NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA,
SURATHKAL

COMPILER LAB PROJECT-1

LEXICAL ANALYZER USING FLEX

Submitted By:

Chirag Jamadagni (13CO117)
Abhijith Anilkumar (13CO102)
George CM (13CO119)

COMPILER LAB

Phase 1 - Lexical Analyzer

Abstract

A compiler, in general, is a computer program that transforms source code written in a programming language into another computer language. A compiler has many parts like the Lexical Analyzer, Parser, Semantic checker, Intermediate-code generator, Code optimizer and Code generator.

The Lexical analyzer of the compiler is the part which identifies each entry in the source program and differentiate them into lexemes of different tokens. It uses regular expression for achieving this task. Apart from identifying the token it also reports a few lexical errors.

This project makes a Lexical analyzer for a subset of C language using the flex tool. The flex tool allows users to create a scanner, with C language as the base and provides the provision for specifying the regular expression associated with each token. It then by default creates the DFA associated with it and identifies and returns the token as the output.

TABLE OF CONTENTS

Title	Page Number
Introduction	5
Design	9
Test Cases	13
Implementation	16
Results	17
Examples	18
Future Work	24
References	24

List of Tables

Title	Page No.
Test Cases	13

List of Figures

Title	Page No.
Scanner code - 1	9
Scanner code - 2	10
Scanner code - 3	10
Scanner code - 4	11
Scanner code - 5	11
Scanner code - 6	12
Run file (Shell Script)	12
Example 1 input	18
Example 1 output	18

Example 2 input	19
Example 2 output	19
Example 3 input	20
Example 3 output	20
Example 4 input	21
Example 4 output	21
Example 5 input	22
Example 5 output	23

Introduction

This project involves building a compiler. There are four phases which leads to the development of the following parts: Lexical Analyzer, Parser, Semantic Checker and Intermediate Code Generator.

Lexical Analyzer

Lexical Analyzer is a component of a compiler that goes through the source program directly and identifies tokens.

The main task of Lexical Analyzer is to read a stream of characters as an input and produce a sequence of tokens such as names, keywords, punctuation marks etc. for syntax analyzer. It discards the white spaces and comments between the tokens and also keep track of line numbers. It reads a lexeme and identifies which token it belongs to. It also makes an entry into the symbol table

Lexical Analyzer also generates errors in the following cases:

- **Unidentified token:** When the lexeme does not match any of the specified regular expressions.
- **Unterminated String:** When the right number of inverted commas are not provided.
- **Nested Comments:** Nested comments are not supported.
- **Nested Strings:** Nestes strings are also not supported.
- **Unmatched Parenthesis:** If there are missing parenthesis, an error message is generated.

Flex Script

Flex - Fast Lexical Analyzer Generator

It is a tool for generating programs that perform pattern-matching on text.

Flex is basically a tool for generating **scanners**: programs which recognized lexical patterns in text. Flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called **rules**. flex generates as output a C source file, `lex.yy.c', which defines a routine `yylex()'. This file is compiled to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

These programs perform character parsing and tokenizing via the use of a deterministic finite automaton (DFA). A DFA is a theoretical machine accepting regular languages. These machines are a subset of the collection of Turing machines. DFAs are equivalent to read-only right moving Turing machines. The syntax is based on the use of regular expressions. See also nondeterministic finite automaton.

Format of the Input File

The flex input file consists of three sections, separated by a line with just `%%' in it:

definitions
%%
rules
%%
user code

The **definitions** section contains declarations of simple **name** definitions to simplify the scanner specification, and declarations of **start condition**.

The **rules** section of the flex input contains a series of rules of the form:

Pattern action: Where the pattern must be unindented and the action must begin on the same line. The patterns in the input are written using an extended set of regular expressions.

Finally, the **user code** section is simply copied to `lex.yy.c' verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too.

Phase 1 – Lexical Analyzer

In the definitions and rules sections, any indented text or text enclosed in `%{' and `%}' is copied verbatim to the output (with the `%{}''s removed). The `%{}''s must appear unindented on lines by themselves.

In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors.

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

C Program

This project aims to create a compiler for subset of C language. C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972. In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard.

C is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. By design, C provides constructs that map efficiently to typical machine instructions, and therefore it has found lasting use in applications that had formerly been coded in assembly language, including operating systems, as well as various application software for computers ranging from supercomputers to embedded systems.

Since this compiler is built for C language, all testing has been done using C programs. Special C programs are written as test cases for testing the working of the compiler.

Design

Scanner.l

Explanation:

This code is the scanner which will be converted to lex.yy.cc. It contains regular expressions for classifying the various lexemes to tokens and also defines the actions that should be taken for the various cases.

Code:

```
1  /* Scanner Program */
2
3   /* DEFINITIONS */
4  D            [0-9]
5  L            [a-zA-Z_]
6  H            [a-fA-F0-9]
7  E            [Ee][+-]?{D}+
8  FS           (f|F|l|L)
9  IS           (u|U|l|L)*
10
11 %{
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <malloc.h>
15 #include <string.h>
16 int found=0,var_cnt=0,i,nestedComment=0,bracCount=0,lineCount=0,flag=0;
17 char *comment;
18 void add_to_comment(char*);
19 void insert(char *yytext,char type);
20 struct holder
21 {
22      char *name;
23      int val;
24      struct holder *next;
25 }*st,*head;
26
27 %}
28 string L?\"(\\.|[^\\"])*\"
29 printf (printf\({string}(\,([^,)])+)*\))
30 prnterr (printf\(.*\))
31 scanf (scanf\({string}(\,([^,)])+)*\))
32 scanerr (scanf\(.*\))
33 comment (\/\/.*)
34 comstr (\/\*)
35 comend (\*\/)
36 keword "auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|
   "signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"|"void"|"volatile"|"while"
37 relop >|<|<=|>=|!=
38 ws [ \t]+
39 %x C_COMMENT
40
41 %%
42
43  /* RULES */
44 ^#([-a-zA-Z0-9.]|{relop}|{ws})* {insert(yytext,'d');}
45 {printf}            {;}
46 {prnterr}           {printf("ERROR: printferror\n");}
47 {scanf}             {;}
                                                                                    1,144         Top
```

Figure 1

Phase 1 – Lexical Analyzer

```
{scanerr}           {printf("ERROR: scanferror\n");}
{comment}           ;

 /*RegEx for Comments*/

{comstr}            {  BEGIN(C_COMMENT);
                       nestedComment++;
                       lineCount++;
                       add_to_comment("\n");
                    }
<C_COMMENT>{comstr} { nestedComment++;
                      if(nestedComment>1)
                      {
                        printf("ERROR: Nested Comment\n");
                        flag = 1;
                      }
                    }
<C_COMMENT>{comend} {
                        if(nestedComment>0)
                                nestedComment--;
                        else
                            printf("ERROR: */ found before /*\n");
                        if(nestedComment==0)
                            BEGIN(INITIAL);
                    }
<C_COMMENT>\n {lineCount++; add_to_comment("\n");}
<C_COMMENT>. {add_to_comment(yytext);}

 /*RegEx for Keywords*/

{keyword}  {if(nestedComment<=0)
              insert(yytext,'k');
            }

 /*RegEx for Variables*/

{L}({L}|{D})*       {  if(nestedComment<=0) insert(yytext,'v'); }

 /*RegEx for Constants*/

0[xX]{H}+{IS}?      {if(nestedComment<=0) insert(yytext,'c');}
0{D}+{IS}?      {if(nestedComment<=0) insert(yytext,'c');}
{D}+{IS}?       {if(nestedComment<=0) insert(yytext,'c');}
L?'(\\.|[^\\'])+'   {if(nestedComment<=0) insert(yytext,'c');}
{D}+{E}{FS}?        {if(nestedComment<=0) insert(yytext,'c');}
{D}*"."{D}+({E})?{FS}?  {if(nestedComment<=0) insert(yytext,'c');}
{D}+"."{D}*({E})?{FS}?  {if(nestedComment<=0) insert(yytext,'c');}

                                                                    48,149        22%
```

Figure 2

```
 /*RegEx for String Literals*/

L?\"(\\.|[^\\"])*\" {if(nestedComment<=0) insert(yytext,'s');}
L?\"(\\.|[^\\"])*   {if(nestedComment<=0) printf("ERROR: String does not end\n");}

 /*RegEx for Operators*/

"..."|">>="|"<<="|"+="|"-="|"*="|"/="       {if(nestedComment<=0) insert(yytext,'o');}

"%="|"&="|"^="|"|="|">>"|"<<"|"++"|"--"|"->"|"&&"|"||"|"<="         {if(nestedComment<=0) insert(yytext,'o');}

">="|"=="|"!="      {if(nestedComment<=0) insert(yytext,'o');}

 /*RegEx for Punctuators*/

";"         {if(nestedComment<=0) insert(yytext,'p');}
"{"         {if(nestedComment<=0) {
                insert(yytext,'p');
                bracCount++;
                }
            }
"<%"        {if(nestedComment<=0) insert(yytext,'p');}
"}"         {if(nestedComment<=0) {
                insert(yytext,'p');
                bracCount--;
                }
            }
"%>"        {if(nestedComment<=0) insert(yytext,'p');}
","         {if(nestedComment<=0) insert(yytext,'p');}
":"         {if(nestedComment<=0) insert(yytext,'p');}
"="         {if(nestedComment<=0) insert(yytext,'p');}
"("         {if(nestedComment<=0) insert(yytext,'p');}
")"         {if(nestedComment<=0) insert(yytext,'p');}
("["|"<:")      {if(nestedComment<=0) insert(yytext,'p');}
("]"|":>")      {if(nestedComment<=0) insert(yytext,'p');}
"."         {if(nestedComment<=0) insert(yytext,'p');}

 /*RegEx for Operators*/

"&"         {if(nestedComment<=0) insert(yytext,'o');}
"!"         {if(nestedComment<=0) insert(yytext,'o');}
"~"         {if(nestedComment<=0) insert(yytext,'o');}
"-"         {if(nestedComment<=0) insert(yytext,'o');}
"+"         {if(nestedComment<=0) insert(yytext,'o');}
"*"         {if(nestedComment<=0) insert(yytext,'o');}
"/"         {if(nestedComment<=0) insert(yytext,'o');}
"%"         {if(nestedComment<=0) insert(yytext,'o');}
"<"         {if(nestedComment<=0) insert(yytext,'o');}

                                                                    143,148-154   45%
```

Figure 3

Phase 1 – Lexical Analyzer

```
">"         {if(nestedComment<=0) insert(yytext,'o');}
"^"         {if(nestedComment<=0) insert(yytext,'o');}
"|"         {if(nestedComment<=0) insert(yytext,'o');}
"?"         {if(nestedComment<=0) insert(yytext,'o');}

[ \t\v\n\f]     {;}
.           { /* ignore bad characters */ }

%%

/* USER CODE */

int main()
{
 comment = (char*)malloc(100*sizeof(char));
 yyin=fopen("input.txt","r");
 yyout=fopen("out.txt","w");
 fprintf(yyout,"\n Symbol Table Format is:\n Lexeme\t\t\t\t\tToken\t\t\t\t\tAttribute Value\n");
 yylex();
 if(nestedComment!=0)
    printf("ERROR: Comment does not end\n");
 if(bracCount!=0)
    printf("ERROR: Bracket mismatch\n");
 fprintf(yyout,"\n");
 if(flag==1)
 {
   lineCount=0;
   fprintf(yyout,"\n\nComment (%d lines):\n",lineCount);
   fprintf(yyout,"ERROR: Nested Comment");
 }
 else
 {
   fprintf(yyout,"\n\nComment (%d lines):",lineCount);
   fputs(comment,yyout);
 }
 fclose(yyout);
}
int yywrap()
{
 return(1);
}

void add_to_comment(char *yytext)
{
 /* Function to display comments separately! */

 int len1,len2;
 char *temp;
                                                                    191,131       68%
```

Figure 4

```
 len1 = strlen(comment);
 len2 = strlen(yytext);
 temp = (char*)malloc((len1+1)*sizeof(char));
 strcpy(temp,comment);
 comment = (char*)malloc((len1+len2+1)*sizeof(char));
 strcat(temp,yytext);
 strcpy(comment,temp);
}

void insert(char *yytext,char type)
{
  /* Function to insert symbols to the Symbol Table */

    int len1 = strlen(yytext);
    char token[20];
    struct holder *symbol,*temp,*nextptr;
    nextptr = head;
    switch(type)
    {
        case 'c':
            strcpy(token,"Constant");
            break;
        case 'v':
            strcpy(token,"Variable");
            break;
        case 'p':
            strcpy(token,"Punctuator");
            break;
        case 'o':
            strcpy(token,"Operator");
            break;
        case 'k':
            strcpy(token,"Keyword");
            break;
        case 's':
            strcpy(token,"String Literal");
            break;
        case 'd':
            strcpy(token,"Preprocessor Statement");
            break;
    }
    if(nestedComment<=0)
    {
        for(i=0;i<var_cnt;i++,nextptr=nextptr->next)
        {
            symbol = nextptr;
            if(strcmp(symbol->name,yytext)==0)
                break;
                                                                    239,118       91%
```

Figure 5

```
209     switch(type)
210     {
211         case 'c':
212             strcpy(token,"Constant");
213             break;
214         case 'v':
215             strcpy(token,"Variable");
216             break;
217         case 'p':
218             strcpy(token,"Punctuator");
219             break;
220         case 'o':
221             strcpy(token,"Operator");
222             break;
223         case 'k':
224             strcpy(token,"Keyword");
225             break;
226         case 's':
227             strcpy(token,"String Literal");
228             break;
229         case 'd':
230             strcpy(token,"Preprocessor Statement");
231             break;
232     }
233     if(nestedComment<=0)
234     {
235         for(i=0;i<var_cnt;i++,nextptr=nextptr->next)
236         {
237             symbol = nextptr;
238             if(strcmp(symbol->name,yytext)==0)
239                 break;
240         }
241         if(i==var_cnt)
242         {
243             temp = (struct holder*)malloc(sizeof(struct holder));
244             temp->name = (char*)malloc((len1+1)*sizeof(char));
245             strcpy(temp->name,yytext);
246             temp->val = i;
247             temp->next = NULL;
248             if(var_cnt==0)
249                 head = temp;
250             else
251                 symbol->next = temp;
252             var_cnt++;
253         }
254         fprintf(yyout,"\n%s\t\t\t\t\t\t%s\t\t\t\t\t\t%d",yytext,token,i);
255     }
256 }
                                                                    256,108       Bot
```

Figure 6

Run

Explanation:

This is a shell script which automates the compiling and execution of the lex code.

Code:

5 lines (4 sloc) | 49 Bytes

```
1  #!/bin/sh
2  lex scanner.l
3  gcc lex.yy.c -ll
4  ./a.out
```

Figure 7

Test Cases

Test Case No.	Input Type	Input	Status
1	Preprocessor statement	#include<stdio.h> #define x 5	Passed Passed

2	Constants	float x = 5.0;
		
		double y = 10.0;
		
		int a = 2;
		
		char c = ‘c’;
			Passed
			Constant = 5.0
			
			Passed
			Constant = 10.0
			
			Passed
			Constant = 2.0
			
			Passed
			Constant = c

3	Keywords	int a = 2;
		
		float x = 5.0;
		
		return (0);
			Passed
			Keyword = int
			
			Passed
			Keyword = float
			
			Passed
			Keyword = return

4	Punctuators	int main()
		
		{
		
		int a,b
		
		int c;
		
		}
		
		int a = 5;
			Passed
			Punctuator = (,)
			
			Passed
			Punctuator = {
			
			Passed
			Punctuator = ,
			
			Passed
			Punctuator = ;
			
			Passed
			Punctuator = }
			
			Passed
			Punctuator = ‘=’

5	Variables	int amount; float rate; char a[50] = “Chirag”; double principle;	Passed Variable = amount Passed Variable = rate Passed Variable = a Passed Variable = principle
6	String Literal	char a[50] = “Chirag”; char z[50] = “Chirag	Passed String = Chirag Passed ERROR: String doesn’t end

7	Operators	a = b * c;
		
		c = a + b;
		
		d = a/c;
		
		a++;
		
		--a;
		
		d = a>b?1:2;
		
		c = !a;
		
		d = a||b;
		
		d = a!:b
			Passed
			Operator = *
			
			Passed
			Operator = +
			
			Passed
			Operator = /
			
			Passed
			Operator = ++
			
			Passed
			Operator = --
			
			Passed
			Operator = >, ?:
			
			Passed
			Operator = !
			
			Passed
			Operator = ||
			
			Passed
			No operator

8	Comments	//Single line comment /* a	Passed Passed

		b */ /* This is a multi /*this */ line comment*/ /* This	 Passed ERROR: Nested comment Passed ERROR: Comment does not end

9	scanf() / printf() error	scanf("",);
		
		printf("",);
			Passed
			ERROR: scanferror
			
			Passed
			ERROR: printferror

Implementation

We have implemented a Lexical Analyzer for a subset of C language using flex tool. This tool takes regular expressions as an input and gives the corresponding token as an output. This phase involves building the Lexical Analyzer which goes through the source program, identifies the tokens and inserts them into the symbol table and constant table.

Identification of tokens:

We have written regular expressions for all tokens. The flex tool performs the longest match and classifies the lexeme into the corresponding token. We have separate regular expressions for keywords, identifiers, strings, comments and so on. For each token, the action taken is also defined separately according to its requirements. For example, when it identifies a comment or a delimiter or a preprocessor directive, it simply strips it out. However, when it is not able to classify a lexeme as a token it throws an error saying “unidentified token”.

Symbol and Constant Table:

We identify keywords, constants, variables, punctuators, operators, preprocessor statements and string literals. The symbol and constant table is stored in a linked list and displayed in the *out.txt* file. The linked list contains the token name and a value. Attribute Value is a sequential identification number given to a lexeme, i.e. that node in the linked list will contain details of that lexeme. At the bottom of the symbol table we are printing the number of comment lines and the comment itself.

Error Generation:

This lexical analyzer also generates errors if found. We have initialized variables which is incremented and decremented to maintain the count of the various brackets and comment symbols. In case of any discrepancy, an appropriate error is thrown. For unidentified token, we have defined a regular

expression that matches to anything that doesn't match any of the tokens defined earlier. We also handle invalid string literals, bad comments, printf() and scanf() errors.

Result

We have made a scanner which scans for a subset of C languages. The example inputs in the next section demonstrate error generation and token identification.

Examples

Example 1

This example demonstrates bracket mismatches, nested and unterminated comments.

Input

Figure 8

Output

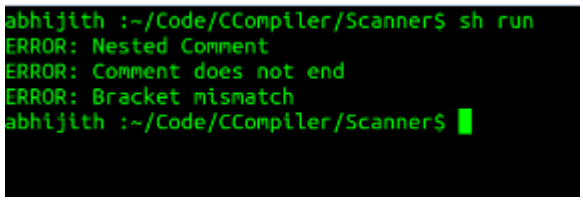


Figure 9

Example 2

This example demonstrates printf() errors, string errors and bracket mismatches.

Input

Figure 10

Output

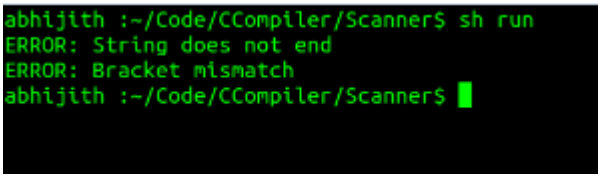


Figure 11

Example 3

This example demonstrates unidentified tokens.

Input

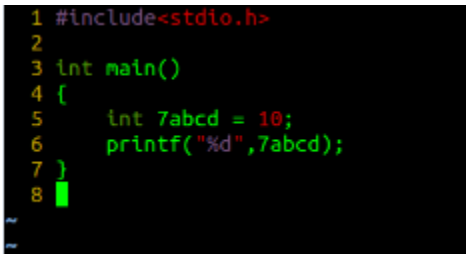


Figure 12

Output

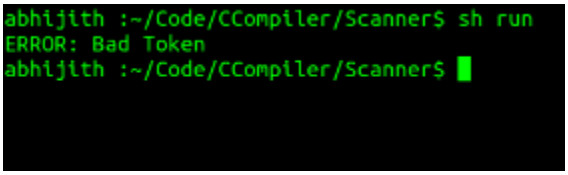


Figure 13

Example 4

This example is an extensive test case with multiple errors.

Input

```
1 #include<stdio.h>
2 /* for for for This is a multi
3 /*I am not a nested comment*/
4 line comment*/
5 int main() {
6    int amount, rate, time, si;
7    printf("\nEnter Principal Amount : ");
8    scanf("%d", &amount);
9
10   printf("\nEnter Rate of Interest : ");
11   scanf("%d", &rate);
12
13   printf("\nEnter Period of Time   : ");
14   scanf("%d", &time);
15   scanf("",);
16   si = (amount * rate * time) / 100;
17   printf("\nSimple Interest : %d", si);
18   printf("",);
19   return(0);
20 }
21 }
22 /* Heyy how are you? */
23 "I am a
~
~
~
```

Figure 14

Output

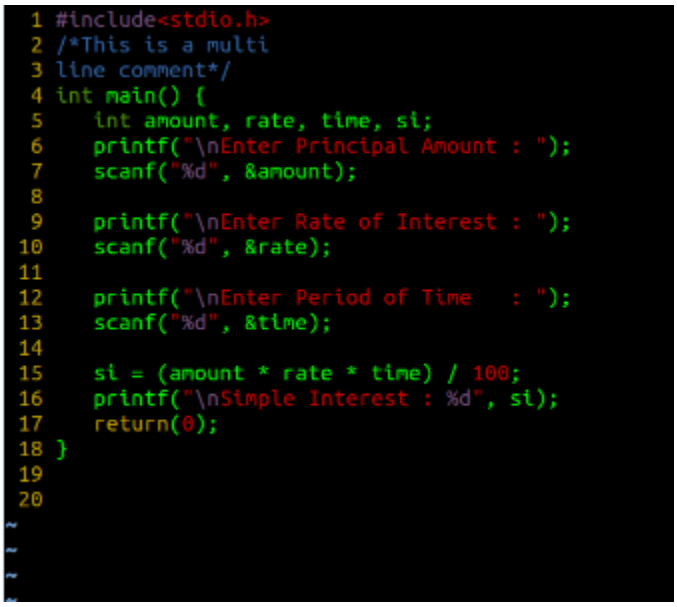
```
abhijith :~/Code/CCompiler/Scanner$ sh run
ERROR: Nested Comment
ERROR: scanferror
ERROR: printferror
ERROR: String does not end
ERROR: Bracket mismatch
abhijith :~/Code/CCompiler/Scanner$
```

Figure 15

Example 5

This example has no errors. The output shows the contents of the output file. The output file contains the symbol-constant table and information about comment lines.

Input



```
1 #include<stdio.h>
2 /*This is a multi
3 line comment*/
4 int main() {
5    int amount, rate, time, si;
6    printf("\nEnter Principal Amount : ");
7    scanf("%d", &amount);
8
9    printf("\nEnter Rate of Interest : ");
10   scanf("%d", &rate);
11
12   printf("\nEnter Period of Time   : ");
13   scanf("%d", &time);
14
15   si = (amount * rate * time) / 100;
16   printf("\nSimple Interest : %d", si);
17   return(0);
18 }
19
20
```

Figure 16

Output

```
Symbol Table Format is:
   Lexeme                          Token                  Attribute Value

  #include<stdio.h>        Preprocessor Statement               0
               int                      Keyword                 1
              main                     Variable                2
                 (                    Punctuator               3
                 )                    Punctuator               4
                 {                    Punctuator               5
               int                       Keyword                1
            amount                     Variable                6
                 ,                    Punctuator               7
              rate                     Variable                8
                 ,                    Punctuator               7
              time                     Variable                9
                 ,                    Punctuator               7
                si                     Variable               10
                 ;                    Punctuator              11
                 ;                    Punctuator              11
                 ;                    Punctuator              11
                 ;                    Punctuator              11
                 ;                    Punctuator              11
                 ;                    Punctuator              11
                 ;                    Punctuator              11
                si                     Variable               10
                 =                    Punctuator              12
                 (                    Punctuator               3
            amount                     Variable                6
                 *                     Operator               13
              rate                     Variable                8
                 *                     Operator               13
              time                     Variable                9
                 )                    Punctuator               4
                 /                     Operator               14
               100                     Constant               15
                 ;                    Punctuator              11
                 ;                    Punctuator              11
            return                       Keyword               16
                 (                    Punctuator               3
                 0                     Constant               17
                 )                    Punctuator               4
                 ;                    Punctuator              11
                 }                    Punctuator              18


Comment (2 lines):
This is a multi
line comment
```

Figure 17

Future Work

Future work involves making the other parts of the compiler - parser, semantic checker and the intermediate code generator.

References

[1] Flex, version 2.5, A fast scanner generator https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html, Edition 2.5, March 1995

[2] Flex Tutorial, http://alumni.cs.ucr.edu/~lgao/teaching/flex.html

[3] Flex: A Scanner Generator for C/C++, http://web.cse.ohio-state.edu/~gurari/course/cse756/html/cse756se8.html

[4] LEX/FLEX Scanner Generator, http://cse.iitkgp.ac.in/~goutam/compiler/lect/lect3.pdf

[5] Introduction to FLEX, http://web.eecs.utk.edu/~bvz/cs461/notes/flex/

[6] An Overview of FLEX, with examples, http://web.mit.edu/gnu/doc/html/flex_1.html