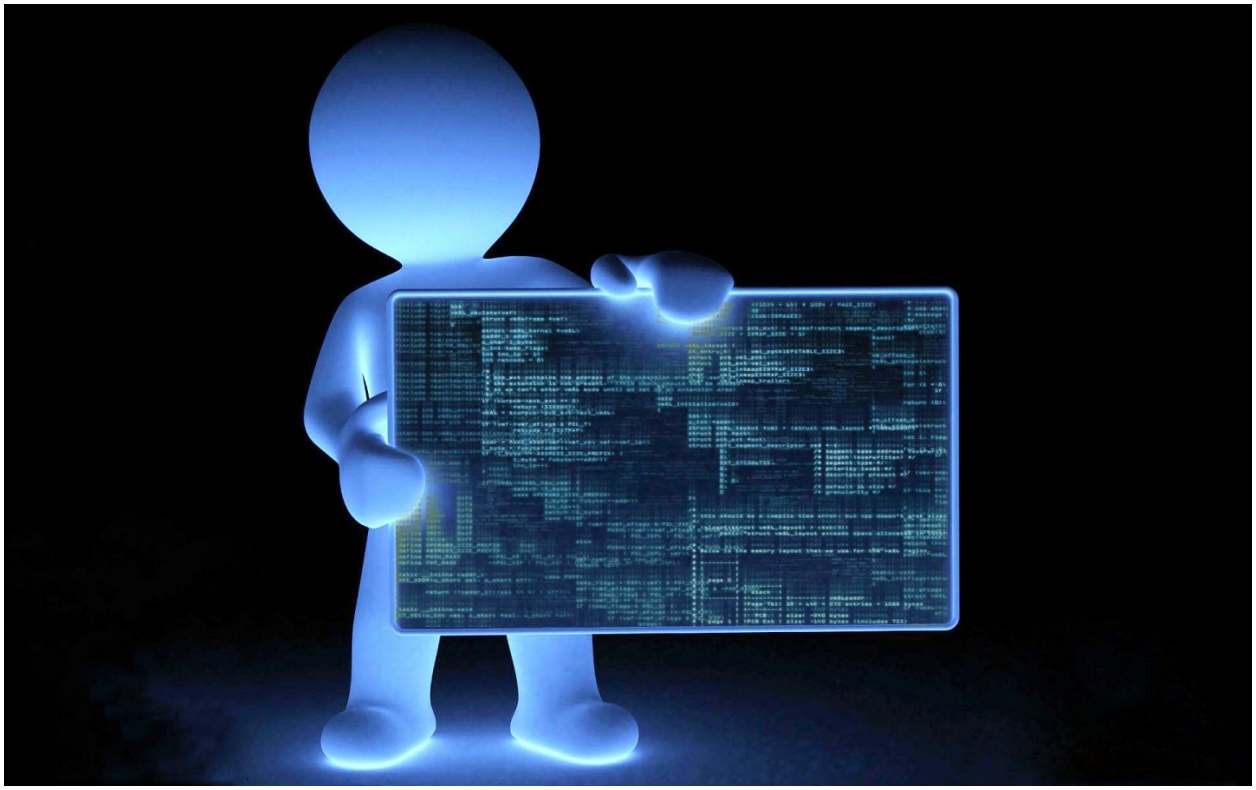


Lexical Analyzer using Flex

COMPILER DESIGN
PROJECT - 1
19.01.2018

Submitted by
Saurabh Yadav, 15CO145
Kiran S, 15CO125

Phase 1 - Lexical Analyzer

Introduction

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as C one line at a time using an *editor*. The file that is created contains what are called the *source statements*.

The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing (running), the compiler first scans all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

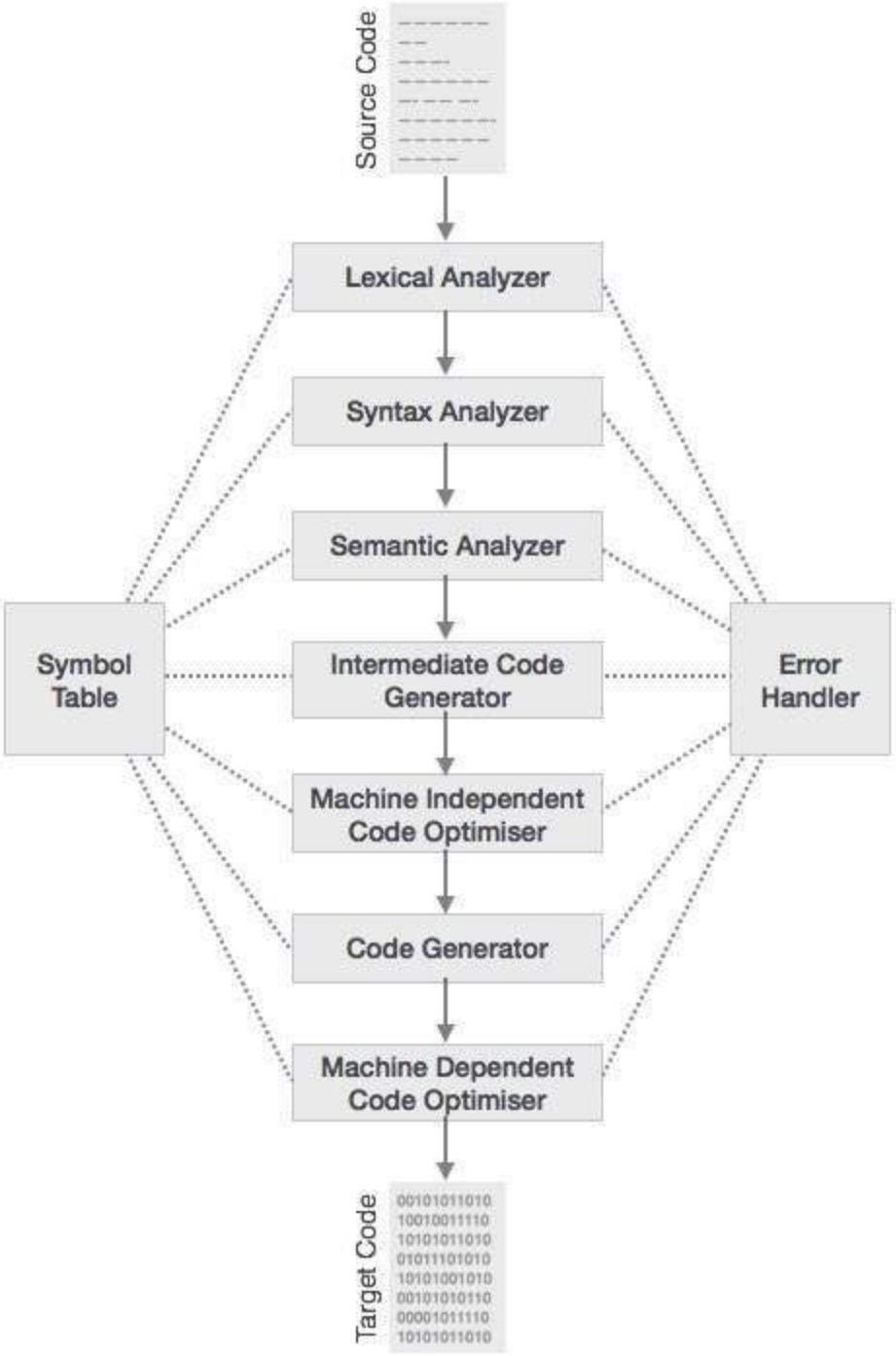
The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

Lexical analysis : This is the initial part of reading and analysing the program text. The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

Syntax analysis : This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing.

Semantic Analysis : This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that does not make sense given the type of the variable, such as trying to use a boolean value as a function pointer.

Intermediate code generation : The program is translated to a simple machine independent intermediate language.



Lexical Analyzer

Lexical analysis is the first phase of a compiler. It is the process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called lexical tokens, or just tokens, which may be handled more easily by a parser. The lexical analyzer reads the source text and, thus, it may perform certain secondary tasks:

- Eliminate comments and white spaces in the form of blanks, tab and newline characters.
- Correlate errors messages from the compiler with the source program (eg, keep track of the number of lines).

The interaction with the parser is usually done by making the lexical analyzer be a sub-routine of the parser.

Token: A token is a group of characters having collective meaning: typically a word or punctuation mark, separated by a lexical analyzer and passed to a parser.

Pattern : A rule that describes the set of strings associated to a token. Expressed as a regular expression and describing how a particular token can be formed. For example, [A-Za-z][A-Za-z_0-9] * The pattern matches each string in the set.

Lexeme : It is an actual character sequence in the source text forming a specific instance of a token that is matched by the pattern for a token.

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme. The lexical analyzer collects information about tokens into their associated attributes.

In practice, a token has usually only a single attribute - a pointer to the symbol-table entry in which the information about the token is kept such as: the lexeme, the line number on which it was first seen.

Source Code

```
%{
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    int count=0,nest_cmt=0,line_count=1,comm_line;
    struct token_list
    {
      int attr_num;
      char *name,token_type[25];
      struct token_list *next;
    }*header;
    void insert(char *yytext,char type);
%}

/*=================================================================*/

keyword
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"doubl
e"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"reg
ister"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|
"typedef"|"union"|"unsigned"|"void"|"volatile"|"while"
letter      [a-zA-Z_]
loop        "for"|"while"
id         {letter}({letter}|{digit})*
digit       [0-9]
hexa        [a-fA-F0-9]
oct         [0-7]
exp         [Ee][+-]?{digit}+
```

```
float_suff    (f|F|l|L)
int_suff    (u|U|l|L|ul|UL|ll|LL|ull|ULL)
singlecomm     (\/\/.*)
comm_beg     (\/\*)
comm_end     (\*\/)
space       [ \t]+
punctuator    ";"|"{"|"}"|","|":"|"("|")"|"."
operator
">>="|"<<="|"+="|"-="|"*="|"/="|"%="|"&="|"^="|"|="|">>"|"<<"|"++"|"-
-"|"->"|"&&"|"||"|"<="|">="|"=="|"!="|"&"|"!"|"~"|"-"|"+"|"*"|"/"|"="
|"%"|"<"|">"|"^"|"|"|"?"

%x COMM

/*=========================================================*/

%%

 /* Line Count */
<*>\n    {line_count++;}

 /* Pre-processor directives */
^#([-a-zA-Z0-9.]|<|>|{space})*     {insert(yytext,'d');}

 /* RegEx for Keywords */
{keyword}                              { insert(yytext,'k'); }

 /* RegEx for Constants Literals */
0[xX]{hexa}+{int_suff}?                {insert(yytext,'1');}
0{oct}+{int_suff}?               {insert(yytext,'2');}
```

```
{digit}+{int_suff}?                    {insert(yytext,'3');}
{digit}+{float_suff}?                  {insert(yytext,'4');}
{digit}+{exp}{float_suff}?             {insert(yytext,'4');}
{digit}*"."{digit}+({exp})?{float_suff}?    {insert(yytext,'4');}
{digit}+"."{digit}*({exp})?{float_suff}?    {insert(yytext,'4');}

 /*RegEx for Identifier */
{id}             { insert(yytext,'i'); }
{digit}+{letter}+          { printf("Line %d: error: Invalid
Identifier\n",line_count);}

 /*RegEx to identify functions and loops*/
{loop}{space}?"("    {insert(yytext,'l');}
{keyword}{space}?"("    {insert(yytext,'k');}
{id}{space}?"("       {insert(yytext,'f');}

      /*
{id}{space}?"(({keyword}{space}"*"?{id}?",")*{keyword}{space}"*"?{i
d})?")"    {insert(yytext,'f');} */
      /* {id}{space}?"(("*"?{id}?",")*"*"?{id})?")"
{insert(yytext,'f');}*/


 /*RegEx for Comments*/
{singlecomm}    {}

{comm_beg}    {
            BEGIN(COMM);
            nest_cmt++;
            comm_line=line_count;
```

```
          }

{comm_end}    { printf("Line %d: error: Invalid Comment
Terminator\n",line_count);}

<COMM>{comm_beg}    {
                nest_cmt++;
                if(nest_cmt>1)    printf("Line %d: error: Nested
Comment\n",line_count);
                     }

<COMM>{comm_end}     {
                    if(nest_cmt>0)    nest_cmt--;
                    if(nest_cmt==0)    BEGIN(INITIAL);
                     }

<COMM>. {}

 /* RegEx for String Literals */
\"(\\.|[^\\"])*\"    {insert(yytext,'s');}
\"(\\.|[^\\"])*       {printf("Line %d: error: Incomplete String
Literal\n",line_count);}

 /*RegEx for Operators*/
{operator}    {insert(yytext,'o');}

 /*RegEx for Punctuators*/
{punctuator}    {}

 /* RegEx to ignore unnecessary spaces */
```

```
[ \t\v\n\f]    {}
"["|"]"|"\'\\0\'" {}

.    {printf("Line %d: error: Bad Character\n",line_count);}

%%

/*=========================================================*/

void insert(char *yytext,char type)
{
    int len1 = strlen(yytext);
    char token_type[25];
    struct token_list *lexeme,*temp,*ptr;
    ptr = header;
    switch(type)
    {
      case 'k':
           strcpy(token_type,"Keyword");
           if(yytext[strlen(yytext)-1]=='('
yytext[strlen(yytext)-1]='\0';
           break;
      case 'l':
           strcpy(token_type,"Keyword (Loop)");
           yytext[strlen(yytext)-1]='\0';
           break;
      case 'i':
           strcpy(token_type,"Identifier (Variable)");
           break;
      case 'f':
```

```
            strcpy(token_type,"Identifier (Function)");
            yytext[strlen(yytext)-1]='\0';
            break;
      case 's':
            strcpy(token_type,"String Literal");
            break;
      case 'o':
            strcpy(token_type,"Operator");
            break;
      case 'd':
            strcpy(token_type,"Preprocessor Directive");
            break;
      case '1':
            strcpy(token_type,"Constant (Hexadecimal)");
            break;
      case '2':
            strcpy(token_type,"Constant (Octal)");
            break;
      case '3':
            strcpy(token_type,"Constant (Integer)");
            break;
      case '4':
            strcpy(token_type,"Constant (Float)");
            break;
   }
   if(nest_cmt==0)
   {
      int i;
      for(i=0;i<count;i++,ptr=ptr->next)
```

```
      {
            lexeme = ptr;
            if(strcmp(lexeme->name,yytext)==0) break;
      }
      if(i==count)
      {
            temp = (struct token_list*)malloc(sizeof(struct
token_list));
            temp->attr_num=line_count;
            temp->name = (char*)malloc((len1+1)*sizeof(char));
            strcpy(temp->name,yytext);
            strcpy(temp->token_type,token_type);
            temp->next = NULL;
            if(count==0) header=temp;
            else lexeme->next = temp;
            count++;
            fprintf(yyout,"\n%35s %30s
%25d",temp->name,temp->token_type,temp->attr_num);
      }
    }
}


/* main() function */

int main()
{
    yyin=fopen("testcase.txt","r");
    yyout=fopen("output.txt","w");
```

```
fprintf(yyout,"\t\t\t\t\t-------------------------------------------
\n");
    fprintf(yyout,"\t\t\t\t\t\t\tSYMBOL TABLE\n");

fprintf(yyout,"\t\t\t\t\t-------------------------------------------
\n");
    fprintf(yyout,"\t\t\tLexeme\t\t\t\tToken\t\t\tAttribute
Value\n");
    yylex();
    if(nest_cmt!=0)    printf("Line %d: error: Unterminated
Comment\n",comm_line);
    fprintf(yyout,"\n");
    fclose(yyout);
}

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

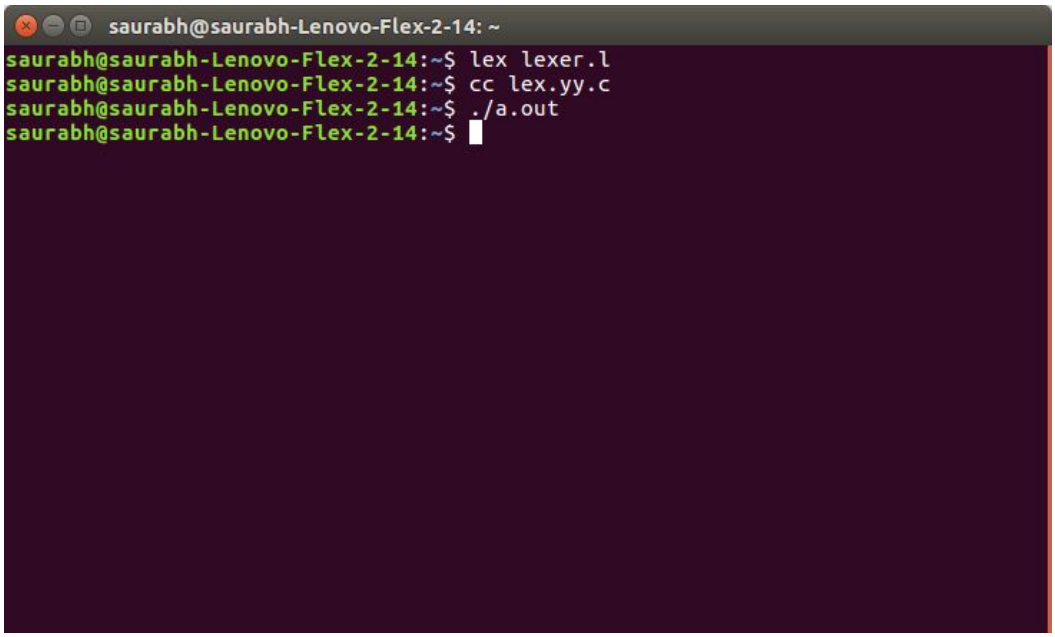
Test Cases

Test Case (1)

test1.c

```c
#include <stdio.h>

int main()
{
    int year;

    printf("Enter a year: ");
    scanf("%d",&year);

    if(year%4 == 0)
    {
        if( year%100 == 0)
        {
            // year is divisible by 400, hence the year is a leap year
            if ( year%400 == 0)
                printf("%d is a leap year.", year);
            else
                printf("%d is not a leap year.", year);
        }
        else
            printf("%d is a leap year.", year );
    }
    else
        printf("%d is not a leap year.", year);
    return 0;
}

```

Output



Test Case (2)

test2.c

```c
#include<stdio.h>
#define X 5

main()
{
    int a;
    /* A multi-line comment
        which is not closed
            sad

    scanf("%d",&a);
    printf("%d",a*X);
}

```

Output

```
saurabh@saurabh-Lenovo-Flex-2-14: ~
saurabh@saurabh-Lenovo-Flex-2-14:~$ lex lexer.l
saurabh@saurabh-Lenovo-Flex-2-14:~$ cc lex.yy.c
saurabh@saurabh-Lenovo-Flex-2-14:~$ ./a.out
Line 7: error: Unterminated Comment
saurabh@saurabh-Lenovo-Flex-2-14:~$
```

Test Case (3)

test3.c

```c
#include <stdio.h>
int main()
{
    int n, 5flag = 0;       // 5flag is an invalid identifier

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    int var;
    for(var=2; var<=n/2; ++var)
    {
        // condition for nonprime number
        if(n @ var==0)  // @ is a bad character
        {
            5flag=1;
            break;
        }
    }

    if (5flag==0)
        printf("%d is a prime number." ,n);
    else
        printf("%d is not a prime number.,n);

    return 0;
}

```

Output

```
saurabh@saurabh-Lenovo-Flex-2-14: ~
saurabh@saurabh-Lenovo-Flex-2-14:~$ lex lexer.l
saurabh@saurabh-Lenovo-Flex-2-14:~$ cc lex.yy.c
saurabh@saurabh-Lenovo-Flex-2-14:~$ ./a.out
Line 4: error: Invalid Identifier
Line 13: error: Bad Character
Line 15: error: Invalid Identifier
Line 20: error: Invalid Identifier
Line 23: error: Incomplete String Literal
saurabh@saurabh-Lenovo-Flex-2-14:~$
```

Test Case (4)

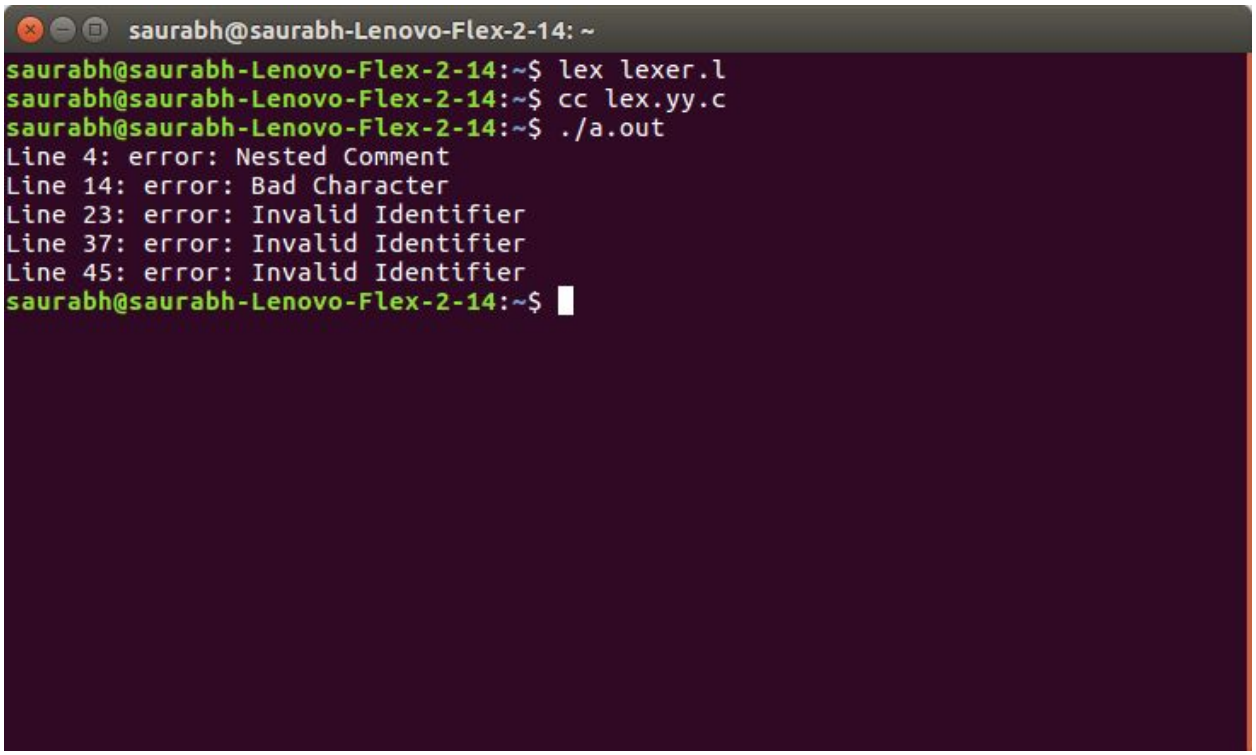
test4.c

```c
#include<stdio.h>

/* This is a
   /* Nested
      multi
      line
   */
   Comment
*/

int string_length(char *pointer)
{
   int c = 0;
   while( *(pointer $ c) != '\0' )
      c++;

   return c;
}

void reverse(char *string)
{
   int length, c;
   char 7xyz=0, *begin, *end, temp;

   length = string_length(string);
   begin  = string;
   end    = string;

```

test4.c

```c

   for(c = 0; c < length - 1; c++)
      end++;

   for(c = 0; c < length/2; c++)
   {
      temp   = *end;
      *end   = *begin;
      *begin = temp;
      7xyz++;
      begin++;
      end--;
   }
}
main()
{
   char str[100];
   int 54ull, 54f, 54s;       // 54s is not a valid integer constant
   printf("Enter a string\n");
   scanf("%d",str);
   reverse(str);
   printf("Reverse of entered string is %s", str);

   return 0;
}

```

Output


saurabh@saurabh-Lenovo-Flex-2-14: ~
saurabh@saurabh-Lenovo-Flex-2-14:~$ lex lexer.l
saurabh@saurabh-Lenovo-Flex-2-14:~$ cc lex.yy.c
saurabh@saurabh-Lenovo-Flex-2-14:~$ ./a.out
Line 4: error: Nested Comment
Line 14: error: Bad Character
Line 23: error: Invalid Identifier
Line 37: error: Invalid Identifier
Line 45: error: Invalid Identifier
saurabh@saurabh-Lenovo-Flex-2-14:~$

Symbol Table

output.txt

SYMBOL TABLE

Lexeme	Token	Attribute Value
#include<stdio.h>	Preprocessor Directive	1
int	Keyword	11
string_length	Identifier (Function)	11
char	Keyword	11
*	Operator	11
pointer	Identifier (Variable)	11
c	Identifier (Variable)	13
=	Operator	13
0	Constant (Integer)	13
while	Keyword (Loop)	14
!=	Operator	14
++	Operator	15
return	Keyword	17
void	Keyword	20
reverse	Identifier (Function)	20
string	Identifier (Variable)	20
length	Identifier (Variable)	22
begin	Identifier (Variable)	23
end	Identifier (Variable)	23
temp	Identifier (Variable)	23
for	Keyword (Loop)	29
<	Operator	29
-	Operator	29
1	Constant (Integer)	29
/	Operator	32
2	Constant (Integer)	32
--	Operator	39
main	Identifier (Function)	42
str	Identifier (Variable)	44
100	Constant (Integer)	44
54ull	Constant (Integer)	45
54f	Constant (Float)	45
printf	Identifier (Function)	46
"Enter a string\n"	String Literal	46
scanf	Identifier (Function)	47
"%d"	String Literal	47
"Reverse of entered string is %s"	String Literal	49

**********THE END**********