Lexical Analyzer for the C Language

National Institute of Technology Karnataka, Surathkal

Date: 16th January 2019

Submitted To: Ms Uma Priya

Group Members:
Mahir Jain - 16CO123
Suraj Singh - 16CO146
Darshan DV - 16CO216

Abstract:

Features to be Implemented:

- **Data Types**
  - int a,b; (Declaration)
  - char c = ‘x’ (Initialization)
  - Similarly short, long, unsigned, signed
- **Arrays**: Example- int a[5];
- **Punctuators**
  - #, <> , [ ] , { } , ; , . ,
- **Operators**
  - ***Arithmetic Operators*** (+, -, *, /, %, post-increment, pre-increment, post-decrement, pre-decrement)
  - ***Relational Operators*** (==, != , >, <, >= & <=) ***Logical Operators*** (&&, || and !)
  - ***Bitwise Operators*** (&, |, ^, ~, >> and <<)
  - ***Assignment Operators*** (=, +=, -=, *=, etc)
- **Comments**
  - ***Single line comments*** . Example : // This is a comment
  - ***Multi line comments*** . Example : /* This is a multiline comments that extends for more than a line */
- **Looping Constructs**
  - while (condition) <single line statement>
  - while (condition) { multi line statements }
- **Conditions**
  - If and else statements.
- **Functions** - Accepting zero or one argument.
- **Literals** - Detecting integer constants and floating constants.
- **Preprocessor Directives:**
  - Eg- #include<stdio.h> #define PI 3.14

Contents:

- Introduction
    - Lexical Analyser
    - 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 with errors
2. Table 2: Test cases without errors
3. Figure 1: Analysis phase of Compiler
4. Figure 2: Symbol Table Entry Structure
5. Figure 3: Creation of symbol table
6. Figure 4: Insertion into symbol table
7. Figure 5: Screenshot of sample output
8. Figure 6: Screenshot of results in symbol table and constants table

Introduction:

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.

Lexical Analysis: This is the first stage/phase of the compiler. In this phase, the modified source code is taken from the language preprocessors. This modified source code is in the form of sentences. The lexical analyzer is responsible for breaking these sentences into a series of tokens, and removes all the whitespaces and comments from the source code. The symbol table is an important data structure created and used by the compilers to store the information regarding each identified token. It is used in both the analysis and synthesis phase of the compiler.

Flex Script:

We have designed the lexical analyser using lex (aka Flex in Linux Ubuntu).

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 r**ules 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.

Design of Programs:

Flow:

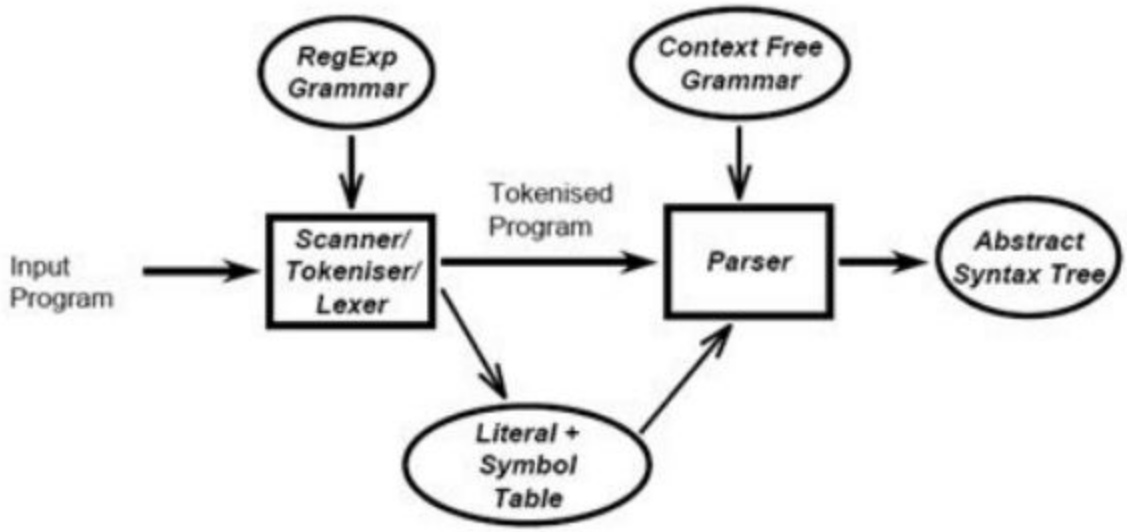

Figure 1

Lexical Analyser generates the series of tokens as well as symbol table which will be fed as input to the Parser (next phase of the compiler).

Flex Script: (.flex file)

```
/*
 * This file will contain the actual scanner written in fLEX.
*/
/* Definition section */
%{
        #include<stdio.h>
        #include <stdlib.h>
        #include "tokens.h"
   #include "symbolTable.h"
```

Lexical Analyser for the C Language

```
        #define NRML  "\x1B[0m"
        #define RED  "\x1B[31m"
        #define BLUE   "\x1B[34m"
        int cmnt_strt =0;

   stEntry** symbol_table;
   stEntry** constant_table;
%}


/* States to simplify readability of the Regular Expressions */
ALPHA [a-zA-Z]
SPACE [ ]
UND [_]
PLUS [+]
NEG [-]
DOT [.]
DIGIT [0-9]
IDENTIFIER {ALPHA}({ALPHA}|{DIGIT}|{UND})*
INVALID_IDENTIFIER [^`@#]
FUNCTION ({UND}|{ALPHA})({ALPHA}|{DIGIT}|{UND})*{SPACE}*\({SPACE}*\)
STRING \"([^\\\"]|\\.)*\"

%option yylineno

%x CMNT

/* Rules section */
%%


"/*"                    {cmnt_strt = yylineno; BEGIN CMNT;}
<CMNT>.|[ ]             ;
<CMNT>\n                        {yylineno++;}
<CMNT>"*/"                      {BEGIN INITIAL;}
<CMNT>"/*"                      {printf("\n%s%30s%30s%30s%d\n", RED,  "Nested comment", yytext
,"Line Number:", yylineno);printf("%s", NRML);}
<CMNT><<EOF>>                   {printf("\n%s%30s%30s%30s%d\n", RED,  "Unterminated comment",
yytext ,"Line Number:", yylineno);printf("%s", NRML); yyterminate();}
 /* Single Line Comment */
\/\/(.)*[\n]  {printf("\n%30s%30s%30s%d%30s%d\n", "SINGLE LINE COMMENT", yytext, "Line
Number:", yylineno, "Token Number:",SINGLE_LINE);}
```

Lexical Analyser for the C Language

```
[\t\r ]+ {
  /* ignore whitespace */ }

 /* Include directives */
#include{SPACE}*<{ALPHA}+\.?{ALPHA}+>[^;] { printf("\n%30s%30s%30s%d%30s%d\n",
"PREPROCESSING DIRECTIVE", yytext, "Line Number:", yylineno, "Token Number:",INCLUDE);}
#include{SPACE}*\"{ALPHA}+\.?{ALPHA}+\"[^;] { printf("\n%30s%30s%30s%d%30s%d\n",
"PREPROCESSING DIRECTIVE", yytext, "Line Number:", yylineno, "Token Number:",INCLUDE);}

 /* Illegal include statements */
#include{SPACE}*<{ALPHA}+\.?{ALPHA}+>{SPACE}*[;] { printf("\n%s%30s%30s%30s%d\n", RED,
"Illegal preprocessing detective. Ended with semicolon at ", yytext ,"Line Number:",
yylineno);printf("%s", NRML);}
#include{SPACE}*\"{ALPHA}+\.?{ALPHA}+\"{SPACE}*[;] { printf("\n%s%30s%30s%30s%d\n", RED,
"Illegal preprocessing detective. Ended with semicolon at", yytext ,"Line Number:",
yylineno);printf("%s", NRML);}
#include[^\n]* { printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal preprocessing directive ", yytext
,"Line Number:", yylineno);printf("%s", NRML);}
 /* #define statements */
#define{SPACE}+{ALPHA}({ALPHA}|{DIGIT}|{UND})*{SPACE}+({PLUS}|{NEG})?{DIGIT}*{DOT}{DIGIT}+[^;
] { printf("\n%30s%30s%30s%d%30s%d\n", "MACRO", yytext, "Line Number:", yylineno, "Token
Number:",DEF );}
#define{SPACE}+{ALPHA}({ALPHA}|{DIGIT}|{UND})*{SPACE}+{DIGIT}+[^;]                    {
printf("\n%30s%30s%30s%d%30s%d\n", "MACRO", yytext, "Line Number:", yylineno, "Token
Number:",DEF );}
#define{SPACE}+{ALPHA}({ALPHA}|{DIGIT}|{UND})*{SPACE}+{ALPHA}({ALPHA}|{UND}|{DIGIT})*[^;]
{ printf("\n%30s%30s%30s%d%30s%d\n", "MACRO", yytext, "Line Number:", yylineno, "Token
Number:",DEF); }


 /* Illegal #define statements */
#define{SPACE}+{ALPHA}({ALPHA}|{DIGIT}|{UND})*{SPACE}+({PLUS}|{NEG})?{DIGIT}*{DOT}{DIGIT}+{S
PACE}*[;] { printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal macro definition. Ended with semicolon
at ", yytext ,"Line Number:", yylineno);printf("%s", NRML);}
#define{SPACE}+{ALPHA}({ALPHA}|{DIGIT}|{UND})*{SPACE}+{DIGIT}+{SPACE}*[;]
{ printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal macro definition. Ended with semicolon at ",
yytext ,"Line Number:", yylineno);printf("%s", NRML);}
#define{SPACE}+{ALPHA}({ALPHA}|{DIGIT}|{UND})*{SPACE}+{ALPHA}({ALPHA}|{UND}|{DIGIT})*{SPA
CE}*[;]   { printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal macro definition. Ended with semicolon
at ", yytext ,"Line Number:", yylineno);printf("%s", NRML);}

 /* Keywords */
```

```
"int"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: INTEGER DATA TYPE", yytext, "Line Number:", yylineno, "Token Number:",INT, NRML);}
"short"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: SHORT INT DATATYPE", yytext, "Line Number:", yylineno, "Token Number:",SHORT,NRML );}
"long"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: LONG INT DATATYPE", yytext, "Line Number:", yylineno, "Token Number:", LONG,NRML );}
"long long"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: LONG LONG INT DATATYPE", yytext, "Line Number:", yylineno, "Token Number:", LONG_LONG,NRML );}
"signed"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: SIGNED INT", yytext, "Line Number:", yylineno, "Token Number:", SIGNED,NRML );}
"unsigned"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: UNSIGNED INT", yytext, "Line Number:", yylineno, "Token Number:",UNSIGNED,NRML );}
"char"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: CHAR DATATYPE", yytext, "Line Number:", yylineno, "Token Number:",CHAR,NRML );}
"if"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: IF", yytext, "Line Number:", yylineno, "Token Number:",IF,NRML );}
"else"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: ELSE", yytext, "Line Number:", yylineno, "Token Number:",ELSE,NRML );}
"while"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: WHILE", yytext, "Line Number:", yylineno, "Token Number:",WHILE,NRML );}
"continue"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: CONTINUE", yytext, "Line Number:", yylineno, "Token Number:",CONTINUE,NRML );}
"break"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: BREAK", yytext, "Line Number:", yylineno, "Token Number:",BREAK,NRML );}
"return"		{printf("\n%s%30s%30s%30s%d%30s%d%s\n",BLUE, "KEYWORD: RETURN", yytext, "Line Number:", yylineno, "Token Number:",RETURN,NRML );}

  /* Strings */
\"([^\\\"]|\\.)* {printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal String", yytext ,"Line Number:", yylineno);printf("%s", NRML);}
\"([^\\\"]|\\.)*\" {printf("\n%30s%30s%30s%d%30s%d\n", "STRING CONSTANT", yytext, "Line Number:", yylineno, "Token Number:",STRING_CONSTANT );}

  /* Rules for numeric constants needs to be before identifiers otherwise giving error */
0([xlX])({DIGIT}|[a-fA-F])+		{printf("\n%30s%30s%30s%d%30s%d\n", "HEXADECIMAL INTEGER", yytext, "Line Number:", yylineno, "Token Number:",HEXADECIMAL_CONSTANT );insert(constant_table,yytext,HEXADECIMAL_CONSTANT);}
0([xlX])({DIGIT}|[a-zA-Z])+		{printf("\n%s%30s%30s%30s%d%s\n", RED,  "Illegal Hexadecimal Constant", yytext ,"Line Number:", yylineno);printf("%s", NRML);}
0([0-7])+		{printf("\n%30s%30s%30s%d%30s%d\n", "OCTAL INTEGER", yytext, "Line Number:", yylineno, "Token Number:",OCTAL_CONSTANT );insert(constant_table,yytext,OCTAL_CONSTANT);}
```

```
0([0-9])+   { printf("\n%s%30s%30s%30s%d\n", RED,  "  Illegal octal constant", yytext ,"Line
Number:", yylineno);printf("%s", NRML);}
{PLUS}?{DIGIT}*{DOT}{DIGIT}+ {printf("\n%30s%30s%30s%d%30s%d\n", "POSITIVE FRACTION",
yytext, "Line Number:", yylineno, "Token Number:",FLOATING_CONSTANT
);insert(constant_table,yytext,FLOATING_CONSTANT);}
{NEG}{DIGIT}*{DOT}{DIGIT}+   {printf("\n%30s%30s%30s%d%30s%d\n", "NEGATIVE FRACTION",
yytext, "Line Number:", yylineno, "Token Number:",FLOATING_CONSTANT
);insert(constant_table,yytext,FLOATING_CONSTANT);}

 /* Invalid mantissa exponent forms */
({PLUS}?|{NEG}){DIGIT}+([e|E])({PLUS}?|{NEG}){DIGIT}*{DOT}{DIGIT}*
{printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal Floating Constant ", yytext ,"Line Number:",
yylineno);printf("%s", NRML);}
({PLUS}?|{NEG}){DIGIT}*{DOT}{DIGIT}+([e|E])({PLUS}?|{NEG}){DIGIT}*{DOT}{DIGIT}*
{printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal Floating Constant ", yytext ,"Line Number:",
yylineno);printf("%s", NRML);}

 /* Valid Mantissa Exponent forms */
({PLUS}?|{NEG}){DIGIT}+([e|E])({PLUS}?|{NEG}){DIGIT}+ {printf("\n%30s%30s%30s%d%30s%d\n",
"FLOATING CONSTANT", yytext, "Line Number:", yylineno, "Token
Number:",FLOATING_CONSTANT );printf("%s",
NRML);insert(constant_table,yytext,FLOATING_CONSTANT);}
({PLUS}?|{NEG}){DIGIT}*{DOT}{DIGIT}+([e|E])({PLUS}?|{NEG}){DIGIT}+
{printf("\n%30s%30s%30s%d%30s%d\n", "FLOATING CONSTANT", yytext, "Line Number:",
yylineno, "Token Number:",FLOATING_CONSTANT );printf("%s",
NRML);insert(constant_table,yytext,FLOATING_CONSTANT);}




{PLUS}?{DIGIT}+          {printf("\n%30s%30s%30s%d%30s%d\n", "POSITIVE INTEGER", yytext, "Line
Number:", yylineno, "Token Number:",INTEGER_CONSTANT
);insert(constant_table,yytext,INTEGER_CONSTANT);}
{NEG}{DIGIT}+            {printf("\n%30s%30s%30s%d%30s%d\n", "NEGATIVE INTEGER", yytext,
"Line Number:", yylineno, "Token Number:",INTEGER_CONSTANT
);insert(constant_table,yytext,INTEGER_CONSTANT);}

({ALPHA}|{DIGIT}|{UND})*{INVALID_IDENTIFIER}+({ALPHA}|{DIGIT}|{UND})* {
printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal identifier ", yytext ,"Line Number:",
yylineno);printf("%s", NRML);}
{IDENTIFIER}             {printf("\n%30s%30s%30s%d%30s%d\n", "IDENTIFIER", yytext, "Line
Number:", yylineno, "Token Number:",IDENTIFIER );insert(symbol_table,yytext,IDENTIFIER);}
```

```
{DIGIT}+({ALPHA}|{UND})+   { printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal identifier ", yytext
,"Line Number:", yylineno);printf("%s", NRML);}



 /* Shorthand assignment operators */
"+="            {printf("\n%30s%30s%30s%d%30s%d\n", "PLUS EQUAL TO", yytext, "Line
Number:", yylineno, "Token Number:",PLUSEQ );}
"-="            {printf("\n%30s%30s%30s%d%30s%d\n", "MINUS EQUAL TO", yytext, "Line
Number:", yylineno, "Token Number:",MINUSEQ );}
"*="            {printf("\n%30s%30s%30s%d%30s%d\n", "MUL EQUAL TO", yytext, "Line Number:",
yylineno, "Token Number:",MULEQ );}
"/="            {printf("\n%30s%30s%30s%d%30s%d\n", "DIV EQUAL TO", yytext, "Line Number:",
yylineno, "Token Number:",DIVEQ );}
"%="            {printf("\n%30s%30s%30s%d%30s%d\n", "MOD EQUAL TO", yytext, "Line
Number:", yylineno, "Token Number:",MODEQ);}

 /* Relational operators */
"="             {printf("\n%30s%30s%30s%d%30s%d\n", "EQUALTO", yytext, "Line Number:",
yylineno, "Token Number:",EQ );}
"!="            {printf("\n%30s%30s%30s%d%30s%d\n", "UNEQUAL", yytext, "Line Number:",
yylineno, "Token Number:",NEQ );}
">"             {printf("\n%30s%30s%30s%d%30s%d\n", "GREATER THAN", yytext, "Line Number:",
yylineno, "Token Number:",GT );}
"<"             {printf("\n%30s%30s%30s%d%30s%d\n", "LESS THAN", yytext, "Line Number:",
yylineno, "Token Number:",LT );}
">="            {printf("\n%30s%30s%30s%d%30s%d\n", "GREATER THAN EQUAL TO", yytext,
"Line Number:", yylineno, "Token Number:",GE );}
"<="            {printf("\n%30s%30s%30s%d%30s%d\n", "LESSER THAN EQUAL TO", yytext, "Line
Number:", yylineno, "Token Number:",LE );}
"=="            {printf("\n%30s%30s%30s%d%30s%d\n", "EQUAL TO EQUAL TO", yytext, "Line
Number:", yylineno, "Token Number:",EQEQ );}

 /* Arithmetic Operators */
"++"            {printf("\n%30s%30s%30s%d%30s%d\n", "INCREMENT", yytext, "Line Number:",
yylineno, "Token Number:",INC );}
"--"            {printf("\n%30s%30s%30s%d%30s%d\n", "DECREMENT", yytext, "Line Number:",
yylineno, "Token Number:",DEC );}
"+"             {printf("\n%30s%30s%30s%d%30s%d\n", "PLUS", yytext, "Line Number:", yylineno,
"Token Number:",PLUS );}
"-"             {printf("\n%30s%30s%30s%d%30s%d\n", "MINUS", yytext, "Line Number:", yylineno,
"Token Number:",MINUS );}
```

```
"*"             {printf("\n%30s%30s%30s%d%30s%d\n", "MULTIPLICATION", yytext, "Line
Number:", yylineno, "Token Number:",MUL );}
"/"             {printf("\n%30s%30s%30s%d%30s%d\n", "FORWARD SLASH or DIVISION", yytext,
"Line Number:", yylineno, "Token Number:",DIV );}
"%"             {printf("\n%30s%30s%30s%d%30s%d\n", "MODULUS", yytext, "Line Number:",
yylineno, "Token Number:",MODULO);}


 /* Punctuators */
","             {printf("\n%30s%30s%30s%d%30s%d\n", "COMMA", yytext, "Line Number:",
yylineno, "Token Number:",COMMA );}
";"             {printf("\n%30s%30s%30s%d%30s%d\n", "SEMICOLON", yytext, "Line Number:",
yylineno, "Token Number:",SEMICOLON );}
"("             {printf("\n%30s%30s%30s%d%30s%d\n", "OPEN PARANTHESIS", yytext, "Line
Number:", yylineno, "Token Number:",OPEN_PARANTHESIS );}
")"             {printf("\n%30s%30s%30s%d%30s%d\n", "CLOSE PARANTHESIS", yytext, "Line
Number:", yylineno, "Token Number:",CLOSE_PARANTHESIS );}
"{"             {printf("\n%30s%30s%30s%d%30s%d\n", "OPEN_BRACE", yytext, "Line Number:",
yylineno, "Token Number:",OPEN_BRACE );}
"}"             {printf("\n%30s%30s%30s%d%30s%d\n", "CLOSE_BRACE", yytext, "Line Number:",
yylineno, "Token Number:",CLOSE_BRACE);}
"["             {printf("\n%30s%30s%30s%d%30s%d\n", "OPEN_SQR_BKT", yytext, "Line
Number:", yylineno, "Token Number:",OPEN_SQR_BKT );}
"]"             {printf("\n%30s%30s%30s%d%30s%d\n", "CLOSE_SQR_BKT", yytext, "Line
Number:", yylineno, "Token Number:",CLOSE_SQR_BKT );}


. {printf("\n%s%30s%30s%30s%d\n", RED,  "Illegal Character ", yytext ,"Line Number:",
yylineno);printf("%s", NRML);}
\n {}
%%

int yywrap(){
   return 1;
}

int main(){
   symbol_table = new_table();
   constant_table = new_table();
   yylex();
   printf("\n\nDisplaying Symbol Table\n");
   display(symbol_table);
   printf("\n\nDisplaying Constant Table\n");
```

```
    display(constant_table);
    return 0;
}
```

Explanation:-

Files Created:

1. scanner.flex - Flex script which generates the stream of tokens and symbol table.
2. tokens.h - Defines all tokens to be identified along with their corresponding token numbers using the ‘enum’ operation.
3. symbolTable.h - Defines all the functions pertaining to creation and updation of symbol table.
4. test.c - The input C program (test case).

The flex script recognises the following classes of tokens from the input:

Preprocessor Directives:

Statements Processed: #include<stdio.h>, #include “tokens.h”, #define var1 var2

Token Generated: Preprocessing Directive

Error Cases Handled: Detecting if the directive ends with a semicolon.

Single Line Comments:

Statements Processed: //.....

Token Generated: Single Line Comment

Multi Line Comments:

Statements Processed: /*.....*/, /*...../*...*/

Token Generated: Multi Line Comment

Error Case Handled: Detecting nested comments, Incomplete Comments (/*.....)

Literals:

Examples of Values Processed:"Hello\nWorld", 2, +2.09, -3E12, 0x123fd, 045, -45,etc
Tokens Generated: Integer Constant, Hexadecimal Constant, Floating Constant, String Constant, Octal Constant
Error Cases Handled: Invalid Hexadecimal Number (Eg- 0xkn12), Invalid floating constant (Eg- 0.2e-9.6), Invalid Octal constant (Eg - 088), Invalid String (Eg - "I like to"play")

Identifier:

Statements Processed: xyz, x1, x_y_z
Token Generated: Identifier
Error Case Handled: Identifier starting with a number.

Operators:

Statements Processed: **Types of operators mentioned in the abstract.**
Token Generated: Operator (with corresponding type)

Punctuators:

Statements Processed: **Types of punctuators mentioned in the abstract.**
Token Generated: Punctuator (with corresponding type)

Keywords:

Statements Processed: int, short, long, long long, signed, unsigned, char, if, else, while, continue, break, return

Token Generated: Keyword

Test Cases:

With Errors:

Table 1

Sl.No	Test Case	Expected Output	Status
1.	/* this nesting shouldn't be /* allowed */	Error: Nested comment	PASS
2.	int d = 0x6j;	Error: Illegal Hexadecimal Constant	PASS
3.	int a@b =5;	Error: Illegal identifier	PASS
4.	/* int a = 5; <<EOF>>	Error: Comment not terminated	PASS
5.	int 1b = c;	Error: Invalid Identifier	PASS

Without Errors:

Table 2

Sl.No	Test Case	Expected Output	Status
1.	long c2 = 4e4;	KEYWORD: long IDENTIFIER: c2 OPERATOR: = 4e4: FLOATING CONSTANT SEMICOLON	PASS
2.	// Hello world	Single Line Comment: //Hello World	PASS
3.	#define a 3	Macro	PASS
4.	if(a>5){;}	KEYWORD: if	PASS

		Open Parentheses: (IDENTIFIER: a Greater Than: > Constant: 5 Close Parentheses:) Open Parentheses: { SEMICOLON Close Parentheses: }	
5.	X_t += 4;	IDENTIFIER: X_t PLUS EQUAL TO: += CONSTANT: 4 SEMICOLON	PASS

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:

1. The Regex for Identifiers: The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.
2. Multiline comments should be supported: This has been supported by using custom regular algorithm especially robust in cases where tricky characters like * or / are used within the comments.
3. Literals: Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, strings, hexadecimal integers, octal integers,etc.
4. Error Handling for Incomplete String: Open and close quote missing, both kind of errors have been handled in the rules written in the script.
5. Error Handling for Nested Comments: This use-case has been handled by the custom defined regular expressions which help throw errors when comment opening or closing is missing.

At the end of the token recognition, the lexer prints a list of all the identifiers and constants present in the program. The symbol table usage has been explained in the following section.

Important regular expressions:

- Multi line comments (using states)
  %x CMNT
  "/*" {cmnt_strt = yylineno; BEGIN CMNT;} //indicates the start of a multiline comment
  <CMNT>.|[ ] ; //ignore all chars and spaces
  <CMNT>\n {yylineno++;} //increment line number on finding a \n
  <CMNT>"*/" {BEGIN INITIAL;} //if we find a close comment, exit the logic for multi line comment and move back to the control DFA used by flex.
  <CMNT>"/*" {printf("\n%s%30s%30s%30s%d\n", RED, "Nested comment", yytext ,"Line Number:", yylineno);printf("%s", NRML);} // case for checking neste comments which aren't allowed by C.
  <CMNT><<EOF>> {printf("\n%s%30s%30s%30s%d\n", RED, "Unterminated comment", yytext ,"Line Number:", yylineno);printf("%s", NRML); yyterminate();} // Case where a multi line comment is not terminated and an EOF is encountered. The yyterminate() function
- Strings - \"([^\\\"]|\\.)*\"
  Strings cannot include backslashes unless they are escaped and no double quotes unless they are escaped.

Symbol Table and Constants Table:

We have implemented a closed hash table for both symbol table and constants table. The datatype of each row or entry is defined as explained below.

```
typedef struct symbolTableEntry
{
        char* lexeme;
        int token;
        struct symbolTableEntry * next;
} stEntry;
```

Figure 2

The struct contains a character pointer to the lexeme that is identified by the lexer, an integer token corresponding to the lexeme and a struct pointer to the next node if there is chaining at that hash table index. The file "**tokens.h**" contains all the integer tokens.

Same hash table is being used as both symbol table and constants table. The initialization is done as shown below.

```
/* Declare the pointers and hash table */

stEntry** symbol_table;
stEntry** constant_table;
symbol_table = new_table();
constant_table = new_table();
```

Figure 3

After the initialization, all the identifiers and constants are added to the symbol table and constants are inserted to symbol table and constants table respectively. Whenever lexer matches the pattern (lexeme), the lexeme with its corresponding integer token is added to the symbol table or constant table using the function **insert().** For example, **insert(symbol_table, yytext, IDENTIFIER)** will insert the identified identifier lexeme and token to symbol table. Similarly **insert(constant_symbol, yytext, INTEGER_TABLE)** to constants table.
The insert function :

```c
void insert(stEntry** hash_table, char* lexeme, int token){
        if(search(hash_table, lexeme)!=NULL)
        {
                return;
        }

        unsigned int index = hash_value(lexeme);
        stEntry* newentry= new_entry(lexeme,token);

        if (newentry == NULL){
                printf("Cannot insert token into the symbol table\n");
                exit(1);
        }

        stEntry* head = hash_table[index];

        if(head==NULL){
                hash_table[index] = newentry;
        }
        else
        {
                newentry->next = hash_table[index];
                hash_table[index] = newentry;
        }
}
```

Figure 4

The process is as follows. When a lexeme is passed to the insert function, a hash for the lexeme is created using djb2 algorithm. Then the hash is mapped to a index in range ***[0, TABLE_SIZE)***. If the lexeme is not already present in the table, then it is inserted at the index. While inserting at the index, if one or more entries are already present at the index then the new entry is added at the beginning and the remaining entries are chained at the end of the newly added entry as a linked list.

Results:

The output can be explained using the following screenshots:

```
darshan@DV:~/CS/CD/Compiler-Design/Scanner$ ./a.out

       PREPROCESSING DIRECTIVE            #include<stdio.h>
                  Line Number:2                  Token Number:150

       PREPROCESSING DIRECTIVE           #include<stdlib.h>
                  Line Number:3                  Token Number:150

    KEYWORD: INTEGER DATA TYPE                        int              Line Number:4              Token Number:1

                  IDENTIFIER                         main              Line Number:4              Token Number:200

             OPEN PARANTHESIS                          (              Line Number:4              Token Number:103

            CLOSE PARANTHESIS                          )              Line Number:4              Token Number:104

                  OPEN_BRACE                           {              Line Number:5              Token Number:105

    KEYWORD: INTEGER DATA TYPE                        int              Line Number:6              Token Number:1

                  IDENTIFIER                           x              Line Number:6              Token Number:200

                       COMMA                           ,              Line Number:6              Token Number:100

                  IDENTIFIER                           y              Line Number:6              Token Number:200

                   SEMICOLON                           ;              Line Number:6              Token Number:101

KEYWORD: LONG LONG INT DATATYPE                   long long            Line Number:7              Token Number:4

    KEYWORD: INTEGER DATA TYPE                        int              Line Number:7              Token Number:1
```

Figure 5

```
Displaying Symbol Table

================================================================
            lexeme              |             token
================================================================
main                            |                           200
b                               |                           200
i                               |                           200
printf                          |                           200

================================================================


Displaying Constant Table

================================================================
            lexeme              |             token
================================================================
0                               |                           250
10                              |                           250

================================================================
```

Figure 6

Future Work:

Currently, the flex script takes care of most of the rules of C language, but is not fully exhaustive in nature. Future work could include taking care of more keywords such as struct, or even handling nested if-else blocks. Further, there may be cases where the Regular Expressions used could be simplified , thus increasing the efficiency overall.

References:

1. Compilers: Principles, Techniques, and Tools (Dragonbook)
2. LLVM Tutorial on how to build a toy language ‘Kaleidoscope’. Used to understand the functioning of each element of a compiler.
3. Basics of Compiler Design - Torben Ægidius Mogensen.