Lexical Analyzer for the C Language

Project - 1 (CO351)
17th January 2019

Submitted to :
Dr. P. Santhi Thilagam
Faculty, Dept of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal

Group members :
Ayush Kumar - 16CO208
Gauri Baraskar - 16CO209
Sanjana Krishnam - 16CO139

Abstract

Objective

The main goal of this project is to design a mini compiler for a subset of the C language as part of the Compiler Design Lab(CO351) course. The compiler is to be built in four phases finishing at the Intermediate Code Generation Phase. The subset of the C language chosen is to include certain data types, constructs and functions as mentioned in the specifications below. The implementation will be carried out with LEX and YACC.

Phases of the project

- Implementation of Scanner/Lexical Analyser
- Implementation of Parser
- Implementation of Semantic Checker for C language
- Intermediate Code generation for C language

Mini-compiler Specifications

The compiler is going to support the following cases :

1. **Keywords** - int, break, continue, else, for, void, goto, char, do, if, return, while
2. **Identifiers** - identified by the regular expression (_ |{letter})({letter}|{digit}| _){0,31}
3. **Comments** - single line comments (specified with // or /* … */), multi-line comments (specified with /* … */)
4. **Strings** - can identify strings mentioned in double quotes
5. **Preprocessor directives** - can identify filenames (stdio.h) after #include
6. **Data types** - int,char (supports comma declaration)
    - Syntax:
        - int a=5;
        - Char newchar = ‘a’
7. **Arrays** - int A[n]
    - Syntax:
        - int A[3]={1,2,3};
8. **Punctuators** - [ ] , <> , {} , , , : , = , ; , # , ” ” , ’ ’
9. **Operators** - arithmetic (+, - , * , /) ,increment(++) and decrement(--), assignment (=)
10. **Condition** - if else
    - Syntax:

        -
          ```
          if (condition == true){
                  //code
          }
          else{
          //code
          }
          ```

11. **Loops -**
    - Syntax
        - while(condition){
          //code
          }
        - for(initialization;condition;increment/decrement){
          //code
          }
        - while(condition){
          //code
          }
        - do{
          //code
          }while(condition);

    The loop control structures that are supported are break,continue and goto.

12. **Functions-**
    Void functions with no return type and a single parameter will be implemented
    - Syntax
        - void sample_function(int a){
          //code
          }

The mini compiler will be implemented in a straightforward fashion , using Lex and YACC tools starting off with the Scanner as the first module. If time permits we will add more features to cover a larger subset of the C language.

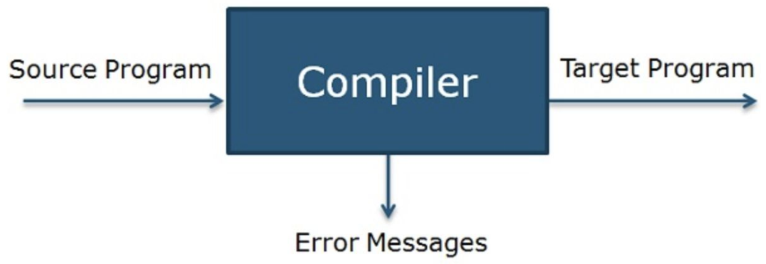
Contents

List of figures and tables :

Introduction

Compiler

A compiler is a program that can read a program in one language - the source language and translate it into an equivalent program in another language - the target language.

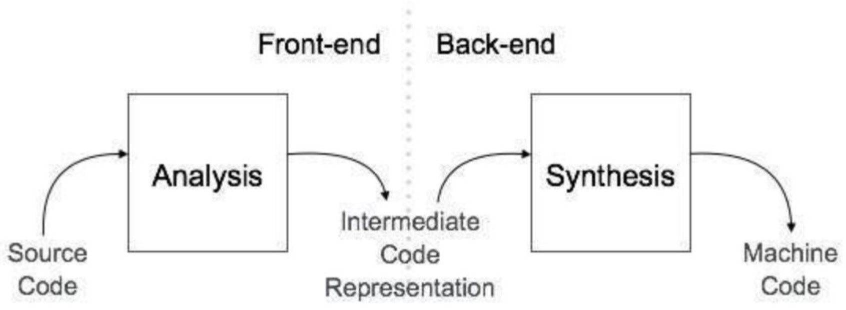


Structure of a compiler

A compiler can broadly be divided into two phases based on the way they compile i.e analysis phase (front end) and synthesis phase (back end).

The analysis phase 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. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. 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 phase 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; the synthesis part is the back end.



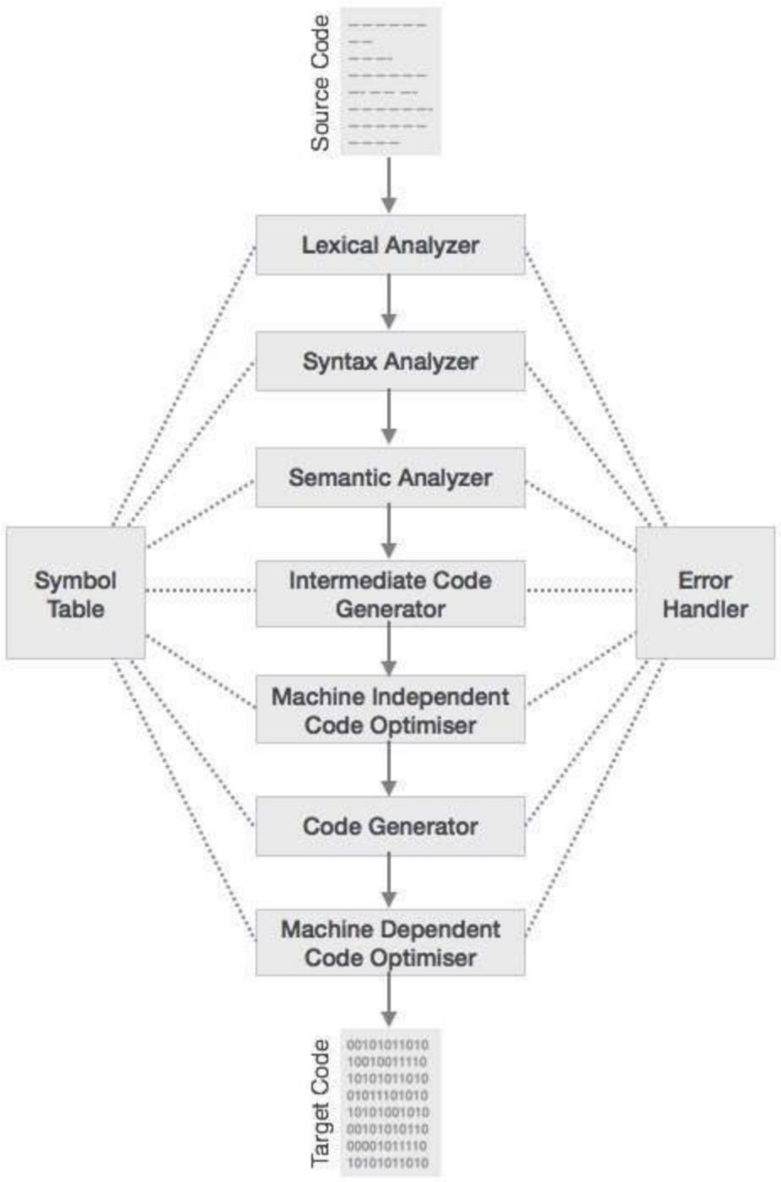
Analysis Phase – The analysis phase can be divided into three phases as follows:

1. Lexical Analyser
2. Syntax Analyser
3. Semantic Analyser

Synthesis Phase – The synthesis phase can be divided into three phases as follows:

1. Intermediate Code Generator
2. Code Optimiser
3. Code Generator

The entire process can be visualised as follows -

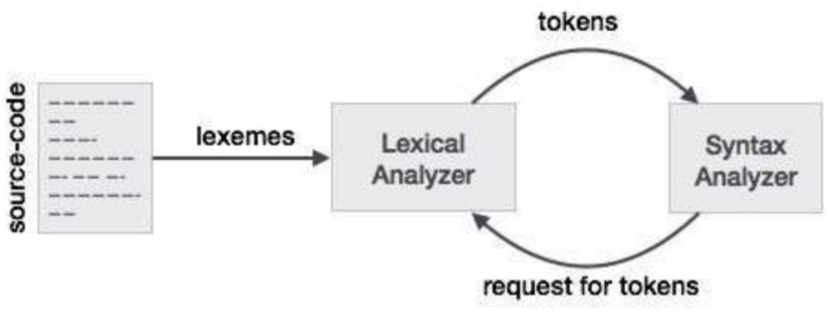


Lexical Analyser

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

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



Flex Script

We have used the lex tool to write a script to generate the Scanner or the Lexical analyser. This script has three sections as followed,

1. Definition Section

   This section is for importing any C header files, variable declarations for variables which are used later in the program etc.

2. Rules Section

   This section is for defining the rules for every token type using regular expressions, the scanner uses these rules to match lexemes in the test file to various tokens.

3. C Code Section

   This section is for C code in the script, here we call functions for setting up the symbol and the constant table along with displaying them, and calling the lexical analyser using the yylex() function.

In our script we have implemented all the functionalities targeted by our scanner.

The definition section is used for including relevant header files and initializing the variables for the symbol table and the constant table.

The rules section defines rules for all the tokens to be identified, this includes keywords, operators, identifiers, preprocessor directives, comments, strings, rules for all the tokens mentioned in the Abstract have been implemented here.

In the C code section, we have written code for specifying input file via command line arguments, we have initialized the two tables, and started the lexical analysis using the yylex() function. In the end we have displayed the symbol table and the constant table.

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 C program is fed to the flex script via command line arguments.

Our scanner has been tested with four C programs each describing valid and invalid cases for tokens, comments, strings.

Design of Programs

Code

1. Flex Script

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

    #include "tables.h"

    entry **SymbolTable = NULL;
    entry **ConstantTable = NULL;

    int ErrFlag = 0;
%}


 /* Declarations */

letter [a-zA-Z]
digit [0-9]
whitespace [ \t\r\f\v]+
identifier (_|{letter})({letter}|{digit}|_)*
hex [0-9a-fA-F]


 /* States */

%x PREPROCESSOR
%x MACROPREPROCESSOR
%x COMMENT
%x SLCOMMENT

%%


 /* Keywords */

"int"                                {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"short"                              {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"long"                               {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
```

```
"char"                                  {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"signed"                                {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"unsigned"                              {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"void"                                  {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"if"                                    {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"else"                                  {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"for"                                   {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"do"                                    {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"while"                                 {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"goto"                                  {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"break"                                 {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"continue"                              {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"main"                                  {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}
"return"                                {printf("\t%-50s : Keyword\n",
yytext);InsertEntry(SymbolTable,yytext,"KEYWORD");}


 /* Constants */

[+/-]?[0][x|X]{hex}+                    {printf("\t%-50s : Hexadecimal Constant\n", yytext);
InsertEntry(ConstantTable,yytext,"HEX_CONSTANT");}
[+/-]?{digit}+                          {printf("\t%-50s : Integer Constant\n", yytext);
InsertEntry(ConstantTable,yytext,"INT_CONSTANT");}
[+/-]?({digit}+)["."]({digit}+)         {printf("\t%-50s : Decimal Constant\n", yytext);
InsertEntry(ConstantTable,yytext,"DEC_CONSTANT");}


{identifier} {
  if(strlen(yytext) <= 32)
    {
      printf("\t%-50s : Identifier\n", yytext);
      InsertEntry(SymbolTable,yytext,"IDENTIFIER");
    }
  else
  {
      printf("Error %d: Identifier too long,must be between 1 to 32 characters\n",
yylineno);
```

```
      ErrFlag = 1;
  }
}


{digit}+({letter}|_)+                {printf("Error %d: Illegal identifier format\n",
yylineno); ErrFlag = 1;}
{whitespace}                         ;


 /* Preprocessor Directives */

^"#include"                          {BEGIN PREPROCESSOR;}
<PREPROCESSOR>{whitespace}           ;
<PREPROCESSOR>"<"[^<>\n]*">"         {printf("\t%-50s : Preprocessor directive\n", yytext);
BEGIN INITIAL;}
<PREPROCESSOR>\"[^<>\n]*\"           {printf("\t%-50s : Preprocessor directive\n", yytext);
BEGIN INITIAL;}
<PREPROCESSOR>"\n"                   {printf("Error %d: Header format not allowed\n",
yylineno); yylineno++; BEGIN INITIAL;ErrFlag=1;}
<PREPROCESSOR>.                      {printf("Error %d: Header format not allowed\n",
yylineno); ErrFlag=1;}


 /* Macropreprocessor Directives */

^"#define"                                          {BEGIN MACROPREPROCESSOR;}
<MACROPREPROCESSOR>{whitespace}                     ;
<MACROPREPROCESSOR>({letter})({letter}|{digit})*  {printf("\t%-50s : Macropreprocessor
directive\n", yytext); BEGIN INITIAL;}
<MACROPREPROCESSOR>\n                               {yylineno++; BEGIN INITIAL;}
<MACROPREPROCESSOR>.                                {printf("Error %d: Invalid definition\n",
yylineno); BEGIN INITIAL;ErrFlag=1;}


 /* Comments */

"/*"                                 {BEGIN COMMENT;}
<COMMENT>.|{whitespace}              ;
<COMMENT>\n                          {yylineno++;}
<COMMENT>"*/"                        {BEGIN INITIAL;}
<COMMENT>"/*"                        {printf("Error %d: Nested comments are invalid\n",
yylineno); ErrFlag=1;yyterminate();}
<COMMENT><<EOF>>                     {printf("Error %d: Unterminated comments are invalid\n",
yylineno);ErrFlag=1; yyterminate();}
"//"                                 {BEGIN SLCOMMENT;}
<SLCOMMENT>.                         ;
<SLCOMMENT>\n                        {yylineno++; BEGIN INITIAL;}
```

Lexical Analyzer for C Language

```
 /* Operators */

"+"                                 {printf("\t%-50s : Arithmetic Operator\n", yytext);}
"-"                                 {printf("\t%-50s : Arithmetic Operator\n", yytext);}
"*"                                 {printf("\t%-50s : Arithmetic Operator\n", yytext);}
"/"                                 {printf("\t%-50s : Arithmetic Operator\n", yytext);}
"="                                 {printf("\t%-50s : Assignment Operator\n", yytext);}
"--"                                {printf("\t%-50s : Decrement Operator\n", yytext);}
"++"                                {printf("\t%-50s : Increment Operator\n", yytext);}

">"                                 {printf("\t%-50s : Comparison Operator\n", yytext);}
"<"                                 {printf("\t%-50s : Comparison Operator\n", yytext);}
">="                                {printf("\t%-50s : Comparison Operator\n", yytext);}
"<="                                {printf("\t%-50s : Comparison Operator\n", yytext);}
"=="                                {printf("\t%-50s : Comparison Operator\n", yytext);}

"||"                                {printf("\t%-50s : Boolean Operator\n", yytext);}
"&&"                                {printf("\t%-50s : Boolean Operator\n", yytext);}
"!"                                 {printf("\t%-50s : Boolean Operator\n", yytext);}


 /* Strings and Characters */

\"[^\"\n]*$                         {printf("Error %d: Illegally terminated string\n",
yylineno);ErrFlag=1; yyterminate();}
\"[^\"\n]*\" {
  if(yytext[yyleng-2]=='\\') {
    yyless(yyleng-1);
    yymore();
  }
  else
  {
    InsertEntry(ConstantTable,yytext,"STRING");
    printf("\t%-50s : String\n", yytext);
  }
}

\'[^\'\n]\'                         {printf("\t%-50s : Character\n", yytext);}


 /* Punctuators */


"["                                 {printf("\t%-50s : Open Square Bracket\n", yytext);}
"]"                                 {printf("\t%-50s : Closed Square Bracket\n", yytext);}
"{"                                 {printf("\t%-50s : Open Curly Bracket\n", yytext);}
"}"                                 {printf("\t%-50s : Closed Curly Bracket\n", yytext);}
"("                                 {printf("\t%-50s : Open Round Bracket\n", yytext);}
")"                                 {printf("\t%-50s : Closed Round Bracket\n", yytext);}
","                                 {printf("\t%-50s : Comma\n", yytext);}
```

```
";"                                 {printf("\t%-50s : Delimiter\n", yytext);}


"\n"                                {yylineno++;}
.                                   {printf("Error %d: Illegal character\n", 
yylineno);ErrFlag=1;}


%%


int main(int argc, char *argv[])
{
  SymbolTable = CreateTable();
  ConstantTable = CreateTable();
  FILE *fh;
  if (argc == 2 && (fh = fopen(argv[1], "r")))
        yyin = fh;
  yylex();

  if (ErrFlag == 0)
  {
    printf("\nPrinting Symbol Table\n" );
    Display(SymbolTable);
    printf("\n\n");
    printf("Printing Constant Table\n" );
    Display(ConstantTable);
  }

  printf("Lexical analysis finished\n");
  return 0;
}

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

The source code for the lexical analyser is in the above script.

In the definition section we have the declarations for our tables and include commands for the necessary files.

The rules section has the regular expressions for all the token classes and the C code section is used for obtaining the input file and for initialising and displaying the symbol and constant tables.

This script outputs the two tables only if no errors were obtained in the analysis, otherwise it shows the respective errors.

2. Source Code for Symbol and Constant Tables

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

#define SIZE 100


struct table_entry{

    char *lexeme;
    char  *token;
    struct table_entry *next;
};

typedef struct table_entry entry;

int hash(char *lexeme)
{
  int hash = 0,i=0;

  for(i=0; i < strlen(lexeme); i++)
    hash += lexeme[i];

  return hash % SIZE;
}

entry** CreateTable()
{
  entry **TablePointer = NULL;

  TablePointer = malloc(sizeof(entry*)*SIZE);

  if(TablePointer == NULL)
    return NULL;

  int i;
  for(i=0;i<SIZE;i++)
    TablePointer[i] = NULL;

  return TablePointer;
}

int Search(entry** TablePointer, char *lexeme)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0)
      return 1;
    else
      head = head->next;
  }
  if(head == NULL)
    return 0;
  return 0;
}

void InsertEntry(entry** TablePointer, char *lexeme,char *Token )
{
  if(Search(TablePointer,lexeme) == 1)
    return;
  else
  {
    int temp = hash(lexeme);
    entry *head = NULL;

    head = TablePointer[temp];

    entry *tempPoint = NULL;
    tempPoint = malloc(sizeof(entry));
    tempPoint->lexeme = strdup(lexeme);
    tempPoint->token = strdup(Token);
    tempPoint->next = NULL;

    if (head == NULL)
    {
      TablePointer[temp] = tempPoint;
    }
    else
    {
      tempPoint->next = TablePointer[temp];
      TablePointer[temp] = tempPoint;
    }

  }
}

void Display(entry** TablePointer)
{
  int i =0;
  entry *temp = NULL;

  printf("\n\n");
  printf("-----------------------------------------\n");

  printf("\n\t(lexeme, token)\n" );

  for(i=0;i<SIZE;i++)
  {
    temp = TablePointer[i];
    while(temp != NULL)
    {
      printf("\t(%5s, %s)\n",temp->lexeme,temp->token);
      temp = temp->next;
    }

  }

  printf("-----------------------------------------\n");
}
```

This file implements all functions involved in creating and managing the symbol table and the constant table.

The tables are implemented as hash tables, with their entries being structs with three fields, one for the lexeme, one for the token name, and one for the next entry, the next entry field is used for open chaining in case of hash conflicts.

The hash function used is a simple modular function which adds the ascii value of all the characters and performs **SUM % 100** operation. Conflicts at the same position are handled by chaining the new entry to the existing entry like a linked list.

The search function is a utility function used by InsertEntry function to put new entries in the hash table. If an entry already exists then the insert operation is aborted.

Display function is used to display the two different tables at the end of lexical analysis phase.

The CreateTable function initializes an empty hash table which is used to enter data.

Functionality and Implementation

The below section explains how the code identifies different keywords , comments , operators, punctuators and preprocessors using rules described in the lex file. All these rules are in the form of regular expressions. All lexemes should match at least one of the regex values else the lexeme is said to not belong in the language and such a file should throw a lexical error.

Keywords

The keywords identified are - **int , short , long , char , signed , unsigned , void , if , else , for , do , while , goto, break, continue , main , return**.

Comments

The scanner in this implementation removes all the valid comments from the C file. We make use of two exclusive states in lex as <SLCOMMENT> and <COMMENT> for single line and multi line comments respectively.

- When we encounter a “//” we start the state <SLCOMMENT> by the command BEGIN SLCOMMENT. Now , when we are this state we match all characters after “//” except new line because this is a single line comment. This is done using “.” which matches all characters except a “\n”. The state can be exited only when we encounter a “\n” which is implemented using BEGIN INITIAL.
- For multi-line comments , we match the pattern “/*” and enter the state <COMMENT>.
- We continue to remain in this state as long as we do not encounter a “*/” pattern. This is matched using (. | ”\n”)* .The state can be exited only when another “*/” pattern is encountered.
- The lexical analyser in this implementation does not support nested comments hence if the scanner encounters “/*” inside <COMMENT> stage it returns an error saying “Nested comments are invalid. “.
- The scanner also accounts for unterminated comments. If the scanner is still the <COMMENT> stage when the EOF if reached , it throws an exception saying “Unterminated comments are invalid.” . The EOF is matched using the regex character <<EOF>>.

Identifiers

- The identifiers are identified using the regex **(_ | { letter })({ letter } | { digit }| _)*** . The { letter } set stands for the character set [A-Za-z] and { digit } for [0-9].
- The rules only identifies lexemes which start either with an underscore or a letter as specified in the C language. The identifier name can then be followed by either a digit , letter or an underscore.
- The identifiers starting with a letter should not be accepted which is implemented using the regex **{ digit } + ({ letter } | _)+**. Error is displayed for the same.
- The length of the identifier cannot be more than 32. So , when the analyser detects an identifier it checks if the length is less than 32. This is achieved using the condition strlen(yytext) < 32. If the length is greater than 32 , the analyser shows an error stating “The identifier length should be between 1 and 32.” .

Preprocessors

- To identify the preprocessors we make use of an exclusive state <PREPROCESSOR>. The preprocessors have a strict syntax that starts with “#include”. Therefore , the scanner enters the state <PREPROCESSOR> when a “#include” is matched.
- The preprocessors can be declared in two ways - <stdio.h> and “stdio.h”. To match with the former pattern we use the pattern **"<"[^<>\n]*">"** and for the latter way we use **\"[^<>\n]*\" .**
- Any other format is unacceptable and is taken care by matching “\n” and all other characters. Corresponding relevant errors are shown if unacceptable forms are encountered.

Macro Preprocessors

- We make use of an exclusive state <MACROPREPROCESSOR>. The scanner enters this state when “#define” pattern is encountered.
- While in this state , the valid definitions are identified using the pattern

  ({ letter })({ letter } | { digit })*.

Constants

- The implementation accounts for integer , decimal and hexadecimal constants. Here { digit } stands for the set [0 - 9]
- Integer constants are identified using the following regular expression

  [+ / -]? { digit }+

  The ‘ + ’ or ‘ - ’ character is used so that it can match both positive and negative characters and the wildcard symbol is used to account for the case when positive integers don’t have any sign.

- Hexadecimal constants are identified using the following regular expression

  [+ / -]? [0] [x | X] { hex }+

  Here { hex } stands for the character set [0-9a-fA-F], as hexadecimal constants can use all of these characters. [0] [x | X] is used because hexadecimal characters in C always begin with ‘ 0x ’. [+ / -]? Is used to account for all types of positive and negative hexadecimal constants.

- Decimal constants are identified using the following regular expression

  [+ / -]? ({ digit }+) [" . "] ({ digit }+)

  The [“ . “] is used to identify the decimal constant with digits at both end, both positive and negative constants are identified using the [+ / -]?.

Strings and Characters

- Strings are identified using the following regular expression:

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

  Characters are identified using a similar expression but with single quotes

  \' [^ \' \n] \'

- In case of strings we also have to take care of escaped characters, here we have taken care of escaped quotes using the yyless and yymore functions. The escaped quote is ignored and the next token is appended to it. Thus resulting in proper recognition of string.

Operators and Punctuators

- Arithmetic Operators (+,-,/,*) are identified verbatim by matching using “+”, “-” etc. in the code. The same thing is carried forward for all the other operators.
- Punctuators (.,; { } <>) are also identified individually in the same format as the operators.

Test Cases

Without Errors:

Serial No	Test Case	Expected Output	Status
1.	#include<stdio.h>	<stdio.h> : Preprocessor directive	PASS
2.	#include “stdlib.h”	“stdlib.h” : Preprocessor directive	PASS
3.	int a;	int : Keyword a : Identifier	PASS
4.	int f = 0xAB;	int : Keyword f : Identifier 0xAB : Hexadecimal Constant	PASS
5.	e = f+g;	e : Identifier f : Identifier + : Arithmetic operator g : Identifier	PASS

6.	if(c=='A')
	a = 10;
	else
	a = 30;
		If : Keyword
		(: Open Round Bracket
		C : Identifier
		== : Comparison Operator
		‘A’ : Character
		) : Closed Round Bracket
		a : Identifier
		= : Assignment Operator
		10 : Integer Constant
		; : Delimiter
		else : Keyword
		A : Identifier
		= : Assignment Operator
		30 : Integer Constant
		; : Delimiter
			PASS

7.	for (a = 0; a<3; a++){
	
	}
		for : Keyword
		(: Open Round Bracket
		a : Identifier
		= : Assignment Operator
		0 : Integer Constant
		; : Delimiter
		a : Identifier
		< : Comparison Operator
		3 : Integer Constant
		; : Delimiter
		a : Identifier
		++ : Increment Operator
		) : Closed Round Bracket
		{ : Open Curly Bracket
		} : Closed Curly Bracket
		} : Closed Curly Bracket
			PASS

With Errors

Serial No	Test Case	Expected Output	Status
1.	#include	Header format not allowed	PASS
2.	int abcdefghijklmnopqrstuvwxyzabcdefgh;	Identifier too long, must be between 1 to 32 characters	PASS
3.	int 2invalid;	Illegal identifier format	PASS
4.	Y = x$x;	Illegal character	PASS
5.	/* Nested comments /*are not*/ allowed */	Nested comments are not allowed	PASS
6.	printf("This string does not terminate);	Illegally terminated string	PASS

Results and Future Work

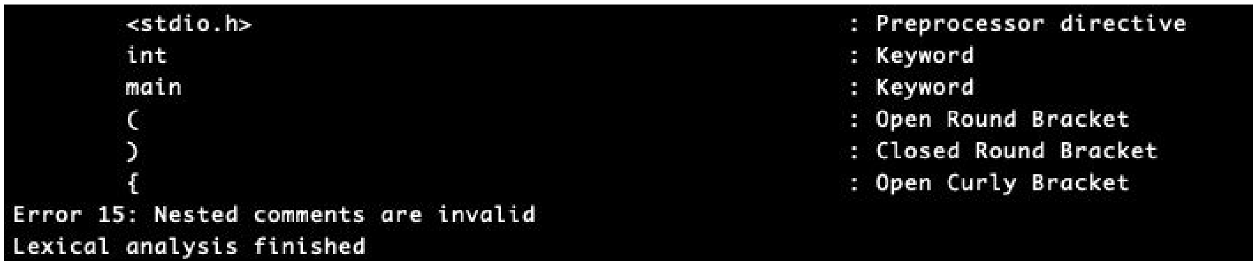
Results

The various C input files used for testing the scanner have been included below, the results obtained on running the tests has also been added.

1. Input File: test-case-comments.c

```
/* The lexical analyser should remove all the valid comments and throw an exception for all
the invalid comments */

#include<stdio.h>

int main(){

       // Single line comment

       /* Single line comment with different beginning */

       /* Multi-line comment

       should be removed by lex */

       /* Nested comments are /* not */ allowed */

  return 0;

}
```

Test Results:



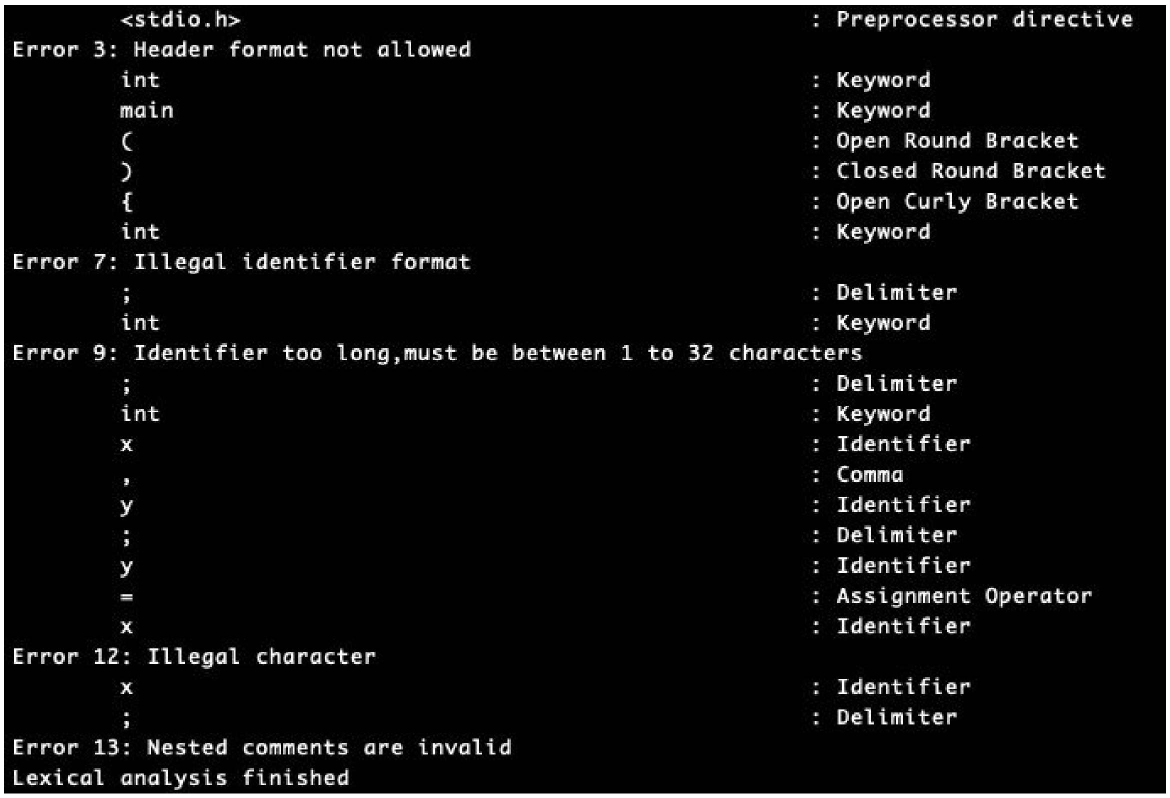
```
        <stdio.h>                                         : Preprocessor directive
        int                                               : Keyword
        main                                              : Keyword
        (                                                 : Open Round Bracket
        )                                                 : Closed Round Bracket
        {                                                 : Open Curly Bracket
Error 15: Nested comments are invalid
Lexical analysis finished
```

As seen in the results above, all the comments are ignored by lex, and all tokens are identified correctly. Line 15, has a nested comment, which isn't allowed by the compiler and hence lex throws an error, because an error occurred during analysis, lex doesn't print the symbol table and the constant table.

2. Input File: test-case-errors.c

```
#include<stdio.h>
//Illegal header format
#include

int main(){
  //Illegal identifier
  int 2invalid;
  //Identifier too long
  int abcdefghijklmnopqrstuvwxyzabcdefgh;
  int x , y;
  //Illegal character
  y = x $ x;
  /* Nested comments /*are not*/ allowed */
  return 0;
}
```

Test Results:



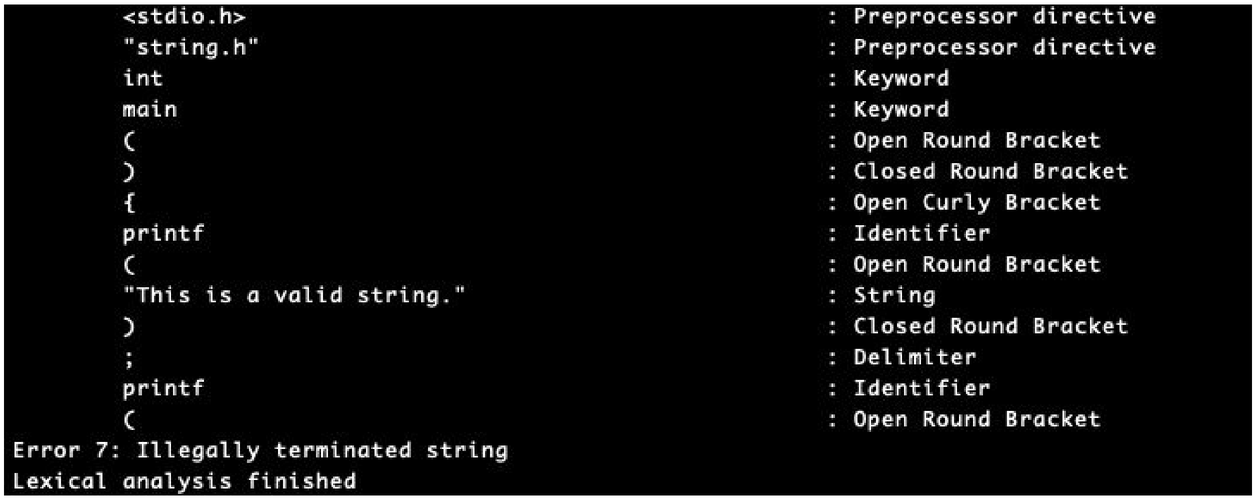
```
        <stdio.h>                                         : Preprocessor directive
Error 3: Header format not allowed
        int                                               : Keyword
        main                                              : Keyword
        (                                                 : Open Round Bracket
        )                                                 : Closed Round Bracket
        {                                                 : Open Curly Bracket
        int                                               : Keyword
Error 7: Illegal identifier format
        ;                                                 : Delimiter
        int                                               : Keyword
Error 9: Identifier too long,must be between 1 to 32 characters
        ;                                                 : Delimiter
        int                                               : Keyword
        x                                                 : Identifier
        ,                                                 : Comma
        y                                                 : Identifier
        ;                                                 : Delimiter
        y                                                 : Identifier
        =                                                 : Assignment Operator
        x                                                 : Identifier
Error 12: Illegal character
        x                                                 : Identifier
        ;                                                 : Delimiter
Error 13: Nested comments are invalid
Lexical analysis finished
```

Here we can notice that, illegally named identifier with a digit at the beginning throws an error. When the identifier name is more than 32 characters, another error is shown. Illegal characters are also recognised, and nested comment is not allowed. No Symbol Table or Constant Table is printed because of the errors encountered.

3. Input File: test-case-strings.c

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

int main(){

        printf("This is a valid string.");
        printf("This string does not terminate);
        return 0;
}
```

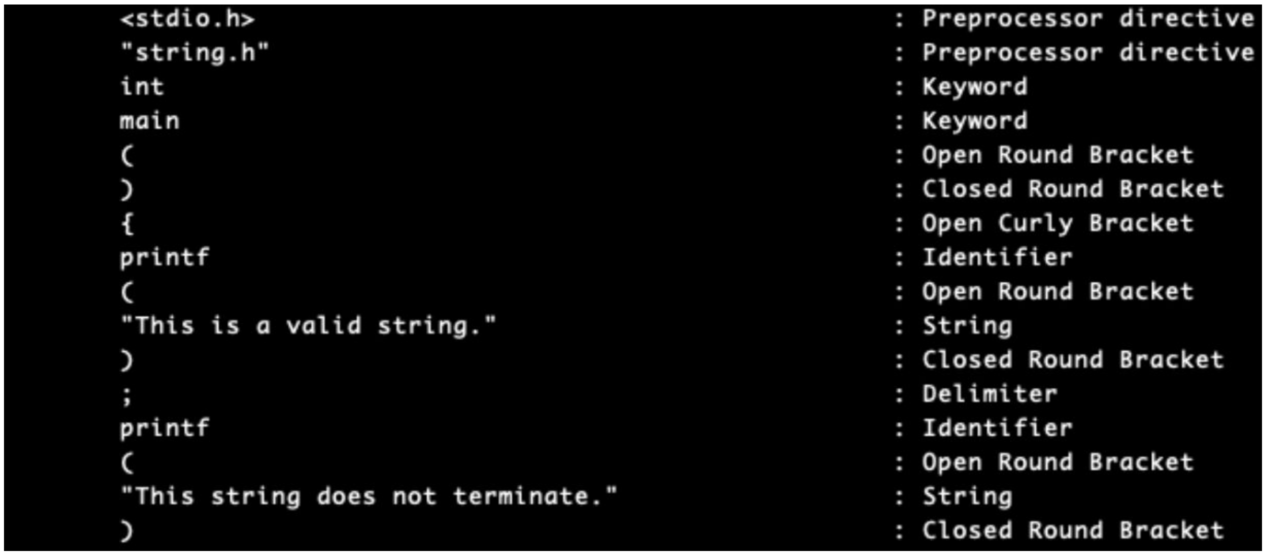
Test Results:



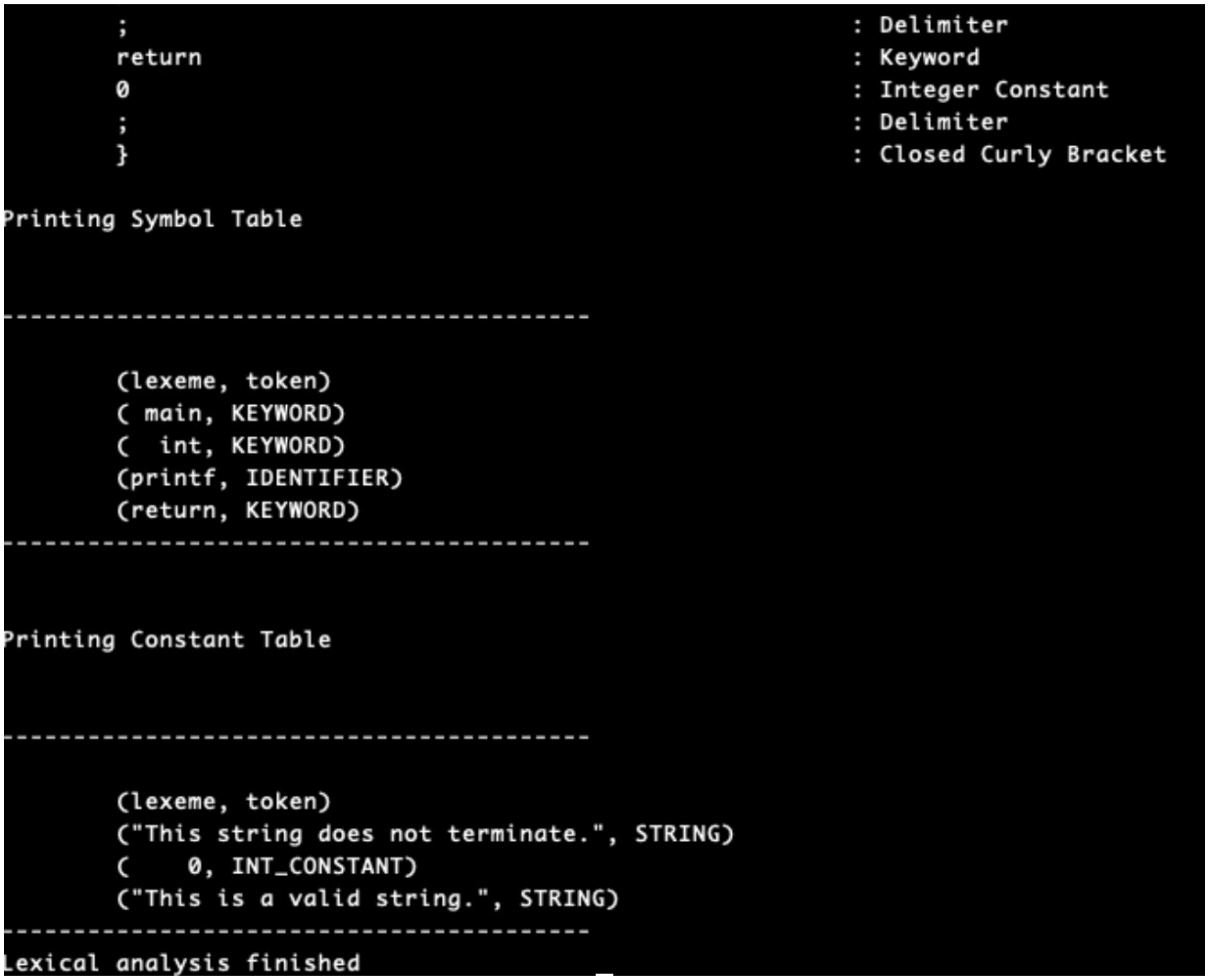
```
        <stdio.h>                                         : Preprocessor directive
        "string.h"                                        : Preprocessor directive
        int                                               : Keyword
        main                                              : Keyword
        (                                                 : Open Round Bracket
        )                                                 : Closed Round Bracket
        {                                                 : Open Curly Bracket
        printf                                            : Identifier
        (                                                 : Open Round Bracket
        "This is a valid string."                         : String
        )                                                 : Closed Round Bracket
        ;                                                 : Delimiter
        printf                                            : Identifier
        (                                                 : Open Round Bracket
Error 7: Illegally terminated string
Lexical analysis finished
```

We can see that when a string which isn't terminated is encountered the lex program throws an error. If there was no illegally terminated code then the results will include the strings in the constant table and print both at the end of lexical analysis.

The result for test case without illegal strings has been added below.



```
       <stdio.h>                                    : Preprocessor directive
       "string.h"                                   : Preprocessor directive
       int                                          : Keyword
       main                                         : Keyword
       (                                            : Open Round Bracket
       )                                            : Closed Round Bracket
       {                                            : Open Curly Bracket
       printf                                       : Identifier
       (                                            : Open Round Bracket
       "This is a valid string."                    : String
       )                                            : Closed Round Bracket
       ;                                            : Delimiter
       printf                                       : Identifier
       (                                            : Open Round Bracket
       "This string does not terminate."            : String
       )                                            : Closed Round Bracket
```



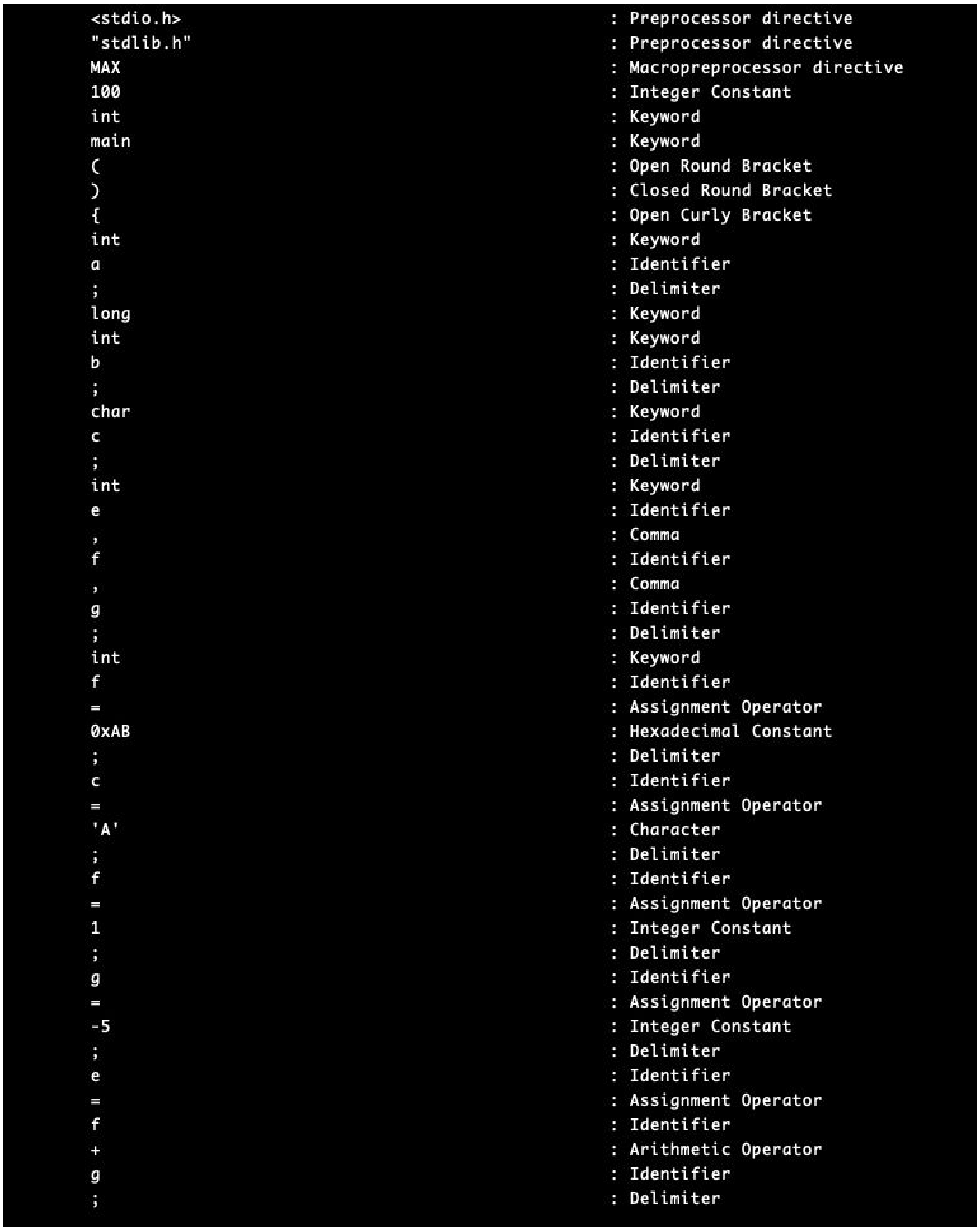
```
        ;                                                          : Delimiter
        return                                                     : Keyword
        0                                                          : Integer Constant
        ;                                                          : Delimiter
        }                                                          : Closed Curly Bracket

Printing Symbol Table


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


        (lexeme, token)
        ( main, KEYWORD)
        (  int, KEYWORD)
        (printf, IDENTIFIER)
        (return, KEYWORD)
-----------------------------------------


Printing Constant Table


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


        (lexeme, token)
        ("This string does not terminate.", STRING)
        (    0, INT_CONSTANT)
        ("This is a valid string.", STRING)
-----------------------------------------
Lexical analysis finished
```

4. Input File : test-case-tokens.c

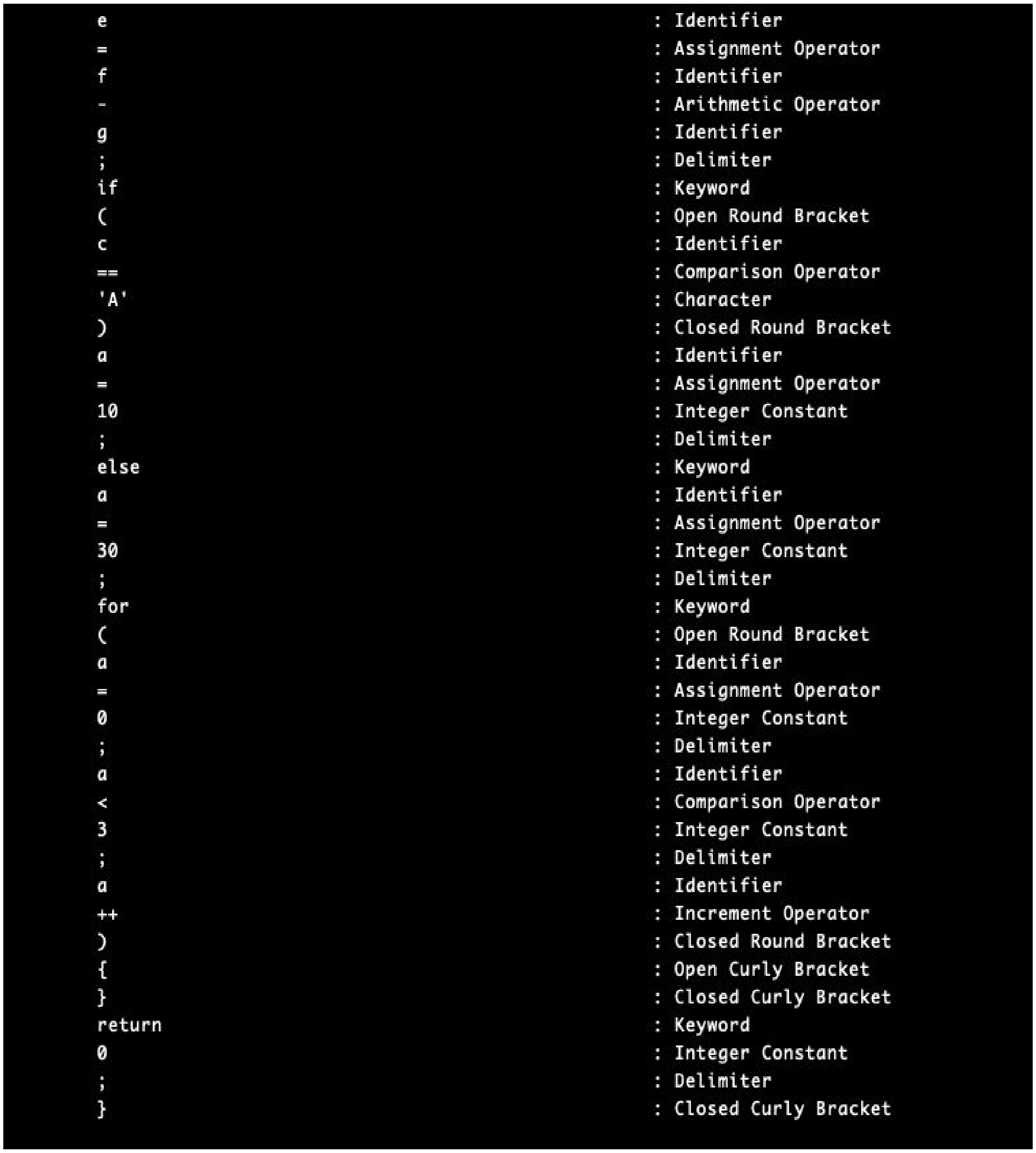
```
/* This file tests the detection of allowed keywords , identifiers and other tokens such as
punctuators , operators */

//Identifies preprocessor directives
#include<stdio.h>
#include "stdlib.h"
//Identifies macro preprocessor directives
#define MAX 100

int main(){

        //Identifies keywords int , long , char , if , else and operators
        int a;
        long int b;
        char c;
        int e,f,g;

        //Identifies constants
        int f = 0xAB;
        c = 'A';
        f = 1;
        g = -5;

        //Identifies arithmetic operators
        e = f+g;
        e = f-g;

        // Identifies conditions
        if(c=='A')
                a = 10;
        else
                a = 30;

        // Identifies loops

        for ( a = 0; a<3; a++){
        }

        return 0;
}
```

Lexical Analyzer for C Language

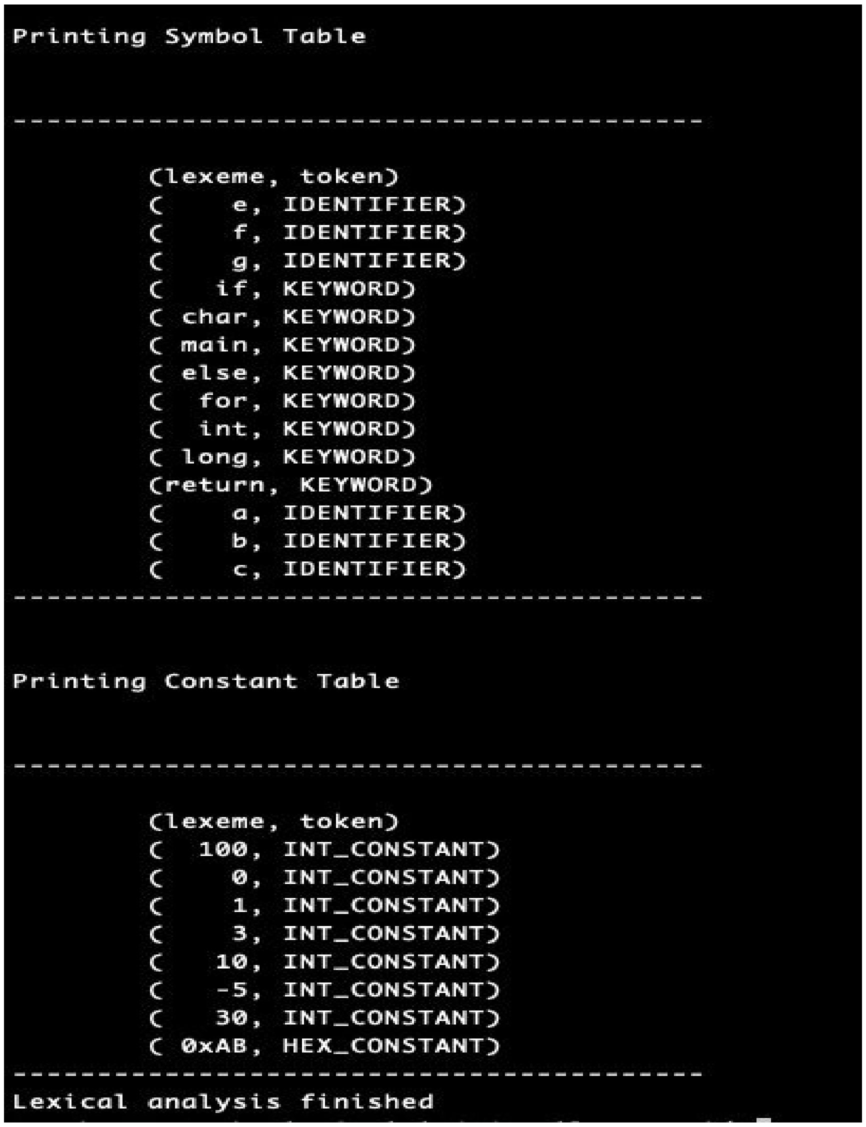
Test Results:



```
<stdio.h>                                   : Preprocessor directive
"stdlib.h"                                  : Preprocessor directive
MAX                                         : Macropreprocessor directive
100                                         : Integer Constant
int                                         : Keyword
main                                        : Keyword
(                                           : Open Round Bracket
)                                           : Closed Round Bracket
{                                           : Open Curly Bracket
int                                         : Keyword
a                                           : Identifier
;                                           : Delimiter
long                                        : Keyword
int                                         : Keyword
b                                           : Identifier
;                                           : Delimiter
char                                        : Keyword
c                                           : Identifier
;                                           : Delimiter
int                                         : Keyword
e                                           : Identifier
,                                           : Comma
f                                           : Identifier
,                                           : Comma
g                                           : Identifier
;                                           : Delimiter
int                                         : Keyword
f                                           : Identifier
=                                           : Assignment Operator
0xAB                                        : Hexadecimal Constant
;                                           : Delimiter
c                                           : Identifier
=                                           : Assignment Operator
'A'                                         : Character
;                                           : Delimiter
f                                           : Identifier
=                                           : Assignment Operator
1                                           : Integer Constant
;                                           : Delimiter
g                                           : Identifier
=                                           : Assignment Operator
-5                                          : Integer Constant
;                                           : Delimiter
e                                           : Identifier
=                                           : Assignment Operator
f                                           : Identifier
+                                           : Arithmetic Operator
g                                           : Identifier
;                                           : Delimiter
```



```
e                                                  : Identifier
=                                                  : Assignment Operator
f                                                  : Identifier
-                                                  : Arithmetic Operator
g                                                  : Identifier
;                                                  : Delimiter
if                                                 : Keyword
(                                                  : Open Round Bracket
c                                                  : Identifier
==                                                 : Comparison Operator
'A'                                                : Character
)                                                  : Closed Round Bracket
a                                                  : Identifier
=                                                  : Assignment Operator
10                                                 : Integer Constant
;                                                  : Delimiter
else                                               : Keyword
a                                                  : Identifier
=                                                  : Assignment Operator
30                                                 : Integer Constant
;                                                  : Delimiter
for                                                : Keyword
(                                                  : Open Round Bracket
a                                                  : Identifier
=                                                  : Assignment Operator
0                                                  : Integer Constant
;                                                  : Delimiter
a                                                  : Identifier
<                                                  : Comparison Operator
3                                                  : Integer Constant
;                                                  : Delimiter
a                                                  : Identifier
++                                                 : Increment Operator
)                                                  : Closed Round Bracket
{                                                  : Open Curly Bracket
}                                                  : Closed Curly Bracket
return                                             : Keyword
0                                                  : Integer Constant
;                                                  : Delimiter
}                                                  : Closed Curly Bracket
```



```
Printing Symbol Table


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

        (lexeme, token)
        (     e, IDENTIFIER)
        (     f, IDENTIFIER)
        (     g, IDENTIFIER)
        (    if, KEYWORD)
        (  char, KEYWORD)
        (  main, KEYWORD)
        (  else, KEYWORD)
        (   for, KEYWORD)
        (   int, KEYWORD)
        (  long, KEYWORD)
        (return, KEYWORD)
        (     a, IDENTIFIER)
        (     b, IDENTIFIER)
        (     c, IDENTIFIER)
-----------------------------------------


Printing Constant Table


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

        (lexeme, token)
        (   100, INT_CONSTANT)
        (     0, INT_CONSTANT)
        (     1, INT_CONSTANT)
        (     3, INT_CONSTANT)
        (    10, INT_CONSTANT)
        (    -5, INT_CONSTANT)
        (    30, INT_CONSTANT)
        (  0xAB, HEX_CONSTANT)
-----------------------------------------
Lexical analysis finished
```

All the tokens are identified correctly and the identifiers, keywords and various constants are added to Symbol table and Constant table based on their types. Both the tables are displayed at the end of the analysis.

5. Input File : Binary Search

```
#include <stdio.h>

int main()
{
   int c, first, last, middle, n, search, array[100];

   printf("Enter number of elements\n");
   scanf("%d",&n);

   printf("Enter %d integers\n", n);

   for (c = 0; c < n; c++)
      scanf("%d",&array[c]);

   printf("Enter value to find\n");
   scanf("%d", &search);

   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;
}
```

Test Results:

```
Printing Symbol Table


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

        (lexeme, token)
        (   if, KEYWORD)
        (    n, IDENTIFIER)
        (break, KEYWORD)
        ( main, KEYWORD)
        (scanf, IDENTIFIER)
        (middle, IDENTIFIER)
        ( else, KEYWORD)
        (  for, KEYWORD)
        (search, IDENTIFIER)
        (  int, KEYWORD)
        ( last, IDENTIFIER)
        (while, KEYWORD)
        (array, IDENTIFIER)
        (first, IDENTIFIER)
        (printf, IDENTIFIER)
        (return, KEYWORD)
        (    c, IDENTIFIER)
-----------------------------------------


Printing Constant Table


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

        (lexeme, token)
        ( "%d", STRING)
        ("%d found at location %d.\n", STRING)
        (  100, INT_CONSTANT)
        ("Enter %d integers\n", STRING)
        (    0, INT_CONSTANT)
        (    1, INT_CONSTANT)
        ("Enter value to find\n", STRING)
        ("Not found! %d isn't present in the list.\n", STRING)
        (   +1, INT_CONSTANT)
        (   /2, INT_CONSTANT)
        ("Enter number of elements\n", STRING)
-----------------------------------------
Lexical analysis finished
```

Due to the large size of the lexical analysis results, only the tables have been printed.

Future Work

The script used by us covers a subset of the C language. It works for all the functionalities targeted in the abstract. The future work in this project will involve using this script with the parser, where the scanner will return the tokens to the parser for the next stage of the compiler. Further work can also be done on the lexical analyser itself ,so that it can cover a larger subset of the C language.

References

1. Aho A.V, Sethi R, and Ullman J.D. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986
2. http://www.di.univr.it/documenti/OccorrenzaIns/matdid/matdid065185.pdf
3. http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf