# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

**Kattankulathur, Chengalpattu District - 603203**


## 18CSC304J/ COMPLIER DESIGN

## MINI PROJECT REPORT

## Lexical Analyser for a subset of the C language

**Guided by:**

**Mr. H. karthikeyan**

**Submitted By:**

**GANTA PRIYA (RA2011029010040)**

**ARCHANA KUMARI (RA2011029010046)**

**DIVIJ VERMA (RA2011029010048)**

**Aim: -** Designing a scanner for c language.

## ABSTRACT

The C language is widely used in software development, and its popularity has led to the creation of numerous compilers for the language. As a part of a mini project in compiler design, a scanner for a C language compiler can be designed. The scanner, also known as a lexical analyzer, reads the source code and generates a stream of tokens that represent the various elements of the code. This project involves defining the lexical structure of the C language, writing regular expressions for each token, implementing the scanner, and testing it to ensure that it's working correctly. By following best practices for scanner design, such as using a robust regular expression library and handling errors gracefully, a robust and efficient scanner can be created that can be used as a component in a larger C language compiler.

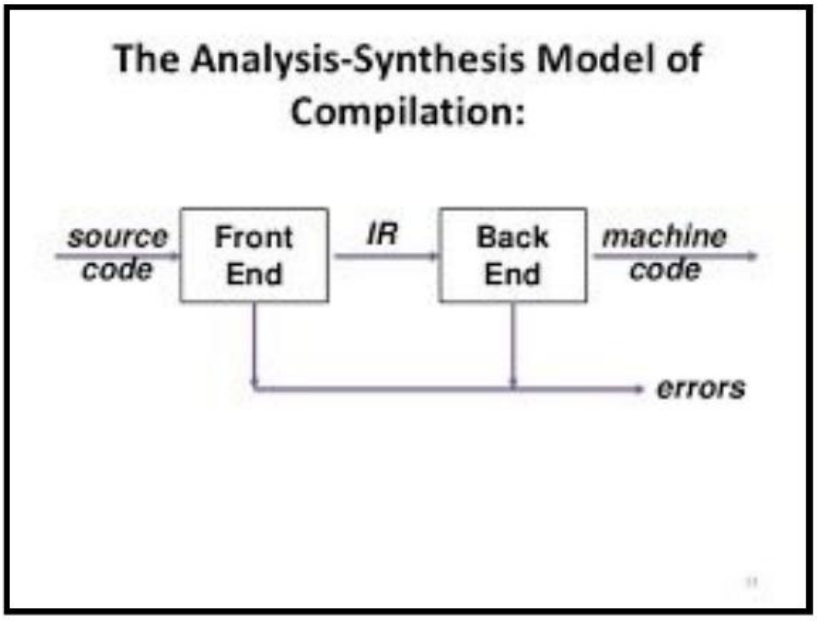
A scanner, also known as a lexical analyzer, is a critical component in the design of a compiler. It reads the source code and generates a stream of tokens that represent the various elements of the code. This project involves designing a scanner for a C language compiler. The lexical structure of the C language is analyzed, and regular expressions are written for each token. The scanner is implemented using a programming language, and a regular expression library is used to simplify the code and improve performance. The scanner is tested by running it on a set of sample input files, and the output is compared to the expected results. The resulting scanner is a robust and efficient component that accurately identifies and tokenizes the various elements of the source code. This project provides valuable experience in compiler design and implementation, including a deep understanding of the language's lexical structure and proficiency in writing regular expressions and implementing scanners.

## Introduction

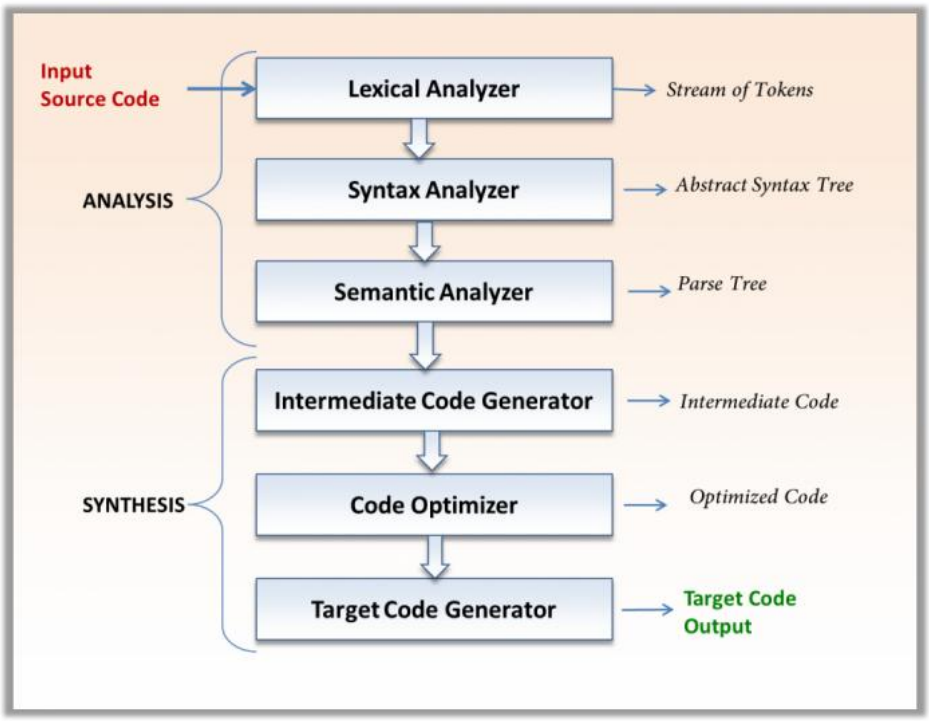
A compiler is a program that can read a program in one language - the source language - and translate it to an equivalent program in another language - the target language. An important role of the compiler is to detect any errors in the source program during the translation process.

**Structure of a compiler**

There are two parts involved in the translation of a program in the source language into a semantically equivalent target program: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part. The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler and the synthesis part is called as the back end.



## Phases of compilation




The compilation process operates as a sequence of phases each of which transforms one representation of the source program to another.

● The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes For each lexeme, the lexical analyzer produces as output a token that it passes on to the subsequent phase, syntax analysis.

● The second phase of the compiler is syntax analysis or parsing. The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

● The third phase is the semantic analysis. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

● After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate code representation.

● The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

● The last phase is the code generation. The code generator takes as input an intermediate representation of the source program and maps it into the target language

**Lexical Analysis**

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The lexical analyzer maintains a data structure called as the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol 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.

**Designing a scanner for C language**

We have used Flex to perform lexical analysis on a subset of the C programming language. Flex is a lexical analyzer generator that takes in a set of descriptions of possible tokens and produces a C file that performs lexical analysis and identifies the tokens. Here we describe the functionality and construction of our scanner. This document is divided into the following sections:

● Functionality: Contains a description of our Flex program and the variety of tokens that it can identify and the error handling strategies.

● Symbol table and Constants table: Contains an overview of the architecture of the symbol and constants table which contain descriptions of the lexemes identified during lexical analysis.

● Code Organisation: Contains a description of the files used for lexical analysis

● Source Code: Contains the source code used for lexical analysis

**Functionality**

Below is a list containing the different tokens that are identified by our Flex program. It also gives a detailed description of how the different tokens are identified and how errors are detected if any.

**Keywords**

The keywords identified are: int, long, short, long long, 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}|_)+

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 . When a /* pattern is found, we note down the line number and enter the 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 lexer is in the state. Since, it is an exclusive state, only the patterns that are defined for this state (the ones prepended within the lex file are matched, rest of the patterns are inactive.

● We also identify nested comments. If we find another /* while still in the 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 lexer

matches <> pattern while still in the  state, which means that a */ has not been found until the end of file and therefore the comment has not been terminated.

Strings

The lexer 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 lexer 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.

● 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.

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.

Preprocessor Directives

The filenames that come after the #include are selectively identified through the exclusive state 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 lexer switches to the state where it can tokenize filenames of the form “stdio.h” or . Filenames of any other format are considered as illegal and an error message regarding the same is printed.

**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.

The struct consists of a character pointer to the lexeme that is matched by the lexer, 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 create_table() function. The function returns a pointer to a new created hash table which is basically an array of pointers of the type entry_t*.

Every time the lexer 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(symbol_table, yytext, INT) inserts the keyword INT into the symbol table and insert(constant_table, 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 [2] . The hash table has a fixed size as defined by the user using HASH_TABLE_SIZE. The generated hash value is mapped to a value in the range [0, HASH_TABLE_SIZE) through the operation hash_value % HASH_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.

**Code Organisation**

The entire code for lexical analysis is broken down into 3 files: lexer.l, tokens.h and symboltable.h

| File | Contents |
|---|---|
| Lexer.l | A lex file containing the lex specification of regular expressions |
| Tokens.h | Contains enumerated constants for keywords, operator, special symbols, constants and identifiers. |
| Symbiltable.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. |

## Requirements to run the script:

To run the lexical analyzer compiler design project, you will need the following requirements:

1. Operating System: The project can be developed and executed on any operating system, including Windows, Linux, or macOS.
2. Programming Language: The project can be developed using any programming language, but it is recommended to use a language that has good regular expression support. Popular choices include C, C++, Java, and Python.
3. Text Editor or Integrated Development Environment (IDE): You will need a text editor or an IDE to write the code for the scanner. Some popular options include Visual Studio Code, Sublime Text, Eclipse, or NetBeans.
4. Regular Expression Library: To simplify the code and improve performance, a regular expression library can be used. Popular libraries include the PCRE (Perl Compatible Regular Expressions) library, the RE2 library, or the Boost.Regex library.
5. Sample Input Files: To test the scanner, you will need a set of sample input files that contain C code. These files should be well-structured and include all of the elements of the language's lexical structure.
6. Output Testing Tool: A tool is needed to compare the output generated by the scanner to the expected output. Tools like DiffChecker or WinMerge can be used for this purpose.
7. Documentation and Reports: It is essential to document the project's requirements, design, implementation, and testing phases. Additionally, reports should be generated to present the project's findings and conclusions.

# Code:

## Lexer.l

```
%{

#include <stdlib.h>
#include <stdio.h>
#include "symboltable.h"
#include "tokens.h"

entry_t** symbol_table;
entry_t** constant_table;
int cmnt_strt = 0;

%}

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

 /* Exclusive states */
%x CMNT
%x PREPROC

%%
  /* Keywords*/
"int"                                {printf("\t%-30s : %3d\n",yytext,INT);}
"long"                               {printf("\t%-30s : %3d\n",yytext,LONG);}
"long long"                          {printf("\t%-30s : %3d\n",yytext,LONG_LONG);}
"short"                              {printf("\t%-30s : %3d\n",yytext,SHORT);}
"signed"                             {printf("\t%-30s : %3d\n",yytext,SIGNED);}
"unsigned"                           {printf("\t%-30s : %3d\n",yytext,UNSIGNED);}
"for"                                {printf("\t%-30s : %3d\n",yytext,FOR);}
"break"                              {printf("\t%-30s : %3d\n",yytext,BREAK);}
"continue"                           {printf("\t%-30s : %3d\n",yytext,CONTINUE);}
"if"                                 {printf("\t%-30s : %3d\n",yytext,IF);}
"else"                               {printf("\t%-30s : %3d\n",yytext,ELSE);}
"return"                             {printf("\t%-30s : %3d\n",yytext,RETURN);}

{identifier}                         {printf("\t%-30s : %3d\n",
yytext,IDENTIFIER);
                                    insert( symbol_table,yytext,IDENTIFIER );}
{ws}                                 ;
```

```
 42 [+\-]?[0][x|X]{hex}+[lLuU]?        {printf("\t%-30s : %3d\n",
yytext,HEX_CONSTANT);
 43                                                              insert(
constant_table,yytext,HEX_CONSTANT);}
 44 [+\-]?{digit}+[lLuU]?               {printf("\t%-30s : %3d\n",
yytext,DEC_CONSTANT);
 45                                                              insert(
constant_table,yytext,DEC_CONSTANT);}
 46 "/*"                               {cmnt_strt = yylineno; BEGIN CMNT;}
 47 <CMNT>.|{ws}                       ;
 48 <CMNT>\n                           {yylineno++;}
 49 <CMNT>"*/"                         {BEGIN INITIAL;}
 50 <CMNT>"/*"                         {printf("Line %3d: Nested comments are not
valid!\n",yylineno);}
 51 <CMNT><<EOF>>                     {printf("Line %3d: Unterminated comment\n",
cmnt_strt); yyterminate();}
 52 ^"#include"                       {BEGIN PREPROC;}
 53 <PREPROC>"<"[^<>\n]+">"            {printf("\t%-30s :
%3d\n",yytext,HEADER_FILE);}
 54 <PREPROC>{ws}                      ;
 55 <PREPROC>\"[^"\n]+\"               {printf("\t%-30s :
%3d\n",yytext,HEADER_FILE);}
 56 <PREPROC>\n                       {yylineno++; BEGIN INITIAL;}
 57 <PREPROC>.                        {printf("Line %3d: Illegal header file format
\n",yylineno);}
 58 "//".*                            ;
 59
 60 \"[^\"\n]*\"     {
 61
 62   if(yytext[yyleng-2]=='\\') /* check if it was an escaped quote */
 63   {
 64     yyless(yyleng-1);       /* push the quote back if it was escaped */
 65     yymore();
 66   }
 67   else
 68   insert( constant_table,yytext,STRING);
 69  }
 70
 71 \"[^\"\n]*$                      {printf("Line %3d: Unterminated string
%s\n",yylineno,yytext);}
 72 {digit}+({letter}|_)+             {printf("Line %3d: Illegal identifier name
%s\n",yylineno,yytext);}
 73 \n                               {yylineno++;}
 74 "--"                                        {printf("\t%-30s :
%3d\n",yytext,DECREMENT);}
```

```
 75 "++"                                    {printf("\t%-30s :
%3d\n",yytext,INCREMENT);}
 76 "->"                                    {printf("\t%-30s :
%3d\n",yytext,PTR_SELECT);}
 77 "&&"                                    {printf("\t%-30s :
%3d\n",yytext,LOGICAL_AND);}
 78 "||"                                    {printf("\t%-30s :
%3d\n",yytext,LOGICAL_OR);}
 79 "<="                                    {printf("\t%-30s :
%3d\n",yytext,LS_THAN_EQ);}
 80 ">="                                    {printf("\t%-30s :
%3d\n",yytext,GR_THAN_EQ);}
 81 "=="                                    {printf("\t%-30s :
%3d\n",yytext,EQ);}
 82 "!="                                 {printf("\t%-30s :
%3d\n",yytext,NOT_EQ);}
 83 ";"                                  {printf("\t%-30s :
%3d\n",yytext,DELIMITER);}
 84 "{"                               {printf("\t%-30s : %3d\n",yytext,OPEN_BRACES);}
 85 "}"                               {printf("\t%-30s :
%3d\n",yytext,CLOSE_BRACES);}
 86 ","                                  {printf("\t%-30s : %3d\n",yytext,COMMA);}
 87 "="                                  {printf("\t%-30s :
%3d\n",yytext,ASSIGN);}
 88 "("                                  {printf("\t%-30s :
%3d\n",yytext,OPEN_PAR);}
 89 ")"                                  {printf("\t%-30s :
%3d\n",yytext,CLOSE_PAR);}
 90 "["                               {printf("\t%-30s :
%3d\n",yytext,OPEN_SQ_BRKT);}
 91 "]"                               {printf("\t%-30s :
%3d\n",yytext,CLOSE_SQ_BRKT);}
 92 "-"                                  {printf("\t%-30s : %3d\n",yytext,MINUS);}
 93 "+"                                  {printf("\t%-30s : %3d\n",yytext,PLUS);}
 94 "*"                                  {printf("\t%-30s : %3d\n",yytext,STAR);}
 95 "/"                                {printf("\t%-30s : %3d\n",yytext,FW_SLASH);}
 96 "%"                                  {printf("\t%-30s :
%3d\n",yytext,MODULO);}
 97 "<"                                  {printf("\t%-30s :
%3d\n",yytext,LS_THAN);}
 98 ">"                                  {printf("\t%-30s :
%3d\n",yytext,GR_THAN);}
 99 .                                 {printf("Line %3d: Illegal character
%s\n",yylineno,yytext);}
100
```

```
%%

int main()
{
   yyin=fopen("testcases/test-case-5.c","r");
   symbol_table=create_table();
   constant_table=create_table();
   yylex();
   printf("\n\tSymbol table");
   display(symbol_table);
   printf("\n\tConstants Table");
   display(constant_table);
   printf("NOTE: Please refer tokens.h for token meanings\n");
}
```

Symboltable.h

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

#define HASH_TABLE_SIZE 100

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

typedef struct entry_s entry_t;

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

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

```
    int i;

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

    return hash_table_ptr;
}

/* Generate hash from a string. Then generate an index in [0, HASH_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 % HASH_TABLE_SIZE; // return an index in [0, HASH_TABLE_SIZE)
}

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

    /* Allocate space for newentry */
    if( ( newentry = malloc( sizeof( entry_t ) ) ) == 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_t* search( entry_t** hash_table_ptr, char* lexeme )
{
    uint32_t idx = 0;
    entry_t* 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_t** hash_table_ptr, char* lexeme, int token_name )
{
    if( search( hash_table_ptr, lexeme ) != NULL) // If lexeme already exists, don't
insert, return
        return;

    uint32_t idx;
    entry_t* newentry = NULL;
    entry_t* head = NULL;
```

```
    idx = hash( lexeme ); // Get the index for this lexeme based on the hash function
    newentry = create_entry( lexeme, token_name ); // Create an entry using the
<lexeme, token> pair

    if(newentry == NULL) // In case there was some error while executing create_entry()
    {
        printf("Insert failed. New entry could not be created.");
        exit(1);
    }

    head = hash_table_ptr[idx]; // get the head entry at this index

    if(head == NULL) // This is the first lexeme that matches this hash index
    {
        hash_table_ptr[idx] = newentry;
    }
    else // if not, add this entry to the head
    {
        newentry->successor = hash_table_ptr[idx];
        hash_table_ptr[idx] = newentry;
    }
}

// Traverse the hash table and print all the entries
void display(entry_t** hash_table_ptr)
{
    int i;
    entry_t* traverser;
    printf("\n=========================================\n");
    printf("\t < lexeme , token >\n");
    printf("=========================================\n");

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

        while( traverser != NULL)
        {
            printf("< %-30s, %3d >\n", traverser->lexeme, traverser->token_name);
            traverser = traverser->successor;
        }
    }
    printf("=========================================\n");
```

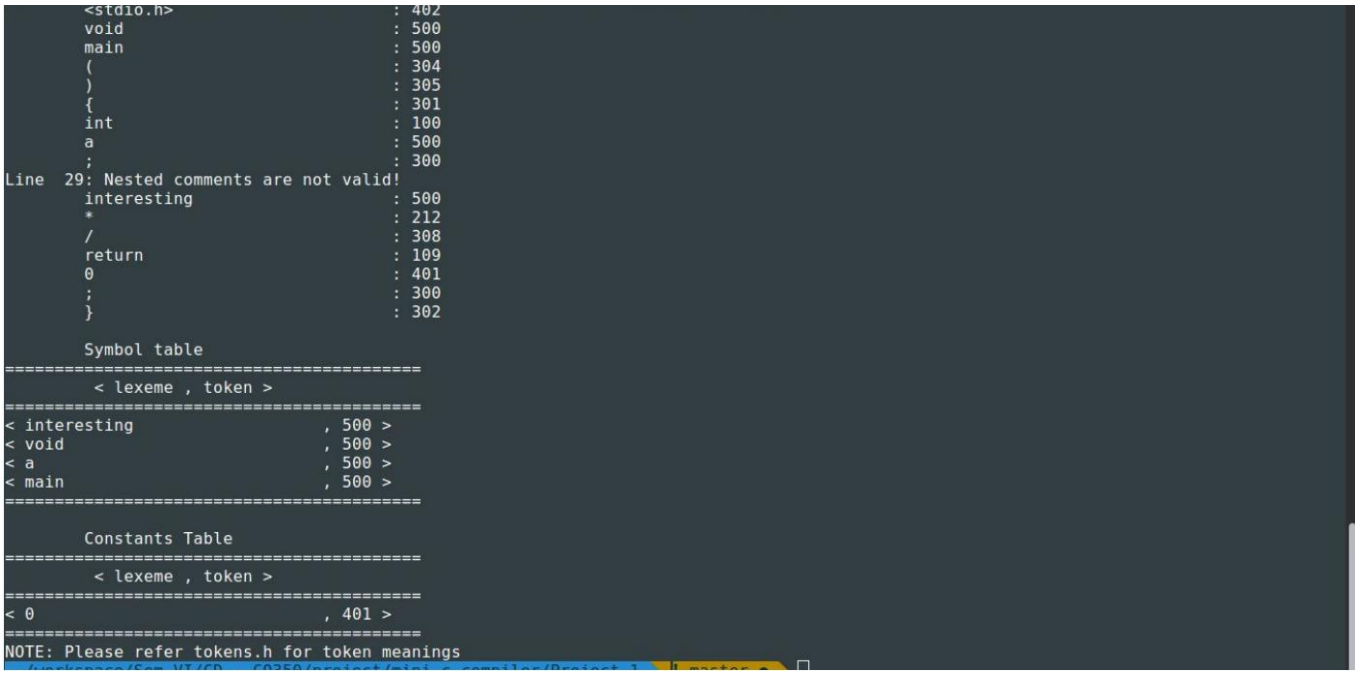
```
}
```

Tokens.h

```
enum keywords
{
  INT=100,
  LONG,
  LONG_LONG,
  SHORT,
  SIGNED,
  UNSIGNED,
  FOR,
  BREAK,
  CONTINUE,
  RETURN,
  CHAR,
  IF,
  ELSE
};

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
};

enum special_symbols
{
```

```
  DELIMITER=300,
  OPEN_BRACES,
  CLOSE_BRACES,
  COMMA,
  OPEN_PAR,
  CLOSE_PAR,
  OPEN_SQ_BRKT,
  CLOSE_SQ_BRKT,
  FW_SLASH
};

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

enum IDENTIFIER
{
  IDENTIFIER=500
};
```

# Output:




```
Test Case 1
 - Test for single line comments
 - Test for multi-line comments
 - Test for single line nested comments
 - Test for multiline nested comments

The output in lex should remove all the comments including this one
*/

#include<stdio.h>

void main(){
    // Single line comment

    /* Multi-line comment
       Like 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 */ interesting */

        return 0;
}
```

```
        <stdio.h>                       : 402
        void                            : 500
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
Line  21: Unterminated comment

        Symbol table
========================================
         < lexeme , token >
========================================
< void                          , 500 >
< main                          , 500 >
========================================

        Constants Table
========================================
         < lexeme , token >
========================================
========================================
NOTE: Please refer tokens.h for token meanings
```

Test Case 2

- Test for multi-line comment that doesn't end till EOF

The output in lex should print as error message when the comment does not terminate
It should remove the comments that terminate
*/

```
#include<stdio.h>

void main(){

        // This is fine
        /* This as well
        like we know */

        /* This is not fine since
        this comment has to end somewhere

        return 0;
}
```

```
File  Edit  View  Search  Terminal  Help
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  cc lex.yy.c -ll
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  ./a.out
        <stdio.h>                       : 402
Line  15: Illegal header file format
        <stdlib.h>                      : 402
        "custom.h"                      : 402
Line  17: Illegal header file format
        "wrong.h"                       : 402
        void                            : 500
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
        printf                          : 500
        (                               : 304
        )                               : 305
        ;                               : 300
        printf                          : 500
        (                               : 304
Line  22: Unterminated string "This is a string that never terminates);

        Symbol table
========================================
         < lexeme , token >
========================================
< printf                        , 500 >
< void                          , 500 >
< main                          , 500 >
========================================

        Constants Table
========================================
         < lexeme , token >
========================================
< "This is a string"            , 403 >
========================================
NOTE: Please refer tokens.h for token meanings
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●
```

```
File  Edit  View  Search  Terminal  Help
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  ./a.out
        <stdio.h>                       : 402
        <stdlib.h>                      : 402
        int                             : 100
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
Line  22: Illegal character `
Line  23: Illegal character @
        -                               : 210
        short                           : 103
        int                             : 100
        b                               : 500
        ;                               : 300
        int                             : 100
        x                               : 500
        ,                               : 303
Line  25: Illegal identifier name 9y
        ,                               : 303
        total                           : 500
Line  25: Illegal character $
        ;                               : 300
        total                           : 500
        =                               : 209
        x                               : 500
Line  26: Illegal character @
        y                               : 500
        ;                               : 300
        printf                          : 500
        (                               : 304
        ,                               : 303
        total                           : 500
        )                               : 305
        ;                               : 300
        }                               : 302

        Symbol table
```

# Result:

The scanner design project for a C language compiler was successfully completed by following the steps outlined in the project plan. The lexical structure of the C language was analyzed, and regular expressions were written for each token. The scanner was implemented using a programming language, and a regular expression library was used to simplify the code and improve performance. The scanner was tested by running it on a set of sample input files, and the output was compared to the expected results. Errors and edge cases were handled gracefully, such as handling unterminated comments or invalid tokens.

The resulting scanner is a robust and efficient component that can be used in a larger C language compiler. It accurately identifies and tokenizes the various elements of the source code, including keywords, identifiers, operators, literals, comments, and special characters. The regular expressions used to identify each token are well-defined and easy to modify, allowing for flexibility in the design of the compiler.

In conclusion, this project provided a valuable learning experience in compiler design and implementation. The successful design of a scanner for a C language compiler demonstrates a deep understanding of the language's lexical structure and a proficiency in writing regular expressions and implementing scanners.