Lexical Analyzer for the C Language

National Institute of Technology Karnataka Surathkal

Date: 17 January 2018

Submitted To:
Prof. P. Santhi Thilagam
CSE Dept, NITK

Group Members:
Namrata Ladda, 16CO121
Mehnaz Yunus, 16CO124
Sharanya Kamath, 16CO140

Abstract

FEATURES

The project objective is to construct a compiler that studies the C programming language. It will have the following features:

- The compiler is going to support the following cases:
  - Keywords : eg: int, char, float
  - Identifiers : eg: maximum, avg
  - Constants : eg. 1, 2, 20
  - Operators: eg: +, -, *
  - Strings: eg: “nitk”, “mehnaz”, “red”
  - Special symbols: eg: [],*, ()

- Support int and char data types and also short, long, signed, unsigned subtypes.
- Detection of arrays with specified datatype (eg: int arr[10])
- Detection of looping constructs such as while, nested while.
- Detection conditional statements such as if-else and nested if-else.
- Identification of user-defined functions with one argument with return types int, char, void.
- Hashing techniques used to maintain symbol and constant tables.
- Support for single-line as well as multiline comments and return appropriate error messages.
- Appropriate error messages for comments and strings that don't end until the end of the file.

RESULTS

- Details of the identified tokens for the source program taken as input.
- Errors in the source program along with appropriate error messages
- Symbol table will be designed using hashing organization techniques.

TOOLS USED

- Flex

Contents

Introduction

Lexical Analysis

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. This phase scans the source code as a stream of characters and converts it into meaningful lexemes, 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. Lexical analyzer represents these lexemes in the form of tokens as below.

```
<token-name, attribute-value>
```

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Datatype token (id, number, real, . . .)
- Punctuation tokens (if, void, return, . . .)
- Alphabetic tokens (keywords)

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

FLEX (fast lexical analyzer generator) is a tool for generating lexical analyzers (scanners or lexers). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. The structure of our flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%%

Rules section

%%

C code section

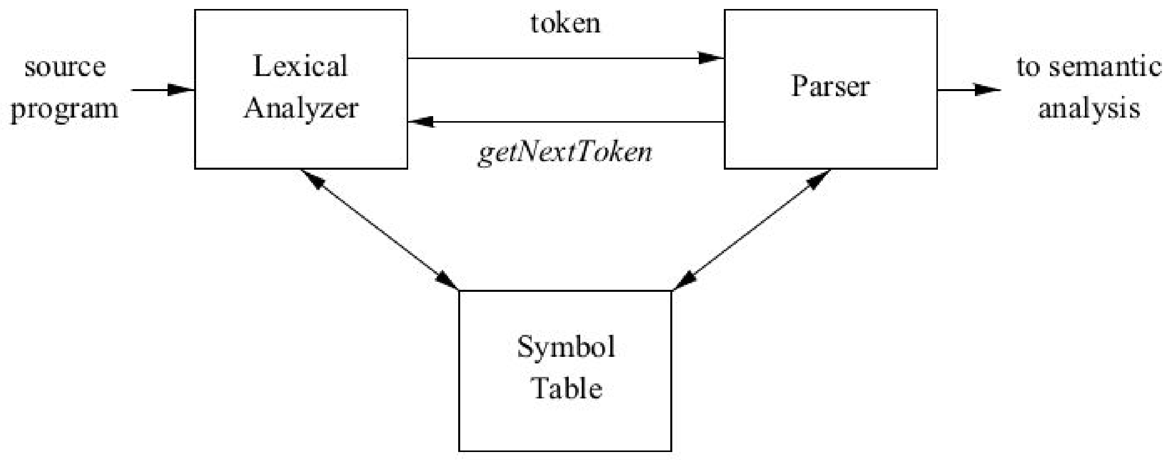
The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code. The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

C Program

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned into 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.
Lexical analysis only takes care of parsing the tokens and identifying their type. For this reason, we have assumed the C program to be syntactically correct and we generate the stream of tokens as well as the symbol table from it.

Design

Flow



Code

```
/*
Namrata Ladda, 16CO121
Mehnaz Yunus, 16CO124
Sharanya Kamath, 16CO140
```

```
Lexical Analyser
*/
%x comment
%{

    #include<stdio.h>
    #include<string.h>
    char bstack[100];
    int btop = -1;
    int nested_comment_stack = 0;
    int line = 0;
    struct hashtable{
        char name[100];
        char type[100];
        int len;
    }table[1000];

    int Hash(char *s){
        int mod = 1001;
        int l = strlen(s), val = 0, i;
        for(i = 0; i < l; i++){
            val = val * 10 + (s[i]-'A');
            val = val % mod;
            while(val < 0){
                val += mod;
            }
        }
        return val;
    }

    void insert_symbol(char *lexeme, char *token_name){

        int l1 = strlen(lexeme);
        int l2 = strlen(token_name);
        int v = Hash(lexeme);
        if(table[v].len == 0){
            strcpy(table[v].name, lexeme);
            strcpy(table[v].type, token_name);

            table[v].len = strlen(lexeme);
            return;
        }

        if(strcmp(table[v].name,lexeme) == 0)
        return;

        int i, pos = 0;

        for (i = 0; i < 1001; i++){
            if(table[i].len == 0){
                pos = i;
                break;
            }
        }

        strcpy(table[pos].name, lexeme);
        strcpy(table[pos].type, token_name);
        table[pos].len = strlen(lexeme);

    }

    void print(){
        int i;
        for(i = 0;i < 1001; i++){
            if(table[i].len == 0){
                continue;
            }
            printf("%s \t %s\n",table[i].name,table[i].type);
        }
    }

%}

LEQ <=
GEQ >=
EQ =
LES <
GRE >
PLUS \+
INCREMENT \+\+
DECREMENT \-\-
MINUS \-
MULT \*
DIV \/
REM %
AND &
OR \|
XOR \^
NOT \~
```

```
PREPROCESSOR
#(include<.*>|define.*|ifdef|endif|if|else|ifndef|undef|pragma)
STRING \".*\"|\'.*\'
WRONG_STRING \"[^"\n]*|\'[^'\n]*
SINGLELINE \/\/.*
MULTILINE "/*"([^*]|\*+[^*/])*\*+"/"
KEYWORD
auto|const|default|enum|extern|register|return|sizeof|static|struct|type
def|union|volatile|break|continue|goto|else|switch|if|case|for|do|while|
char|double|float|int|long|short|signed|unsigned|void
IDENTIFIER [a-z|A-Z]([a-z|A-Z]|[0-9])*
NUMBER_CONSTANT [1-9][0-9]*(\.[0-9]+)?|0(\.[0-9]+)?
OPERATOR {INCREMENT}|{DECREMENT}|{PLUS}|{MINUS}|{MULT}|{DIV}|{EQ}
COMPARISON {LEQ}|{GEQ}|{LES}|{GRE}
BITWISE {XOR}|{REM}|{AND}|{OR}|{NOT}
INVALID [^\n\t ]
WRONG_ID ([0-9]+[a-zA-Z][a-zA-Z0-9]*)


%%
\n line++;
[\t ] ;
; {printf("%s \t---- SEMICOLON DELIMITER\n", yytext);}
, {printf("%s \t---- COMMA DELIMITER\n", yytext);}
\{ {printf("%s \t---- PARENTHESIS\n", yytext);
      if(btop==-1){
            bstack[0]='{'; btop=1;}
      else {bstack[btop]='{';
      btop++;
    }
      }
\} {printf("%s \t---- PARENTHESIS\n", yytext);
      if(bstack[btop-1]!='{')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      btop--;
      }
\( {printf("%s \t---- PARENTHESIS\n", yytext);
      if(btop==-1){
            bstack[0]='('; btop=1;}
      else {
        bstack[btop]='(';
      btop++;
    }
```

```
      }
\) {printf("%s \t---- PARENTHESIS\n", yytext);
      if(bstack[btop-1]!='(')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      btop--;
      }
\[ {printf("%s \t---- PARENTHESIS\n", yytext);
      if(btop==-1){
            bstack[0]='['; btop=1;}
      else {
        bstack[btop]='[';
      btop++;
    }
      }
\] {printf("%s \t---- PARENTHESIS\n", yytext);
      if(bstack[btop-1]!='[')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      btop--;
      }
\\ {printf("%s \t- FSLASH\n", yytext);}
\. {printf("%s \t- DOT DELIMITER\n", yytext);}

"/*"                    {BEGIN(comment); nested_comment_stack=1;
yymore();}
<comment><<EOF>>        {printf("\nERROR: MULTILINE COMMENT: \"");
yyless(yyleng-2); ECHO; printf("\", NOT TERMINATED AT LINE NUMBER:
%d",line); yyterminate();}
<comment>"/*"           {nested_comment_stack++; yymore();}
<comment>.              {yymore();}
<comment>\n             {yymore();line++;}
<comment>"*/"           {nested_comment_stack--;
                        if(nested_comment_stack<0)
                        {
                          printf("\n \"%s\"\t---- ERROR: UNBALANCED
COMMENT AT LINE NUMBER: %d.", yytext, line);
                          yyterminate();
                        }
                        else if(nested_comment_stack==0)
                        {
                          BEGIN(INITIAL);
                        }
                        else
                          yymore();
                        }

"*/"                    {printf("%s \t---- ERROR: UNINITIALISED COMMENT
AT LINE NUMBER: %d\n", yytext,line); yyterminate();}

"//".*                  {printf("%s \t---- SINGLE LINE COMMENT\n",
yytext);}

{PREPROCESSOR} printf("%s \t---- PREPROCESSOR\n", yytext);
{STRING} {printf("%s \t---- STRING \n", yytext);
insert_symbol(yytext,"STRING CONSTANT");}
{MULTILINE} {printf("%s \t---- MULTI LINE COMMENT\n", yytext);}
{KEYWORD} {printf("%s \t---- KEYWORD\n", yytext); insert_symbol(yytext,
"KEYWORD");}
{IDENTIFIER} {printf("%s \t---- IDENTIFIER\n", yytext);
insert_symbol(yytext, "IDENTIFIER");}
{WRONG_ID} {printf("%s \t---- ERROR: ILL-FORMED IDENTIFIER\n", yytext);}
{NUMBER_CONSTANT} {printf("%s \t---- NUMBER CONSTANT\n", yytext);
insert_symbol(yytext, "NUMBER CONSTANT");}
{OPERATOR} {printf("%s \t---- ARITHMETIC OPERATOR\n", yytext);}
{BITWISE} {printf("%s \t---- BITWISE OPERATOR\n", yytext);}
{COMPARISON} {printf("%s \t---- COMPARISON OPERATOR\n", yytext);}
{WRONG_STRING} {printf("%s \t---- ERROR: UNTERMINATED STRING AT LINE
NUMBER: %d\n", yytext,line);}
{INVALID} {printf("%s \t---- ERROR: ILL-FORMED IDENTIFIER AT LINE
NUMBER: %d\n", yytext,line); }

%%

int yywrap(){
    return 1;
}

int main(){

    int i;
    for (i = 0; i < 1001; i++){
        table[i].len=0;
    }
    yyin = fopen("test-1.c","r");
    yylex();
    printf("\n\n---------------------\nSYMBOL
TABLE\n---------------------\n\n");
    print();
}
```

Explanation

Symbol Table:
It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Files :
1. lexer.l : Lex file which generates the stream of tokens and symbol table.
2. tester.c: The input C program

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

- *Preprocessor directives*
  Statements processed : `#include<stdio.h>, #define var1 var2, #ifdef var1`
  Token generated : Preprocessor Directive

- *Single-line comments*
  Statements processed : `//...........`

- *Multi-line comments*
  Statements processed : `/*...........*/, /*.../*...*/ …. */`

- *Parentheses (all types)*
  Statements processed : `(..), {..}, [..]` (without errors)
  `(..)..), {..}..}, [..]..], (..., [...` (with errors)
  Tokens generated : Parenthesis (without error) / Error with line number (with error)

- *Operators*
  Statements processed : `+, -, *, /, %`
  Tokens generated : Operators

- *Keywords*
  Statements processed : `auto, const, default, enum, extern , register, return, sizeof, static, struct, typedef, union, volatile, break, continue, goto, else, switch, if, case, for, do, while, char, double, float, int, long, short, signed, unsigned, void` and so on.
  Tokens generated: Keyword

- *Identifiers*
  Statements processed : `a, abc, a_b, a12b4`
  Tokens generated : Identifier

- *Errors for incomplete strings*
  Statements processed : `char a[]= "abcd`
  Error generated: Error Incomplete string and line number

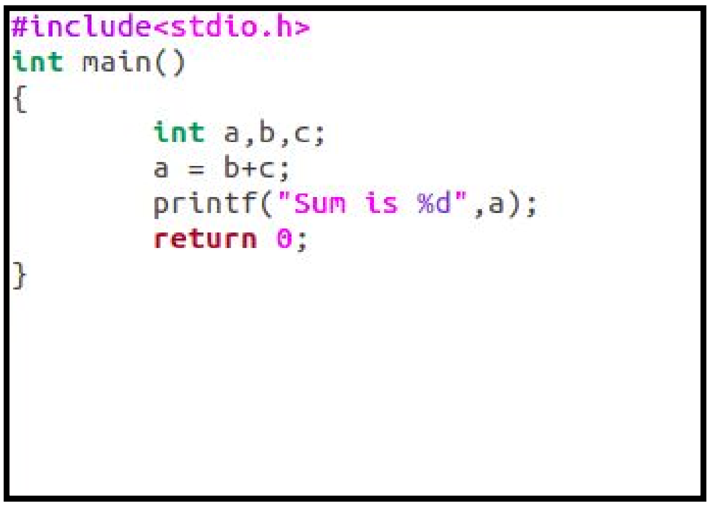
- *Errors for nested comments*
  Statements processed : `/*....../*....*/....`
  Error generated: Error with line number

- *Errors for unmatched comments*
  Statements processed : `/*..........`
  Error generated: Error with line number

Test Cases and Screenshots

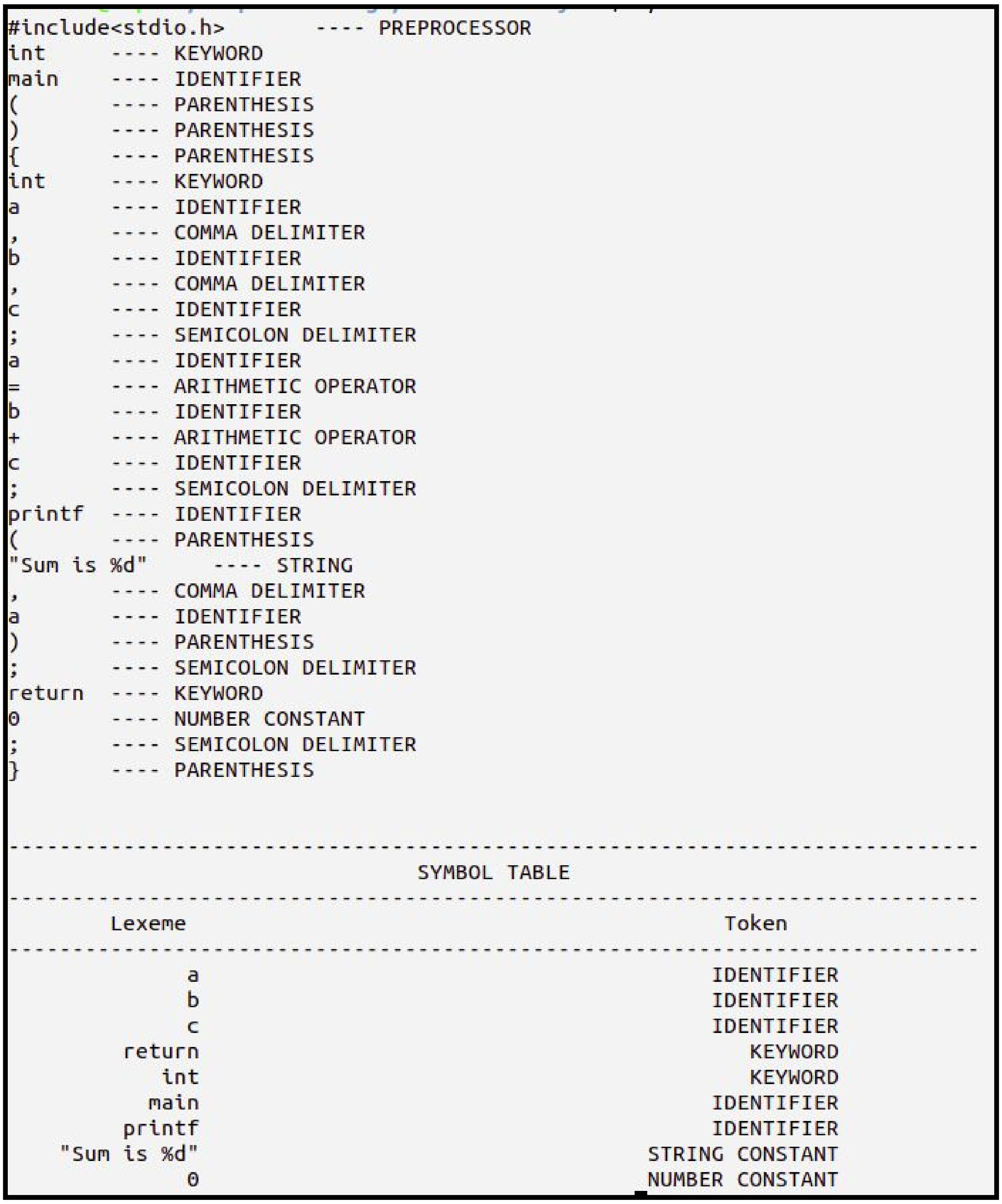
Test Case 1

Without errors



```c
#include<stdio.h>
int main()
{
        int a,b,c;
        a = b+c;
        printf("Sum is %d",a);
        return 0;
}
```

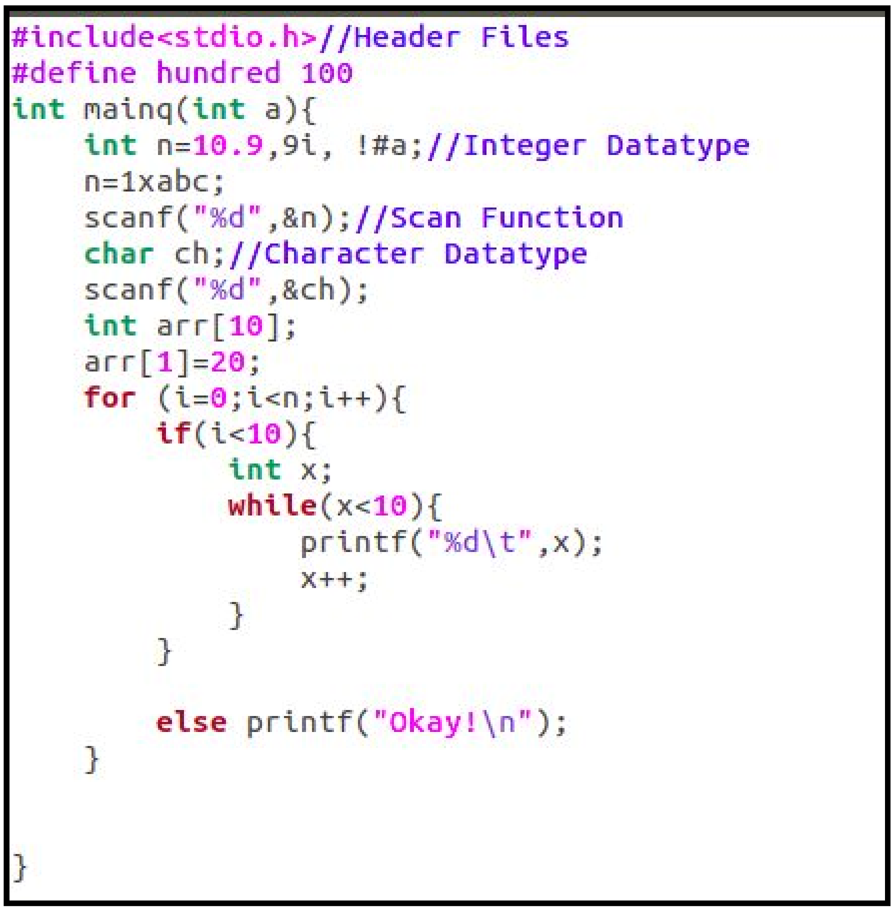
Output for Test Case 1



```
#include<stdio.h>       ---- PREPROCESSOR
int      ---- KEYWORD
main     ---- IDENTIFIER
(        ---- PARENTHESIS
)        ---- PARENTHESIS
{        ---- PARENTHESIS
int      ---- KEYWORD
a        ---- IDENTIFIER
,        ---- COMMA DELIMITER
b        ---- IDENTIFIER
,        ---- COMMA DELIMITER
c        ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
a        ---- IDENTIFIER
=        ---- ARITHMETIC OPERATOR
b        ---- IDENTIFIER
+        ---- ARITHMETIC OPERATOR
c        ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
printf   ---- IDENTIFIER
(        ---- PARENTHESIS
"Sum is %d"      ---- STRING
,        ---- COMMA DELIMITER
a        ---- IDENTIFIER
)        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
return   ---- KEYWORD
0        ---- NUMBER CONSTANT
;        ---- SEMICOLON DELIMITER
}        ---- PARENTHESIS


----------------------------------------------------------------------------
                                SYMBOL TABLE
----------------------------------------------------------------------------
        Lexeme                                          Token
----------------------------------------------------------------------------
              a                                    IDENTIFIER
              b                                    IDENTIFIER
              c                                    IDENTIFIER
         return                                       KEYWORD
            int                                       KEYWORD
           main                                    IDENTIFIER
         printf                                    IDENTIFIER
    "Sum is %d"                               STRING CONSTANT
              0                               NUMBER CONSTANT
```

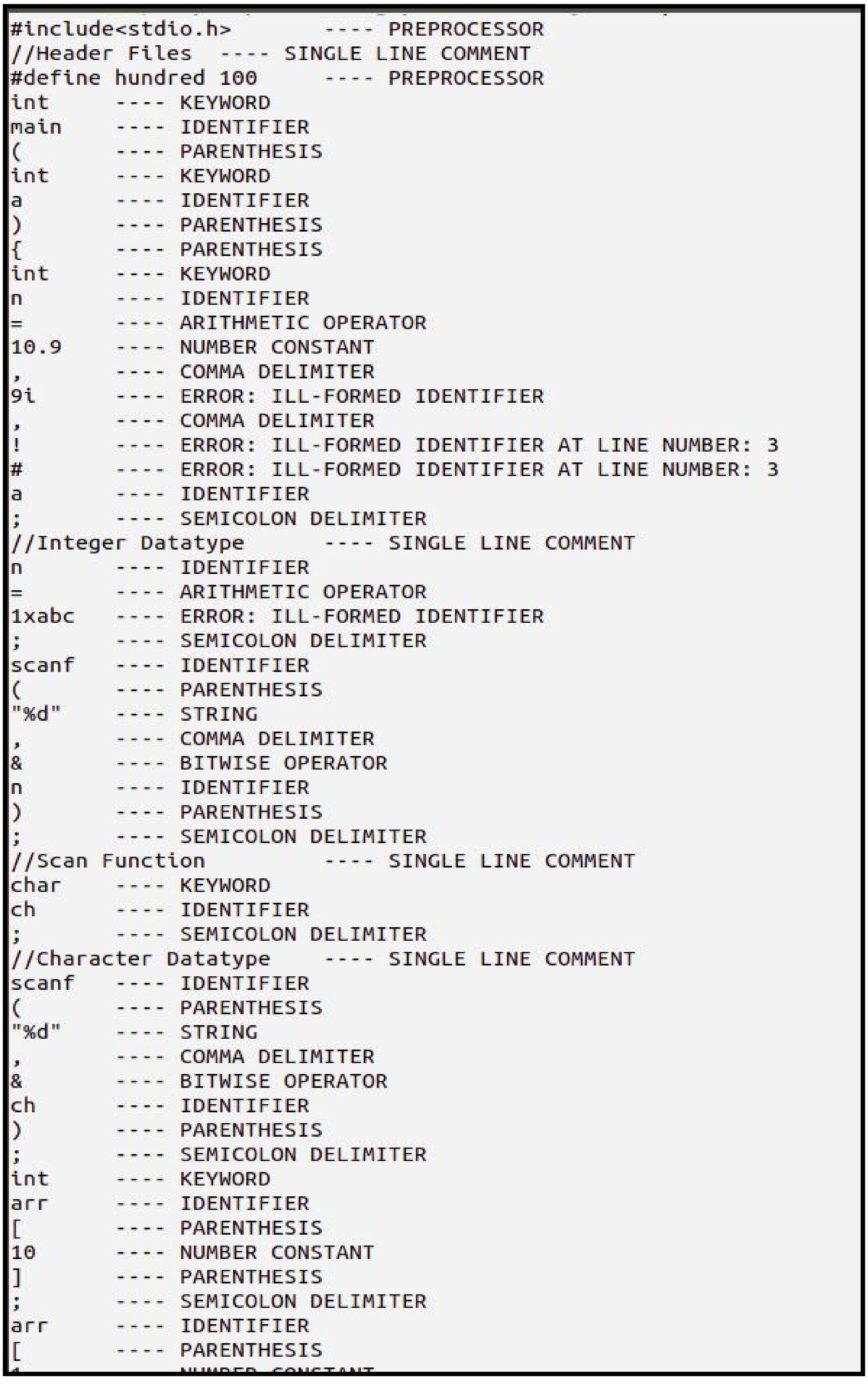
Test Case 2

Error of ill-formed identifier.

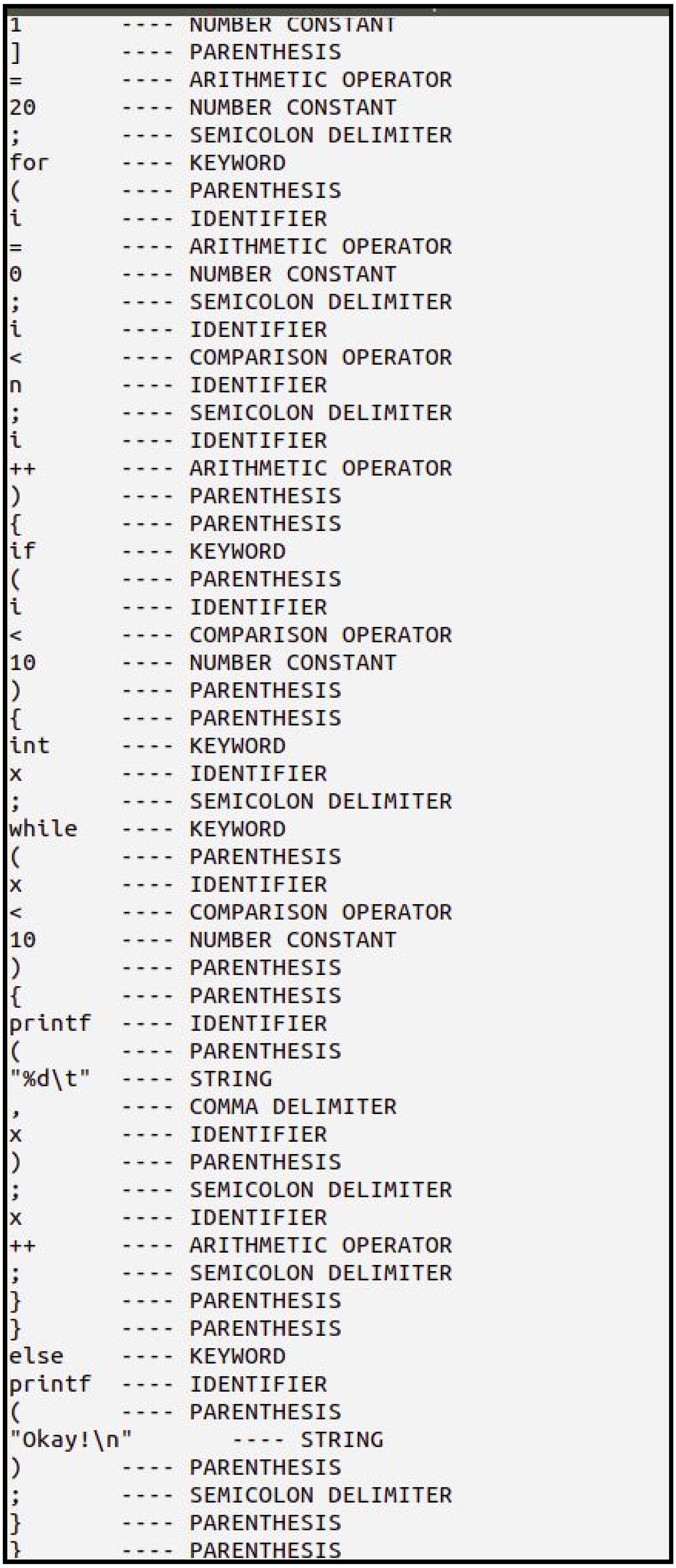


```
#include<stdio.h>//Header Files
#define hundred 100
int mainq(int a){
    int n=10.9,9i, !#a;//Integer Datatype
    n=1xabc;
    scanf("%d",&n);//Scan Function
    char ch;//Character Datatype
    scanf("%d",&ch);
    int arr[10];
    arr[1]=20;
    for (i=0;i<n;i++){
        if(i<10){
            int x;
            while(x<10){
                printf("%d\t",x);
                x++;
            }
        }

        else printf("Okay!\n");
    }


}
```

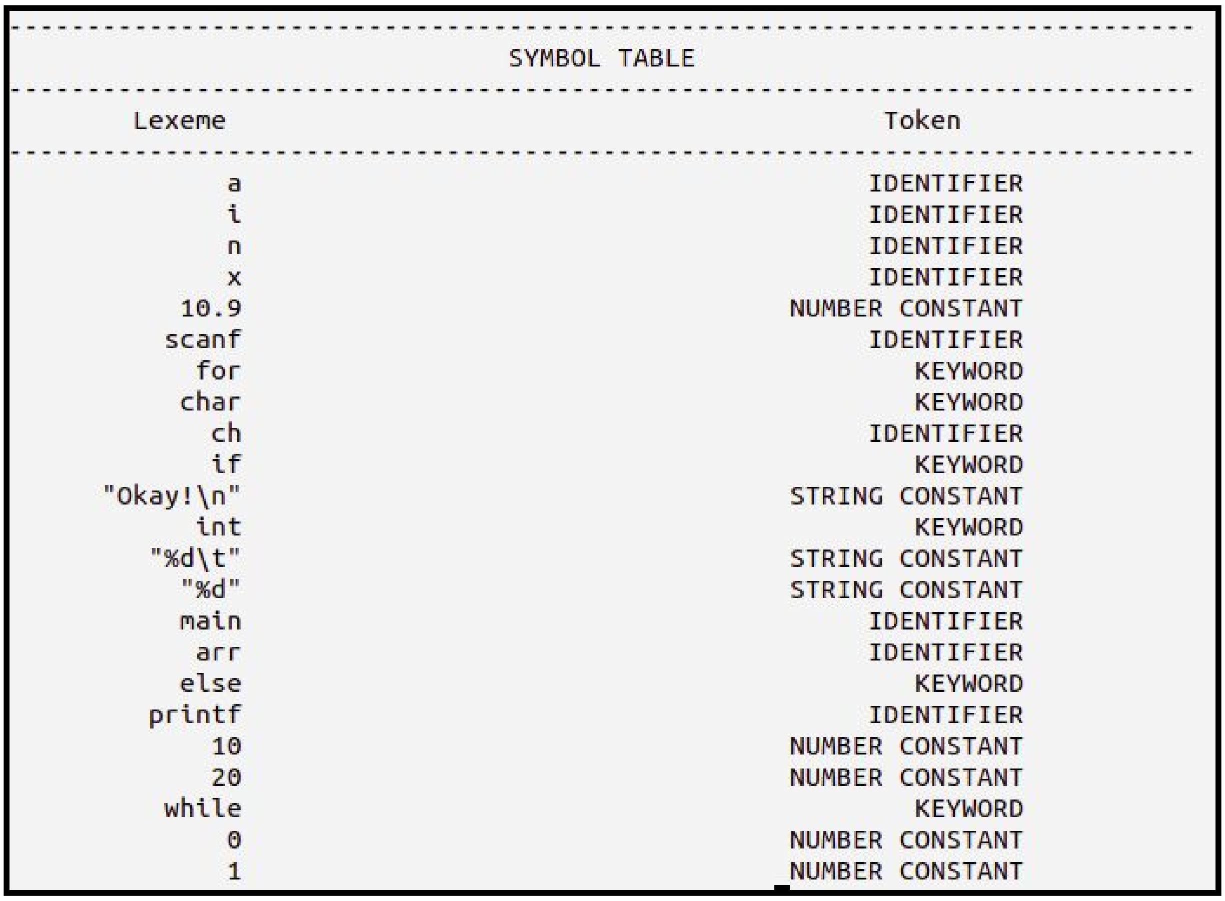
Output for Test Case 2



```
#include<stdio.h>       ---- PREPROCESSOR
//Header Files  ---- SINGLE LINE COMMENT
#define hundred 100     ---- PREPROCESSOR
int      ---- KEYWORD
main     ---- IDENTIFIER
(        ---- PARENTHESIS
int      ---- KEYWORD
a        ---- IDENTIFIER
)        ---- PARENTHESIS
{        ---- PARENTHESIS
int      ---- KEYWORD
n        ---- IDENTIFIER
=        ---- ARITHMETIC OPERATOR
10.9     ---- NUMBER CONSTANT
,        ---- COMMA DELIMITER
9i       ---- ERROR: ILL-FORMED IDENTIFIER
,        ---- COMMA DELIMITER
!        ---- ERROR: ILL-FORMED IDENTIFIER AT LINE NUMBER: 3
#        ---- ERROR: ILL-FORMED IDENTIFIER AT LINE NUMBER: 3
a        ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
//Integer Datatype      ---- SINGLE LINE COMMENT
n        ---- IDENTIFIER
=        ---- ARITHMETIC OPERATOR
1xabc    ---- ERROR: ILL-FORMED IDENTIFIER
;        ---- SEMICOLON DELIMITER
scanf    ---- IDENTIFIER
(        ---- PARENTHESIS
"%d"     ---- STRING
,        ---- COMMA DELIMITER
&        ---- BITWISE OPERATOR
n        ---- IDENTIFIER
)        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
//Scan Function         ---- SINGLE LINE COMMENT
char     ---- KEYWORD
ch       ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
//Character Datatype    ---- SINGLE LINE COMMENT
scanf    ---- IDENTIFIER
(        ---- PARENTHESIS
"%d"     ---- STRING
,        ---- COMMA DELIMITER
&        ---- BITWISE OPERATOR
ch       ---- IDENTIFIER
)        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
int      ---- KEYWORD
arr      ---- IDENTIFIER
[        ---- PARENTHESIS
10       ---- NUMBER CONSTANT
]        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
arr      ---- IDENTIFIER
[        ---- PARENTHESIS
```



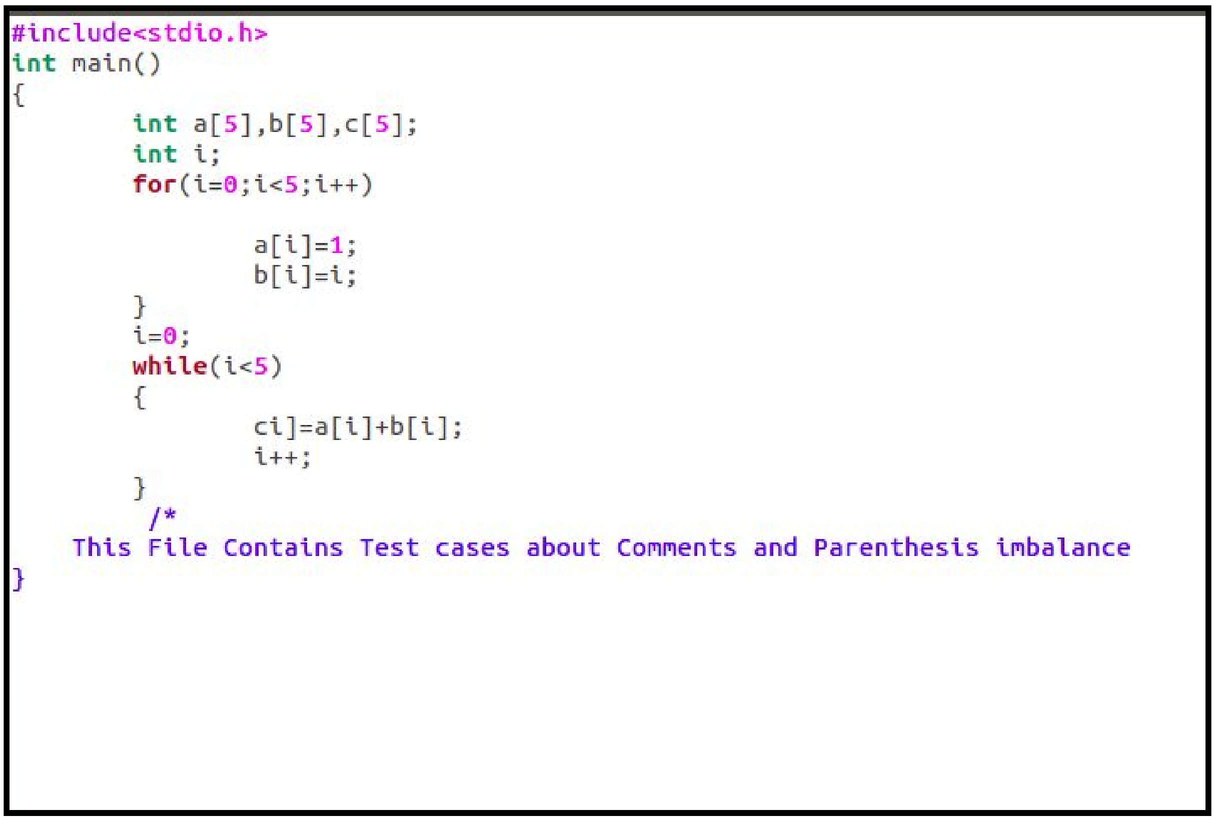
```
1        ---- NUMBER CONSTANT
]        ---- PARENTHESIS
=        ---- ARITHMETIC OPERATOR
20       ---- NUMBER CONSTANT
;        ---- SEMICOLON DELIMITER
for      ---- KEYWORD
(        ---- PARENTHESIS
i        ---- IDENTIFIER
=        ---- ARITHMETIC OPERATOR
0        ---- NUMBER CONSTANT
;        ---- SEMICOLON DELIMITER
i        ---- IDENTIFIER
<        ---- COMPARISON OPERATOR
n        ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
i        ---- IDENTIFIER
++       ---- ARITHMETIC OPERATOR
)        ---- PARENTHESIS
{        ---- PARENTHESIS
if       ---- KEYWORD
(        ---- PARENTHESIS
i        ---- IDENTIFIER
<        ---- COMPARISON OPERATOR
10       ---- NUMBER CONSTANT
)        ---- PARENTHESIS
{        ---- PARENTHESIS
int      ---- KEYWORD
x        ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
while    ---- KEYWORD
(        ---- PARENTHESIS
x        ---- IDENTIFIER
<        ---- COMPARISON OPERATOR
10       ---- NUMBER CONSTANT
)        ---- PARENTHESIS
{        ---- PARENTHESIS
printf   ---- IDENTIFIER
(        ---- PARENTHESIS
"%d\t"   ---- STRING
,        ---- COMMA DELIMITER
x        ---- IDENTIFIER
)        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
x        ---- IDENTIFIER
++       ---- ARITHMETIC OPERATOR
;        ---- SEMICOLON DELIMITER
}        ---- PARENTHESIS
}        ---- PARENTHESIS
else     ---- KEYWORD
printf   ---- IDENTIFIER
(        ---- PARENTHESIS
"Okay!\n"        ---- STRING
)        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
}        ---- PARENTHESIS
}        ---- PARENTHESIS
```



```
----------------------------------------------------------------------------
                               SYMBOL TABLE
----------------------------------------------------------------------------
       Lexeme                                              Token
----------------------------------------------------------------------------
            a                                         IDENTIFIER
            i                                         IDENTIFIER
            n                                         IDENTIFIER
            x                                         IDENTIFIER
         10.9                                    NUMBER CONSTANT
        scanf                                         IDENTIFIER
          for                                            KEYWORD
         char                                            KEYWORD
           ch                                         IDENTIFIER
           if                                            KEYWORD
   "Okay!\n"                                     STRING CONSTANT
          int                                            KEYWORD
      "%d\t"                                     STRING CONSTANT
        "%d"                                     STRING CONSTANT
         main                                         IDENTIFIER
          arr                                         IDENTIFIER
         else                                            KEYWORD
       printf                                         IDENTIFIER
           10                                    NUMBER CONSTANT
           20                                    NUMBER CONSTANT
        while                                            KEYWORD
            0                                    NUMBER CONSTANT
            1                                    NUMBER CONSTANT
```

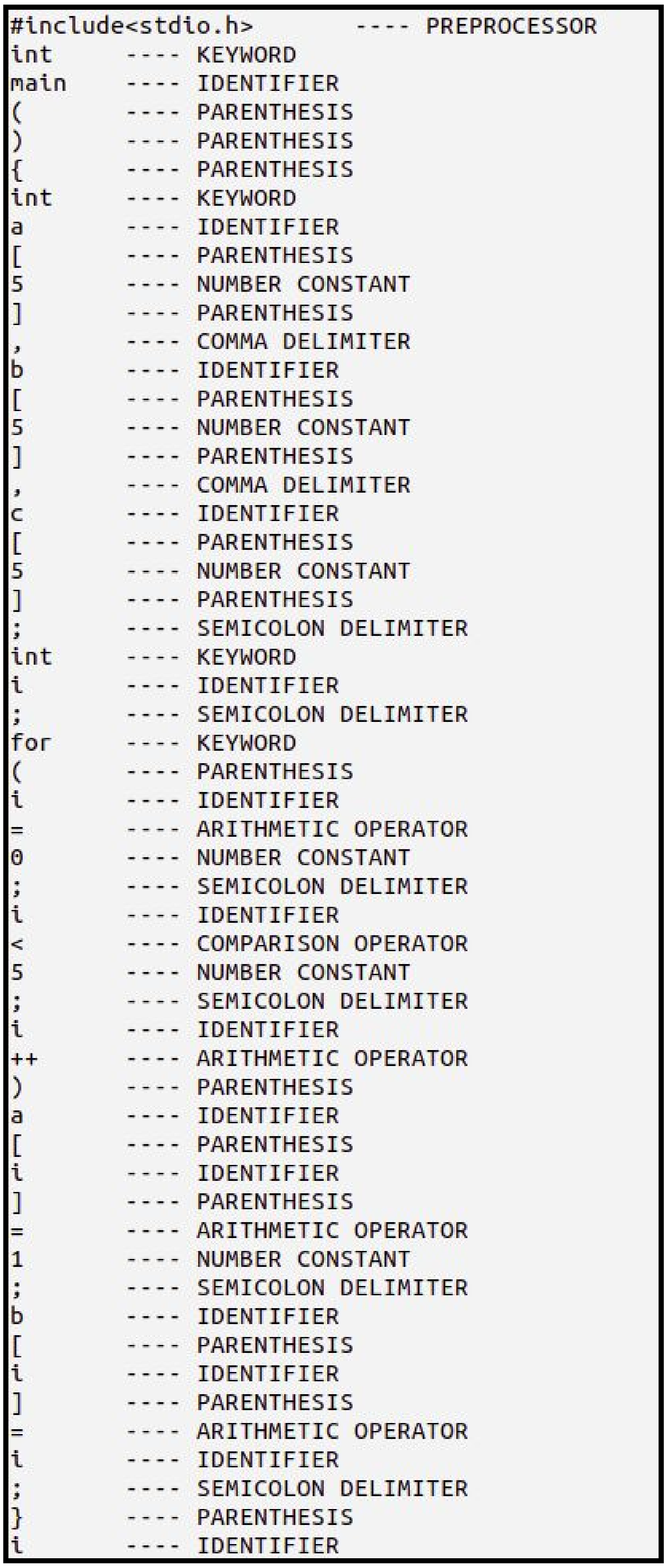
Test Case 3

Error of unbalanced parenthesis and unterminated comment.

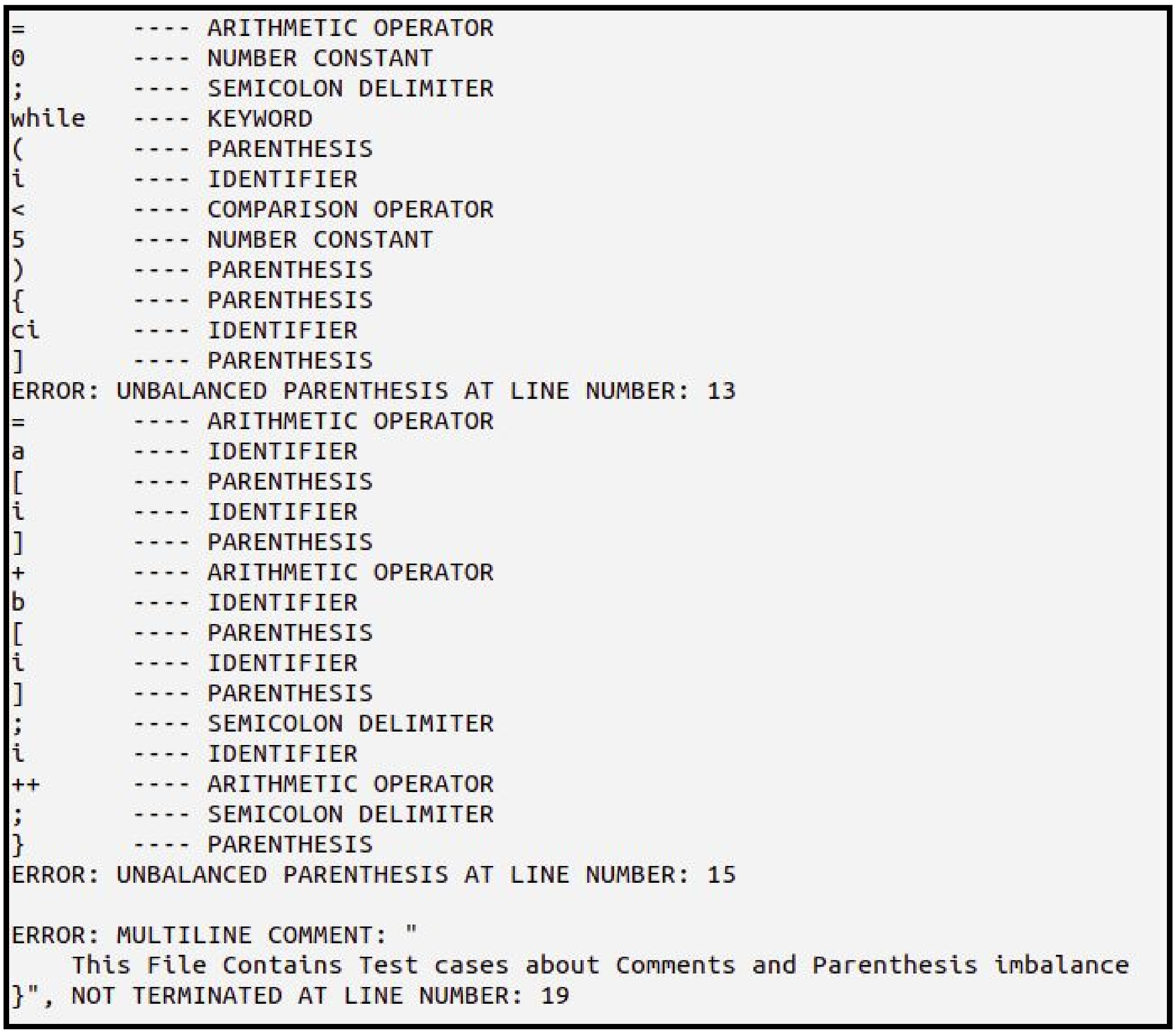


```
#include<stdio.h>
int main()
{
        int a[5],b[5],c[5];
        int i;
        for(i=0;i<5;i++)

                a[i]=1;
                b[i]=i;
        }
        i=0;
        while(i<5)
        {
                ci]=a[i]+b[i];
                i++;
        }
         /*
    This File Contains Test cases about Comments and Parenthesis imbalance
}
```

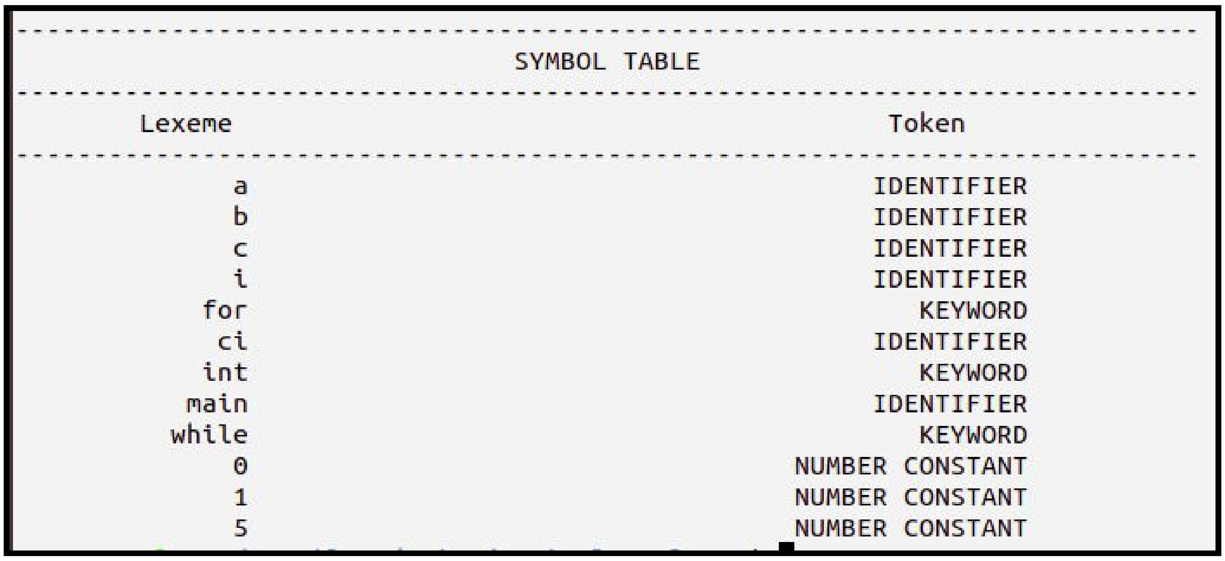
Output for Test Case 3



```
#include<stdio.h>       ---- PREPROCESSOR
int      ---- KEYWORD
main     ---- IDENTIFIER
(        ---- PARENTHESIS
)        ---- PARENTHESIS
{        ---- PARENTHESIS
int      ---- KEYWORD
a        ---- IDENTIFIER
[        ---- PARENTHESIS
5        ---- NUMBER CONSTANT
]        ---- PARENTHESIS
,        ---- COMMA DELIMITER
b        ---- IDENTIFIER
[        ---- PARENTHESIS
5        ---- NUMBER CONSTANT
]        ---- PARENTHESIS
,        ---- COMMA DELIMITER
c        ---- IDENTIFIER
[        ---- PARENTHESIS
5        ---- NUMBER CONSTANT
]        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
int      ---- KEYWORD
i        ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
for      ---- KEYWORD
(        ---- PARENTHESIS
i        ---- IDENTIFIER
=        ---- ARITHMETIC OPERATOR
0        ---- NUMBER CONSTANT
;        ---- SEMICOLON DELIMITER
i        ---- IDENTIFIER
<        ---- COMPARISON OPERATOR
5        ---- NUMBER CONSTANT
;        ---- SEMICOLON DELIMITER
i        ---- IDENTIFIER
++       ---- ARITHMETIC OPERATOR
)        ---- PARENTHESIS
a        ---- IDENTIFIER
[        ---- PARENTHESIS
i        ---- IDENTIFIER
]        ---- PARENTHESIS
=        ---- ARITHMETIC OPERATOR
1        ---- NUMBER CONSTANT
;        ---- SEMICOLON DELIMITER
b        ---- IDENTIFIER
[        ---- PARENTHESIS
i        ---- IDENTIFIER
]        ---- PARENTHESIS
=        ---- ARITHMETIC OPERATOR
i        ---- IDENTIFIER
;        ---- SEMICOLON DELIMITER
}        ---- PARENTHESIS
i        ---- IDENTIFIER
```



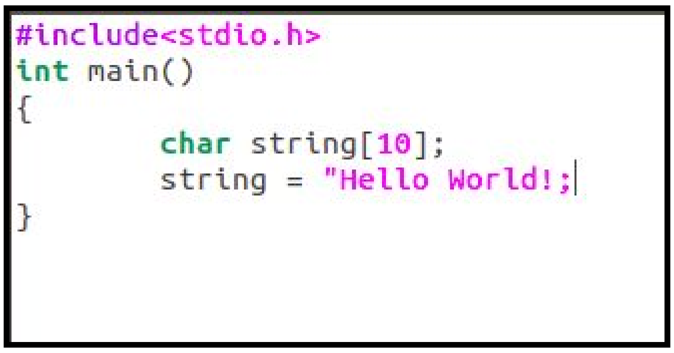
```
=        ---- ARITHMETIC OPERATOR
0        ---- NUMBER CONSTANT
;        ---- SEMICOLON DELIMITER
while    ---- KEYWORD
(        ---- PARENTHESIS
i        ---- IDENTIFIER
<        ---- COMPARISON OPERATOR
5        ---- NUMBER CONSTANT
)        ---- PARENTHESIS
{        ---- PARENTHESIS
ci       ---- IDENTIFIER
]        ---- PARENTHESIS
ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER: 13
=        ---- ARITHMETIC OPERATOR
a        ---- IDENTIFIER
[        ---- PARENTHESIS
i        ---- IDENTIFIER
]        ---- PARENTHESIS
+        ---- ARITHMETIC OPERATOR
b        ---- IDENTIFIER
[        ---- PARENTHESIS
i        ---- IDENTIFIER
]        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
i        ---- IDENTIFIER
++       ---- ARITHMETIC OPERATOR
;        ---- SEMICOLON DELIMITER
}        ---- PARENTHESIS
ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER: 15

ERROR: MULTILINE COMMENT: "
    This File Contains Test cases about Comments and Parenthesis imbalance
}", NOT TERMINATED AT LINE NUMBER: 19
```



```
---------------------------------------------------------------------------
                               SYMBOL TABLE
---------------------------------------------------------------------------
       Lexeme                                          Token
---------------------------------------------------------------------------
             a                                    IDENTIFIER
             b                                    IDENTIFIER
             c                                    IDENTIFIER
             i                                    IDENTIFIER
           for                                       KEYWORD
            ci                                    IDENTIFIER
           int                                       KEYWORD
          main                                    IDENTIFIER
         while                                       KEYWORD
             0                               NUMBER CONSTANT
             1                               NUMBER CONSTANT
             5                               NUMBER CONSTANT
```

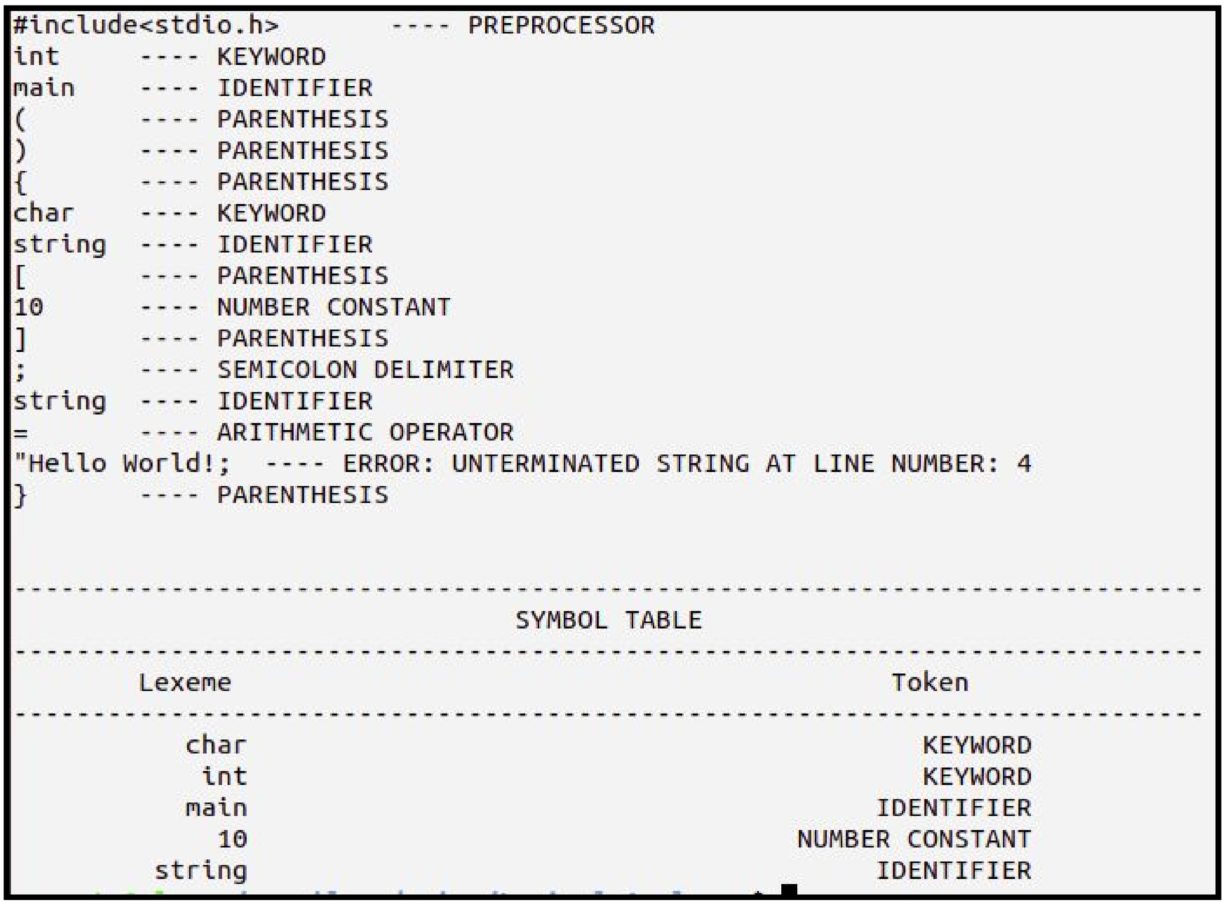
Test Case 4

Error in string literal.



```
#include<stdio.h>
int main()
{
	char string[10];
	string = "Hello World!;
}
```

Output for Test Case 4



```
#include<stdio.h>       ---- PREPROCESSOR
int      ---- KEYWORD
main     ---- IDENTIFIER
(        ---- PARENTHESIS
)        ---- PARENTHESIS
{        ---- PARENTHESIS
char     ---- KEYWORD
string   ---- IDENTIFIER
[        ---- PARENTHESIS
10       ---- NUMBER CONSTANT
]        ---- PARENTHESIS
;        ---- SEMICOLON DELIMITER
string   ---- IDENTIFIER
=        ---- ARITHMETIC OPERATOR
"Hello World!;   ---- ERROR: UNTERMINATED STRING AT LINE NUMBER: 4
}        ---- PARENTHESIS


----------------------------------------------------------------------------
                                SYMBOL TABLE
----------------------------------------------------------------------------
        Lexeme                                              Token
----------------------------------------------------------------------------
           char                                               KEYWORD
            int                                               KEYWORD
           main                                            IDENTIFIER
             10                                       NUMBER CONSTANT
         string                                            IDENTIFIER
```

Implementation

Regular expression for identifiers: The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.

```
[a-z|A-Z]([a-z|A-Z]|[0-9])*
```

Multiline comments: This has been supported by checking the
occurence of ‘/*’ and ‘*/’ in the code. The statements between them has been excluded.
Errors for unmatched and nested comments have also been displayed.

Error Handling for Incomplete String: Open and close quote missing, both kind of
errors have been handled in the rules written in the script.

Error Handling for Nested Comments: This use-case has been handled by checking for occurrence of multiple successive ‘/*’ or ‘*/’ in the C code, and by omitting the text in between them.

At the end of the token recognition, the lexer prints a list of all the tokens present in the program. As and when successive tokens are encountered, their respective values are stored in the symbol table structure and then later displayed.

Future Work

The lexical analyser that was created in this project helps in breaking source program into tokens define by the C programming language.

In the next phase, parser will be designed which will call upon the Flex program to give it tokens and the lexical analyzer will return to the parser the integer value associated with the tokens as and when required by the parser.

Together with the symbol table, the parser will prepare a syntax tree with the help of a grammar that we provide it with. The parser can then logically group the tokens to form meaningful statements and can detect C programming constructs such as arrays, loops, and functions. The parser will also help us identify errors that could not be detected in the lexical analysis phase such as unbalanced parentheses, unterminated statements, missing operators, two operators in a row, etc.

References

- Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- https://www.geeksforgeeks.org/cc-tokens/
- http://www.isi.edu/~pedro/Teaching/CSCI565-Spring11/Practice/SDT-Sample.pdf
- StackOverflow for regex