += lexeme[i];
      hash += ( hash << 10 );
```

```
        hash ^= ( hash >> 6 );
        }
          hash += ( hash << 3 );
          hash ^= ( hash >> 11 );
         hash += ( hash << 15 );

          return hash % TABLE_SIZE; // return an index in [0, TABLE_SIZE)
}

/* Create an entry for a lexeme, token pair. This will be called from the insert function */
entry_into_table *create( char *lexeme, int token_name )
{
          entry_into_table *newentry;

          /* Allocate space for newentry */
          if( ( newentry = ( entry_into_table* )malloc( sizeof( entry_into_table ) ) ) == NULL ) {
                    return NULL;
          }
          /* Copy lexeme to newentry location using strdup (string-duplicate). Return NULL if it fails
*/
          if( ( newentry->lexeme = strdup( lexeme ) ) == NULL ) {
                    return NULL;
          }

          newentry->token_name = token_name;
          newentry->successor = NULL;

          return newentry;
}

/* Search for an entry given a lexeme. Return a pointer to the entry of the lexeme exists, else
return NULL */
entry_into_table* search( entry_into_table** hash_table_ptr, char* lexeme )
{
          uint32_t idx = 0;
          entry_into_table* myentry;

        // get the index of this lexeme as per the hash function
          idx = hash( lexeme );

          /* Traverse the linked list at this idx and see if lexeme exists */
          myentry = hash_table_ptr[idx];

          while( myentry != NULL && strcmp( lexeme, myentry->lexeme ) != 0 )
          {
                    myentry = myentry->successor;
          }

          if(myentry == NULL) // lexeme is not found
                    return NULL;

          else // lexeme found
                    return myentry;
```

```
}

/* Insert an entry into a hash table. */
void insert( entry_into_table** ptr, char* lexeme, int token_name )
{
        if( search( ptr, lexeme ) != NULL) // If lexeme already exists int the table, then don't insert
           return;

        uint32_t idx;
        entry_into_table* newentry = NULL;
        entry_into_table* head = NULL;

        idx = hash( lexeme ); // Retrieving the index for this lexeme using the hash function
        newentry = create( lexeme, token_name ); // Create new entry using the <lexeme, token>
pair

        if(newentry == NULL) // handling any error due to insufficient memory or other errors
        {
                printf("Insert failed. New entry could not be created.");
                exit(1);
        }

        head = ptr[idx]; // finding head entry at idx

        if(head == NULL) // The first lexeme that matches the hash index value
        {
                ptr[idx] = newentry;
        }
        else // normal entry in the table
        {
                newentry->successor = ptr[idx];
                ptr[idx] = newentry;
        }
}
```

token_number.h file

```
/*
 File : token_number.h
 Description : This file defines tokens and the values associated to them.
*/

enum IDENTIFIER
{
  IDENTIFIER=500
};
```

```
enum special_symbols
{
  DELIMITER=300,
  OPEN_BRACES,
  CLOSE_BRACES,
  COMMA,
  OPEN_PAR,
  CLOSE_PAR,
  OPEN_SQ_BRKT,
  CLOSE_SQ_BRKT,
  FW_SLASH,
  MAINFUNC
};

enum keywords
{
  INT=100,
  CHAR,
  FLOAT,
  VOID,
  LONG,
  LONG_LONG,
  SHORT,
  SIGNED,
  UNSIGNED,
  FOR,
  WHILE,
  BREAK,
  CONTINUE,
  RETURN,
  IF,
  ELSE,
  IFDEF,
  IFNDEF,
  IFF,
  IELSE,
  IELIF,
  IENDIF,
  ERROR,
  PRAGMA


};

enum constants
{
  HEX_CONSTANT=400,
  DEC_CONSTANT,
  HEADER_FILE,
  DEFINE_FILE,
  STRING
};

enum operators
{
  DECREMENT=200,
  INCREMENT,
  PTR_SELECT,
  LOGICAL_AND,
  LOGICAL_OR,
  LS_THAN_EQ,
  GR_THAN_EQ,
  EQ,
  NOT_EQ,
  ASSIGN,
  MINUS,
  PLUS,
  STAR,
  MODULO,
  LS_THAN,
  GR_THAN,
  ADDASS,
  SUBASS,
  MULASS,
  DIVASS,
  MODASS
};
```

Explanation:

Comments:

Single and multi line comments are identified. Single line comments are identified by //.* regular expression. The multiline regex is identified as follows:

- We make use of an exclusive state called **<COMMENT>** . When a /* pattern is found, we note down the line number and enter the **<COMMENT>** exclusive state. When the */ is found, we go back to the INITIAL state, the default state in Flex, signifying the end of the comment.
- Then we define patterns that are to be matched only when the scanner is in the **<COMMENT>** state. Since, it is an exclusive state, only the patterns that are defined for this state (the ones prepended with **<COMMENT>** in the lex file are matched, rest of the patterns are inactive.
- We also identify nested comments. If we find another /* while still in the **<COMMENT>** state, we print an error message saying that nested comments are invalid.
- If the comment does not terminate until EOF , and error message is displayed along with the line number where the comment begins. This is implemented by checking if the scanner matches **<<EOF>>** pattern while still in the **<COMMENT>** state, which

means that a */ has not been found until the end of file and therefore the comment has not been terminated.

Preprocessor Directives:

The filenames that come after the #include are selectively identified through the exclusive state **<PREIN>** since the regular expressions for treating the filenames must be kept different from other regexes.

Upon encountering a #include at the beginning of a line, the scanner switches to the state **<PREIN>** where it can tokenize filenames of the form "stdio.h" or <stdio.h> .Filenames of any other format are considered as illegal and an error message regarding the same is printed.

Integer Constants:

- The Flex program can identify two types of numeric constants: decimal and hexadecimal. The regular expressions for these are [+-]?{digit}+[lLuU]? and [+-]?0[xX]{hex}+[lLuU]? Respectively.
- The sign is considered as optional and a constant without a sign is by default positive. All hexadecimal constants should begin with 0x or 0X .
- The definition of {digit} is all the decimal digits 0-9 . The definition of {hex} consists of the hexadecimal digits 0-9 and characters a-f .
- Some constants which are assigned to long or unsigned variables may suffix l or L and u or U or a combination of these characters with the constant. All of these conditions are taken care of by the regular expression.

Keywords:

The keywords identified are: int, char, float, long, short, long long, void, signed, unsigned, for, break, continue, if, else, return.

Identifiers:

- Identifiers are identified and added to the symbol table. The rule followed is represented by the regular expression (_|{letter})({letter}|{digit}|_){0,31} .
- The rule only identifies those lexemes as identifiers which either begin with a letter or an underscore and is followed by either a letter, digit or an underscore with a maximum length of 32.
- The first part of the regular expression (_|{letter}) ensures that the identifiers begin with an underscore or a letter and the second part ({letter}|{digit}|_){0,31} matches a combination of letters, digits and underscore and ensures that the maximum length does not exceed 32. The definitions of {letter} and {digit} can be seen in the code at the end.

- Any identifier that begins with a digit is marked as a lexical error and the same is displayed on the output. The regex used for this is

  { digit}+({letter}|_)+

Strings:

The scanner can identify strings in any C program. It can also handle double quotes that are escaped using a \ inside a string. Further, error messages are displayed for unterminated strings. We use the following strategy.

- We first match patterns that are within double quotes.
- But if the string is something like “This is \” a string”, it will only match “This is \” . So as soon as a match is found we first check if the last double quote is escaped using a backslash.
- If the last quote is not escaped with a backslash we have found the string we are looking for and we add it to the constants table.
- But in case the last double quote is escaped with a backslash we push the last double quote back for scanning. This can be achieved in lex using the command yyless(yyleng - 1).
- yyless(n) [1] tells lex to “push back” all but the first n characters of the matched token. yyleng [1] holds the length of the matched token.
- And hence yyless(yyleng -1) will push back the last character i.e the double quote back for scanning and lex will continue scanning from “ is a string” .
- We use another built-in lex function called yymore() [1] which tells lex to append the next matched token to the currently matched one.
- Now the scanner continues and matches “is a string” and since we had called yymore() earlier it appends it to the earlier token “This is \ giving us the entire string “This is \” a string” . Notice that since we had called yyless(yyleng - 1) the last double quote is left out from the first matched token giving us the entire string as required.
- The following lines of code accomplish the above described process.

```
\" [^ \"\n ]* \" {

            if ( yytext [ yyleng - 2 ]== '\\' )          /* check if it was an escaped quote */
            {
            yyless ( yyleng - 1 );                    /* push the quote back if it was escaped */
            yymore ();                                     /* Append next token to this one */
            }
            else
            {
            insert ( constant_table , yytext , STRING );
            }
}
```

- We use the regular expression \ "[^\"\n]*$ to check for strings that don’t terminate. This regular expression checks for a sequence of a double quote followed by zero or more occurrences of characters excluding double quotes and new line and this sequence should not have a close quote. This is specified by the $ character which tests for the end of line. Thus, the regular expression checks for strings that do not terminate till end of line and it prints an error message on the screen.

Symbol Table & Constants table:

We implement a generic hash table with chaining than can be used to declare both a symbol table and a constant table. Every entry in the hash table is a struct of the following form.

```
/* struct to hold each entry */
struct entry_into_state
{
        char* lexeme;
        int token_name;
        struct entry_into_state* successor;
};

typedef struct entry_into_state entry_into_table;
```

The struct consists of a character pointer to the lexeme that is matched by the scanner, an integer token that is associated with each type of token as defined in “ tokens.h ” and a pointer to the next node in the case of chaining in the hash table.

A symbol table or a constant table can be created using the table_create() function. The function returns a pointer to a new created hash table which is basically an array of pointers of the type entry_into_table* . This is achieved by the following lines:

```
/* declare pointers and assign hash tables */
entry_into_table** stable;
entry_into_table** ctable;
stable=table_create();
ctable=table_create();
```

Every time the scanner matches a pattern, the text that matches the pattern (lexeme) is entered into the associated hash table using an insert() function. There are two hash tables maintained: **the symbol table and the constants table**. Depending on whether the lexeme is a constant or a symbol, an appropriate parameter is passed to the insert function.

For example, insert(stable, yytext, INT) inserts the keyword INT into the symbol table and insert(ctable, yytext, HEX_CONSTANT) inserts a hexadecimal constant into the constants table. The values associated with INT, HEX_CONSTANT and other tokens are defined in the tokens.h file.

A hash is generated using the matched pattern string as input. We use the Jenkins hash function. The hash table has a fixed size as defined by the user using TABLE_SIZE . The generated hash value is mapped to a value in the range [0, TABLE_SIZE) through the operation hash_value % TABLE_SIZE . This is the index in the hash table for this particular entry. In case the indices clash, a linked list is created and the multiple clashing entries are chained together at that index.

3. Test Cases

Without Errors:

S.No	Test Case	Expected Output	Status
1	int a=100;	int : 100- INT a : 500- IDENTIFIER = : 209- ASSIGNMENT_OPERATOR 100 : 401- DECIMAL_CONSTANT ; : 300- DELIMITER	PASS
2	float a[10];	float : 102- FLOAT a[10] : ARRAY ; : 300- DELIMITER	PASS
3	unsigned int a = 0x0f;	unsigned : 108- UNSIGNED int : 100- INT a : 500- IDENTIFIER = : 209- ASSIGNMENT_OPERATOR 0x0f : 400- HEXADECIMAL_CONSTANT ; : 300- DELIMITER	PASS
4	total = x + y;	total : 500- IDENTIFIER = : 209- ASSIGNMENT_OPERATOR x : 500- IDENTIFIER + : 211- PLUS y : 500- IDENTIFIER ; : 300- DELIMITER	PASS
5	printf("This is a string");	printf : 500- IDENTIFIER (: 304- OPEN_PARENTHESIS) : 305- CLOSE_PAR ; : 300- DELIMITER	PASS
6	Str = “ \” hello \” ”	Str : 500- IDENTIFIER = : 209- ASSIGNMENT_OPERATOR	PASS

With Errors:

S.No	**Test Case**	**Expected Output**	**Status**
1	/* To make things /* nested multi-line comment */ interval	Nested comments are not valid!	PASS

	*/		
2	str= “star	str : 500- IDENTIFIER = : 209- ASSIGNMENT_OPERATOR Unterminated string "star	PASS
3	#include<<stdio.h>	Illegal header file format	PASS
4	#inc	Illegal character #	PASS
5	#include ""wrong.h"	Illegal header file format	PASS
6	9f = d @ j ;	Line 1: Illegal identifier name 9f = : 209- ASSIGNMENT_OPERATOR d : 500- IDENTIFIER Line 1: Illegal character @ j : 500- IDENTIFIER ; : 300- DELIMITER	PASS
7	int atgd$;	int : 100- INT atgd : 500- IDENTIFIER Line 1: Illegal character $; : 300- DELIMITER	PASS

Test Case 1:

```
/*
-Identification of preprocessor directive
-identification of Single and Multi-line Comments
Following errors must be detected
 - Invalid nested Multi-line comments
*/
#include<stdio.h>
void main(){
   // Single line comment
      /* Multi-line comment
      this */
      /* here */ int a; /* "int a" should be untouched */
// This nested comment // This comment should be removed should be removed
/* To make things /*  nested multi-line comment */ interval */
          return 0;
}
```

Output 1:

```
            <stdio.h>  : 402- HEADER_FILE
            void       : 103- VOID
            main()     : 309- MAINFUNC
            {          : 301- OPEN_BRACES
            int        : 100- INT
            a          : 500- IDENTIFIER
            ;          : 300- DELIMITER
Line 17: Nested comments are not valid!
            interval   : 500- IDENTIFIER
            *          : 212- STAR
            /          : 308- DIV
            return     : 113- RETURN
            0          : 401- DECIMAL_CONSTANT
            ;          : 300- DELIMITER
            }          : 302- CLOSE_BRACES

            Symbol table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< a                   , 500 >
< interval            , 500 >

            Constant Table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< 0                   , 401 >
```

Test Case 2:

```
/*
-Identification of preprocessor directive
-identification of Single and Multi-line Comments
Following errors must be detected
 - Invalid Multi-line comments
*/
#include<stdio.h>
void main()
{
        // This is correct
        /* This as correct
        too */
        /* This is not correct since
        this comment is not ending
        return 0;
}
```

Output 2 :

```
       <stdio.h>  : 402- HEADER_FILE
        void      : 103- VOID
        main()    : 309- MAINFUNC
        {         : 301- OPEN_BRACES
Line 11: Unterminated comment

        Symbol table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
         < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


        Constant Table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
         < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Test Case 3:

```
/*
-Identification of preprocessor directive
-identification of String literals
Following errors must be detected
 - Invalid preprocessor directive
 - Invalid strings
*/
#include<stdio.h>
#include <<stdlib.h>
#include "correct.h"
#include ""wrong.h"
void main()
{
        printf("This string terminates");
        printf("This string does not terminate);
}
```

Output 3 :

```
<stdio.h>  : 402- HEADER_FILE
Line  4: Illegal header file format
        <stdlib.h> : 402- HEADER_FILE
        "correct.h" : 402- HEADER_FILE
Line  6: Illegal header file format
        "wrong.h"  : 402- HEADER_FILE
        void      : 103- VOID
        main()     : 309- MAINFUNC
        {         : 301- OPEN_BRACES
        printf     : 500- IDENTIFIER
        (         : 304- OPEN_PARENTHESIS
        )         : 305- CLOSE_PAR
        ;         : 300- DELIMITER
        printf     : 500- IDENTIFIER
        (         : 304- OPEN_PARENTHESIS
Line 11: Unterminated string "This string does not terminate);
        }       : 302- CLOSE_BRACES

        Symbol table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
         < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< printf          , 500 >

        Constant Table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
         < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< "This string terminates", 404 >
```

Test Case 4:

```
/*
Following errors must be detected
 - Invalid identifiers: 9s, total$
 - Invalid operator: #
 - Escaped quoted should be part of the string that is identified
 - Stray characters: `, #, -

The output should display appropriate errors
*/
#include<stdio.h>
#include<stdlib.h>

int main()
{
  `
 # -
 short int b;
```

```
  int x, 6s, agg$;
  agg = x # y;
  printf ("Next = %d \n \" ", agg);
}
```

Output 4 :

```
        <stdio.h>  : 402- HEADER_FILE
         <stdlib.h> : 402- HEADER_FILE
         int      : 100- INT
       main()     : 309- MAINFUNC
         {        : 301- OPEN_BRACES
Line 17: Illegal character `
Line 18: Illegal character #
         -        : 210- MINUS
         short      : 106- SHORT
         int      : 100- INT
         b        : 500- IDENTIFIER
         ;        : 300- DELIMITER
         int      : 100- INT
         x        : 500- IDENTIFIER
         ,        : 303- COMMA
Line 20: Illegal identifier name 6s
         ,        : 303- COMMA
         agg       : 500- IDENTIFIER
Line 20: Illegal character $
         ;        : 300- DELIMITER
         agg       : 500- IDENTIFIER
         =        : 209- ASSIGNMENT_OPERATOR
         x        : 500- IDENTIFIER
Line 21: Illegal character #
         y        : 500- IDENTIFIER
         ;        : 300- DELIMITER
         printf     : 500- IDENTIFIER
         (        : 304- OPEN_PARENTHESIS
          ,        : 303- COMMA
        agg       : 500- IDENTIFIER
          )        : 305- CLOSE_PAR
          ;        : 300- DELIMITER
          }        : 302- CLOSE_BRACES

          Symbol table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
           < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< b                , 500 >
< y                , 500 >
< x                , 500 >
< agg              , 500 >
< printf           , 500 >

          Constant Table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
           < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< "Next = %d \n \" "  , 404 >
```

Test Case 5:

```
/*
Identifying tokens and displaying symbol and constants table


Following tokens must be detected
 - Keywords   (int, char, float, void, long int, long long int, main include, signed, unsigned)
 - Identifiers (main,total,x,y,printf),
 - Constants  (-10, 20, 0x0f, 123456l)
 - Strings ("Total = %d \n")
 - Special symbols and Brackets ( (), {}, ;, ,)
 - Operators (+,-,=,*,/,%,--,++)

The output should display appropriate tokens with their type and also the symbol and constants table
*/
#include<stdio.h>
#include<stdlib.h>

void print(){
  printf("Hello World!\n");
}

int main()
{
  int x, y;
  char k;
  long long int total, diff;
  int *ptr;
  unsigned int a = 0x0f;
  long int mylong = 123456l;
  long int i, j;
  for(i=0; i < 10; i++){
    for(j=10; j > 0; j--){
    printf("%d",i);
    }
  }
  if(a == 0x0f){
    printf("Correct\n");
  }
  else {
    printf("Wrong\n");
  }
  x = -10, y = 20;
  x=x*3/2;
  total = x + y;
  diff = x - y;
  int rem = x % y;
  printf ("Total = %d \n", total);
  return 0;
}
```

Output 5 :

```
     <stdio.h>  : 402- HEADER_FILE
      <stdlib.h> : 402- HEADER_FILE
      void      : 103- VOID
      print     : 500- IDENTIFIER
      (         : 304- OPEN_PARENTHESIS
      )         : 305- CLOSE_PAR
      {         : 301- OPEN_BRACES
      printf    : 500- IDENTIFIER
      (         : 304- OPEN_PARENTHESIS
      )         : 305- CLOSE_PAR
      ;         : 300- DELIMITER
    }         : 302- CLOSE_BRACES
      int       : 100- INT
      main()    : 309- MAINFUNC
      {         : 301- OPEN_BRACES
      int       : 100- INT
      x         : 500- IDENTIFIER
      ,         : 303- COMMA
      y         : 500- IDENTIFIER
      ;         : 300- DELIMITER
      char      : 101- CHAR
      k         : 500- IDENTIFIER
      ;         : 300- DELIMITER
      long long  : 105- LONG_LONG
      int       : 100- INT
      total     : 500- IDENTIFIER
      ,         : 303- COMMA
```

```
diff      : 500- IDENTIFIER
;        : 300- DELIMITER
int      : 100- INT
*         : 212- STAR
ptr      : 500- IDENTIFIER
;        : 300- DELIMITER
unsigned   : 108- UNSIGNED
int      : 100- INT
a         : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
0x0f      : 400- HEXADECIMAL_CONSTANT
;        : 300- DELIMITER
long      : 104- LONG
int      : 100- INT
mylong     : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
123456l    : 401- DECIMAL_CONSTANT
;        : 300- DELIMITER
long      : 104- LONG
int      : 100- INT
i        : 500- IDENTIFIER
,        : 303- COMMA
j        : 500- IDENTIFIER
;        : 300- DELIMITER
for      : 109- FOR
(        : 304- OPEN_PARENTHESIS
i        : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
0         : 401- DECIMAL_CONSTANT
;        : 300- DELIMITER
i        : 500- IDENTIFIER
<         : 214- LS_THAN
10        : 401- DECIMAL_CONSTANT
;        : 300- DELIMITER
i        : 500- IDENTIFIER
```

```
++        : 201- INCREMENT
)         : 305- CLOSE_PAR
{         : 301- OPEN_BRACES
for       : 109- FOR
(         : 304- OPEN_PARENTHESIS
j         : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
10        : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
j         : 500- IDENTIFIER
>         : 215- GR_THAN
0         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
j         : 500- IDENTIFIER
--        : 200- DECREMENT
)         : 305- CLOSE_PAR
{         : 301- OPEN_BRACES
printf    : 500- IDENTIFIER
(         : 304- OPEN_PARENTHESIS
,         : 303- COMMA
i         : 500- IDENTIFIER
)         : 305- CLOSE_PAR
;         : 300- DELIMITER
}         : 302- CLOSE_BRACES
}         : 302- CLOSE_BRACES
if        : 114- IF
(         : 304- OPEN_PARENTHESIS
a         : 500- IDENTIFIER
==        : 207- EQUAL_TO
0x0f      : 400- HEXADECIMAL_CONSTANT
)         : 305- CLOSE_PAR
{         : 301- OPEN_BRACES
printf    : 500- IDENTIFIER
(         : 304- OPEN_PARENTHESIS
)         : 305- CLOSE_PAR
```

```
;         : 300- DELIMITER
}         : 302- CLOSE_BRACES
else      : 115- ELSE
{         : 301- OPEN_BRACES
printf    : 500- IDENTIFIER
(         : 304- OPEN_PARENTHESIS
)         : 305- CLOSE_PAR
;         : 300- DELIMITER
}         : 302- CLOSE_BRACES
x         : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
-10       : 401- DECIMAL_CONSTANT
,         : 303- COMMA
y         : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
20        : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
x         : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
x         : 500- IDENTIFIER
*         : 212- STAR
3         : 401- DECIMAL_CONSTANT
/         : 308- DIV
2         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
total     : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
x         : 500- IDENTIFIER
+         : 211- PLUS
y         : 500- IDENTIFIER
;         : 300- DELIMITER
diff      : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
x         : 500- IDENTIFIER
-         : 210- MINUS
```

```
             y          : 500- IDENTIFIER
             ;          : 300- DELIMITER
             int        : 100- INT
             rem        : 500- IDENTIFIER
             =          : 209- ASSIGNMENT_OPERATOR
             x          : 500- IDENTIFIER
             %          : 213- MOD
             y          : 500- IDENTIFIER
             ;          : 300- DELIMITER
             printf     : 500- IDENTIFIER
             (          : 304- OPEN_PARENTHESIS
             ,          : 303- COMMA
             total      : 500- IDENTIFIER
             )          : 305- CLOSE_PAR
             ;          : 300- DELIMITER
             return     : 113- RETURN
             0          : 401- DECIMAL_CONSTANT
             ;          : 300- DELIMITER
             }          : 302- CLOSE_BRACES

             Symbol table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
              < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< total              , 500 >
< ptr                , 500 >
< k                  , 500 >
< a                  , 500 >
< rem                , 500 >
< y                  , 500 >
< diff               , 500 >
< j                  , 500 >
< x                  , 500 >
< mylong             , 500 >
< printf             , 500 >
< i                 , 500 >
< print             , 500 >


          Constant Table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
           < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< 123456l           , 401 >
< "Total = %d \n"   , 404 >
< 0                 , 401 >
< 3                 , 401 >
< "Hello World!\n"  , 404 >
< -10               , 401 >
< "Correct\n"       , 404 >
< "Wrong\n"         , 404 >
< 10                , 401 >
< 20                , 401 >
< "%d"              , 404 >
< 0x0f              , 400 >
< 2                 , 401 >
```

Test Case 6:

```
/* Test Case 6 - Binary Search  */
#include <stdio.h>

int main()
{
  int c, first, last, middle, n, search;
  int array[10]={0,1,2,3,4,5,6,7,8,9};

  n = 10;
  search = 3;

  first = 0;
  last = n - 1;
  middle = (first+last)/2;

  while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d isn't present in the list.\n", search);

  return 0;
}
```

Output 6 :

```
          <stdio.h>  : 402- HEADER_FILE
          int        : 100- INT
          main()     : 309- MAINFUNC
          {          : 301- OPEN_BRACES
          int        : 100- INT
          c          : 500- IDENTIFIER
          ,          : 303- COMMA
          first      : 500- IDENTIFIER
          ,          : 303- COMMA
          last       : 500- IDENTIFIER
          ,          : 303- COMMA
          middle     : 500- IDENTIFIER
```

```
,        : 303- COMMA
n         : 500- IDENTIFIER
,        : 303- COMMA
search     : 500- IDENTIFIER
;        : 300- DELIMITER
int       : 100- INT
array[10]  : ARRAY
=         : 209- ASSIGNMENT_OPERATOR
{        : 301- OPEN_BRACES
0         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
1         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
2         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
3         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
4         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
5         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
6         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
7         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
8         : 401- DECIMAL_CONSTANT
,        : 303- COMMA
9         : 401- DECIMAL_CONSTANT
}        : 302- CLOSE_BRACES
;        : 300- DELIMITER
n         : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
10        : 401- DECIMAL_CONSTANT
;        : 300- DELIMITER
search     : 500- IDENTIFIER
```

```
=         : 209- ASSIGNMENT_OPERATOR
3         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
first     : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
0         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
last      : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
n         : 500- IDENTIFIER
-         : 210- MINUS
1         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
middle    : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
(         : 304- OPEN_PARENTHESIS
first     : 500- IDENTIFIER
+         : 211- PLUS
last      : 500- IDENTIFIER
)         : 305- CLOSE_PAR
/         : 308- DIV
2         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
while     : 110- WHILE
(         : 304- OPEN_PARENTHESIS
first     : 500- IDENTIFIER
<=        : 205- LS_THAN_EQ
last      : 500- IDENTIFIER
)         : 305- CLOSE_PAR
{         : 301- OPEN_BRACES
if        : 114- IF
(         : 304- OPEN_PARENTHESIS
array     : 500- IDENTIFIER
[         : 306- OPEN_SQ_BRKT
middle    : 500- IDENTIFIER
]         : 307- CLOSE_SQ_BRKT
<          : 214- LS_THAN
search     : 500- IDENTIFIER
)         : 305- CLOSE_PAR
first      : 500- IDENTIFIER
=          : 209- ASSIGNMENT_OPERATOR
middle     : 500- IDENTIFIER
+          : 211- PLUS
1          : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
else       : 115- ELSE
if        : 114- IF
(         : 304- OPEN_PARENTHESIS
array      : 500- IDENTIFIER
[         : 306- OPEN_SQ_BRKT
middle     : 500- IDENTIFIER
]         : 307- CLOSE_SQ_BRKT
==         : 207- EQUAL_TO
search     : 500- IDENTIFIER
)         : 305- CLOSE_PAR
{          : 301- OPEN_BRACES
printf     : 500- IDENTIFIER
(         : 304- OPEN_PARENTHESIS
,         : 303- COMMA
search     : 500- IDENTIFIER
,         : 303- COMMA
middle     : 500- IDENTIFIER
+1         : 401- DECIMAL_CONSTANT
)         : 305- CLOSE_PAR
;         : 300- DELIMITER
break      : 111- BREAK
;         : 300- DELIMITER
}          : 302- CLOSE_BRACES
else       : 115- ELSE
last       : 500- IDENTIFIER
```

```
=         : 209- ASSIGNMENT_OPERATOR
middle    : 500- IDENTIFIER
-         : 210- MINUS
1         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
middle    : 500- IDENTIFIER
=         : 209- ASSIGNMENT_OPERATOR
(         : 304- OPEN_PARENTHESIS
first     : 500- IDENTIFIER
+         : 211- PLUS
last      : 500- IDENTIFIER
)         : 305- CLOSE_PAR
/         : 308- DIV
2         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
}         : 302- CLOSE_BRACES
if        : 114- IF
(         : 304- OPEN_PARENTHESIS
first     : 500- IDENTIFIER
>         : 215- GR_THAN
last      : 500- IDENTIFIER
)         : 305- CLOSE_PAR
printf    : 500- IDENTIFIER
(         : 304- OPEN_PARENTHESIS
,         : 303- COMMA
search    : 500- IDENTIFIER
)         : 305- CLOSE_PAR
;         : 300- DELIMITER
return    : 113- RETURN
0         : 401- DECIMAL_CONSTANT
;         : 300- DELIMITER
}         : 302- CLOSE_BRACES

          Symbol table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
          < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< c              , 500 >
< last           , 500 >
< search         , 500 >
< first          , 500 >
< n              , 500 >
< array          , 500 >
< printf         , 500 >
< middle         , 500 >

         Constant Table
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
          < lexeme , token >
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
< 8              , 401 >
< 0              , 401 >
< "%d found at location %d.\n", 404 >
< 3              , 401 >
< 9              , 401 >
< 5              , 401 >
< 6              , 401 >
< 4              , 401 >
< 7              , 401 >
< +1             , 401 >
< 10             , 401 >
< "Not found! %d isn't present in the list.\n", 404 >
< 1              , 401 >
< 2              , 401 >
```

4. 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 scanner must correctly recognize all valid identifiers in C, including the ones having one or more underscores.

- **Multiline comments are supported**: 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.

- **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 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.

At the end of the token recognition, the scanner prints a list of all the tokens present in the program. We use the following technique to implement this:

- We have assigned unique integers to all different kinds of tokens present in the C code.

- For storing these tokens and their attributes in the symbol table, we have defined a structure.

```
struct entry_into_state
{
        char* lexeme;
        int token_name;
        struct entry_into_state* successor;
};
```

Where, lexeme stores the name of the token, token_name stores the id of token and successor points to the next token in the linked list.

As and when successive tokens are encountered, their respective values are stored in the structure and then later displayed. We also have functionalities for checking and accordingly omitting duplicate entries in the symbol table.

- In the end, each token is printed along with its type and line number.

- Errors like unmatched multi line comment, nested multi line comments, incomplete strings and unmatched parenthesis are also displayed along with their line numbers.

- The symbol table is displayed, having columns Serial Number, Token and attribute.

5.Results and Future Works:

Results:

Tokens are identified. The keywords mentioned in the abstract are identified. The various literals- integer, float and strings etc are being identified. Valid comments are being ignored. Invalid Comments are pointed out with line numbers. Valid preprocessor directives are identified while invalid ones are pointed out. Array variables are identified. Different Operators, punctuation, data-types are identified.

1. Tokens are displayed in the manner given below:

Token ---- Token Type

2. Symbol Table is displayed in the manner given below :

Token ---- Attribute

3. Constant table displayed in the manner given below :

Token ---- Attribute

Future Works:

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. Some of the remaining keywords and data structures will be incorporated in the future stages of code.

6.References

1. Lex and Yacc By John R. Levine, Tony Mason, Doug Brown

2. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

3. https://en.wikipedia.org/wiki/Symbol_table

4.https://www.cse.iitb.ac.in/~br/courses/cs699-autumn2013/refs/lextut-victor-eijkhout.pdf

5.https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/tutorial-large.pdf

6.http://dinosaur.compilertools.net/lex/

7.Jenkins Hash Function on Wikipedia :

https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time