GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: **System Programming & Compiler Construction**

Experiment No: 4

Title of Experiment: Implementation of Lexical Analyzer.

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	10	
2	Oral	5	
	Total	15	

Signature of Subject Teacher

Theory:

Lexical Analyzer

A lexical analyzer is a crucial component in the construction of a compiler. Its main task is to scan the source code of a program and break it down into a sequence of tokens or lexemes. These tokens are then used by the parser to build a parse tree representing the structure of the program.

The process of lexical analysis involves several steps. First, the input source code is read character by character, and these characters are grouped together into words or symbols. Next, the lexical analyzer identifies the type of each word or symbol and assigns it a token type. For example, the word "if" might be assigned a token type of "IF_KEYWORD," while the symbol "+" might be assigned a token type of "ADD_OPERATOR."

In addition to assigning token types, the lexical analyzer may also perform some basic checks on the input source code. For example, it may check that all identifiers are declared before they are used, that all numeric literals are within the range of valid values for their data type, and that all strings are properly terminated with a closing quote.

The output of the lexical analyzer is a sequence of tokens, each representing a single unit of meaning in the source code. These tokens are then passed on to the parser, which uses them to build a parse tree representing the structure of the program.

One of the key benefits of using a lexical analyzer in compiler construction is that it simplifies the input to the compiler. By breaking down the input source code into a sequence of tokens, it makes it easier for the other components of the compiler to process the input and generate the output executable code.

Overall, the lexical analyzer is an essential component in the construction of a compiler. It plays a critical role in simplifying the input to the compiler and ensuring that the output executable code is correct and efficient.

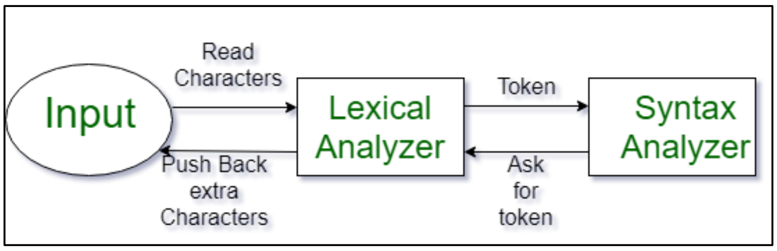
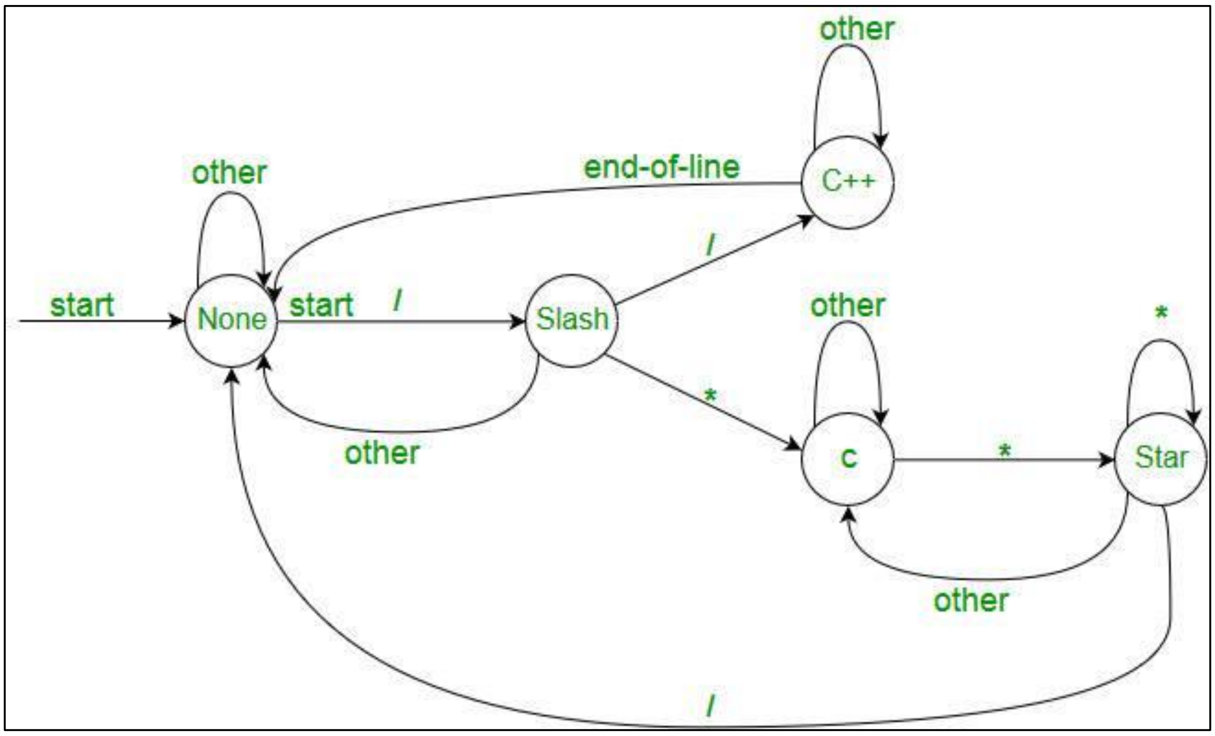


Fig. Lexical Analysis

How Lexical Analyzer works-

1. Input preprocessing: This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.
2. Tokenization: This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.
3. Token classification: In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.
4. Token validation: In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.
5. Output generation: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.



Program Code –

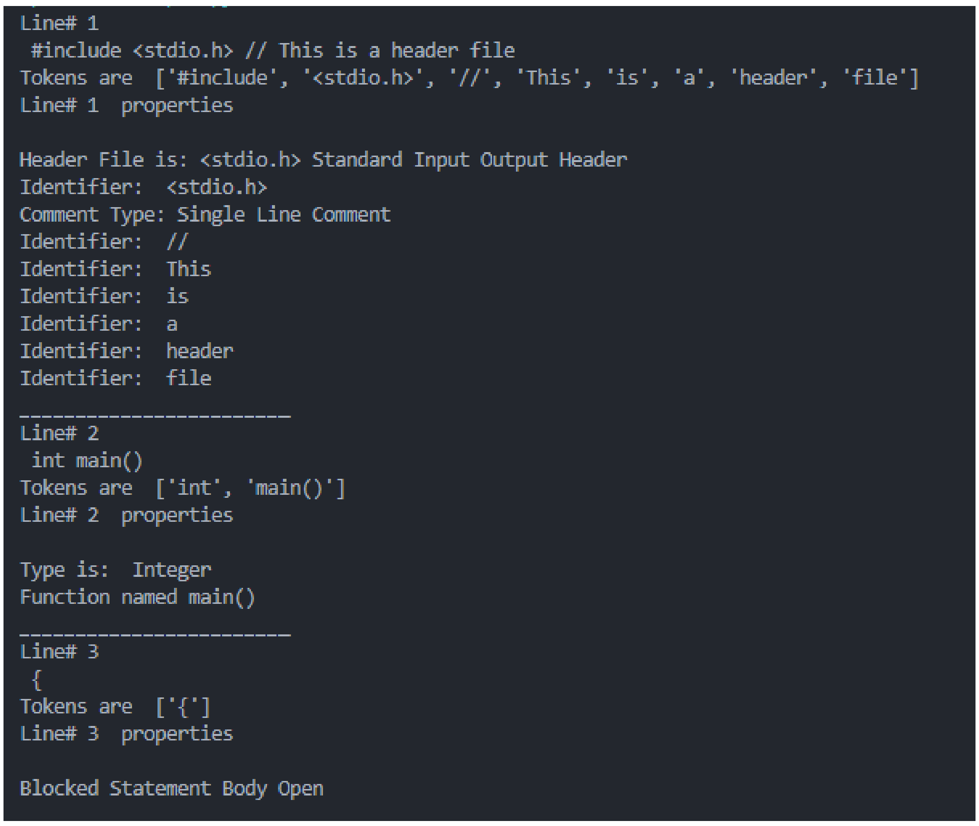
```
import re


f = open("Expt4.c", "r")

operators = {"=": "Assignment Operator","+": "Additon Operator","-":
"Substraction Operator","/": "Division Operator","*": "Multiplication
Operator","++": "increment Operator","--": "Decrement Operator"}

optr_keys = operators.keys()

comments = {r"//": "Single Line Comment",r"/*": "Multiline Comment
Start",r"*/": "Multiline Comment End","/**/": "Empty Multiline comment"}

comment_keys = comments.keys()

header = {".h": "headerfile"}

header_keys = header.keys()

sp_header_files = {"<stdio.h>": "Standard Input Output
Header","<string.h>": "String Manipulation Library",

}

macros = {r"#\w+": "macro"}

macros_keys = macros.keys()

datatype = {"int": "Integer","float": "Floating Point","char":
"Character","long": "longint"}

datatype_keys = datatype.keys()

keyword = {"return": "keyword that returns a value from a block"}

keyword_keys = keyword.keys()

delimiter = {";": "terminator symbol semi colon(;)"}

delimiter_keys = delimiter.keys()

blocks = {"{": "Blocked Statement Body Open", "}": "Blocked Statement Body
Closed"}

block_keys = blocks.keys()

builtin_functions = {"printf": "printf prints its argument on the
console"}

non_identifiers = ["_","-
","+","/","*","`","~","!","@","#","$","%","^","&","*","(",")","=","|",'"
',":",";","{","}","[","]","<",">","?","/",]
```

```
numerals = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
# Flags
dataFlag = False
i = f.read()
count = 0
program = i.split("\n")
for line in program:
    count = count + 1
    print("Line#", count, "\n", line)
    tokens = line.split(" ")
    print("Tokens are ", tokens)
    print("Line#", count, " properties\n")
    for token in tokens:
        if "\r" in token:
            position = token.find("\r")
            token = token[:position]
        # print1
        if token in block_keys:
            print(blocks[token])
        if token in optr_keys:
            print("Operator is:", operators[token])
        if token in comment_keys:
            print("Comment Type:", comments[token])
        if token in macros_keys:
            print("Macro is:", macros[token])
        if ".h" in token:
            print("Header File is:", token, sp_header_files[token])
        if "()" in token:
            print("Function named", token)
        if dataFlag == True and (token not in non_identifiers) and ("()" not in token):
            print("Identifier: ", token)
        if token in datatype_keys:
            print("Type is: ", datatype[token])
        dataFlag = True
        if token in keyword_keys:
            print(keyword[token])
        if token in delimiter:
            print("Delimiter", delimiter[token])
        if "#" in token:
            match = re.search(r"#\w+", token)
        if token in numerals:
            print(token, type(int(token)))
    dataFlag = False
    print("________________________")
f.close()
```

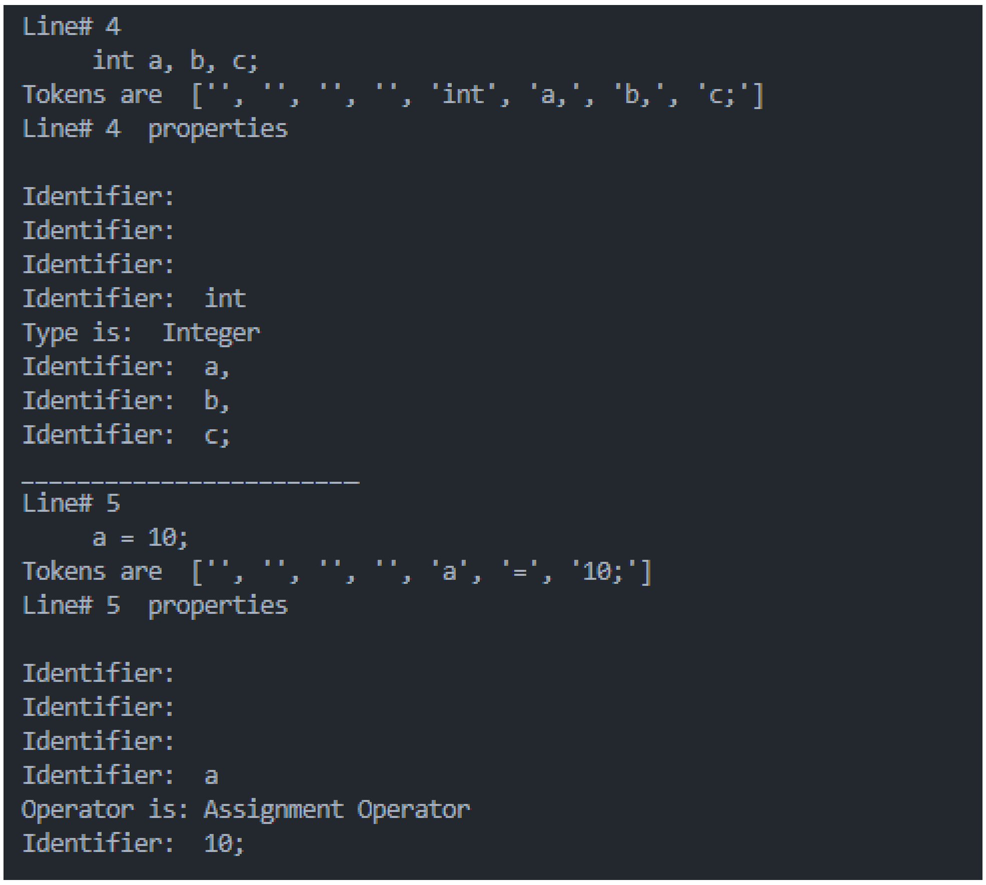
Input File

```
#include <stdio.h> // This is a header file
int main()
{
    int a, b, c;
    a = 10;
    b = 10;
    c = a + b;
    printf("The addition is %d", c);
    return0;
}
```

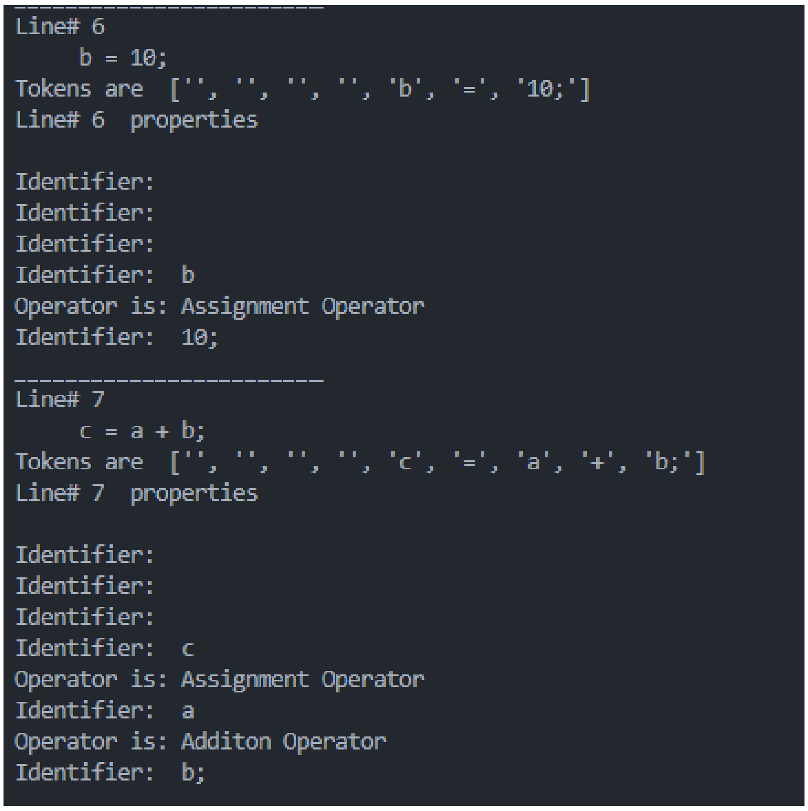
Output –



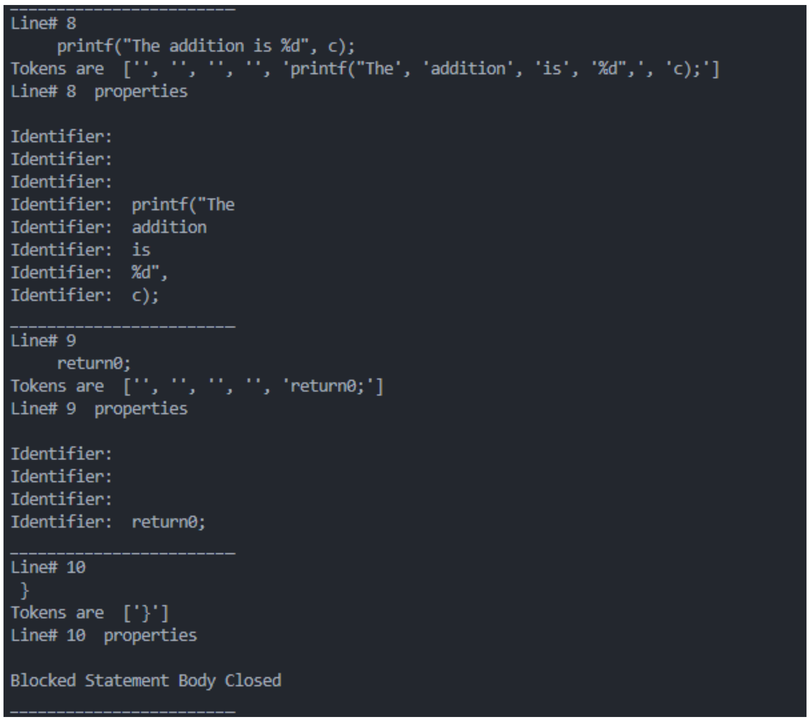
```
Line# 1
 #include <stdio.h> // This is a header file
Tokens are  ['#include', '<stdio.h>', '//', 'This', 'is', 'a', 'header', 'file']
Line# 1  properties

Header File is: <stdio.h> Standard Input Output Header
Identifier:  <stdio.h>
Comment Type: Single Line Comment
Identifier:  //
Identifier:  This
Identifier:  is
Identifier:  a
Identifier:  header
Identifier:  file

________________________
Line# 2
 int main()
Tokens are  ['int', 'main()']
Line# 2  properties

Type is:  Integer
Function named main()

________________________
Line# 3
 {
Tokens are  ['{']
Line# 3  properties

Blocked Statement Body Open
```



```
Line# 4
     int a, b, c;
Tokens are  ['', '', '', '', 'int', 'a,', 'b,', 'c;']
Line# 4  properties

Identifier:
Identifier:
Identifier:
Identifier:  int
Type is:  Integer
Identifier:  a,
Identifier:  b,
Identifier:  c;

________________________
Line# 5
     a = 10;
Tokens are  ['', '', '', '', 'a', '=', '10;']
Line# 5  properties

Identifier:
Identifier:
Identifier:
Identifier:  a
Operator is: Assignment Operator
Identifier:  10;
```



```
-------------------------
Line# 6
     b = 10;
Tokens are  ['', '', '', '', 'b', '=', '10;']
Line# 6  properties

Identifier:
Identifier:
Identifier:
Identifier:  b
Operator is: Assignment Operator
Identifier:  10;

-------------------------
Line# 7
     c = a + b;
Tokens are  ['', '', '', '', 'c', '=', 'a', '+', 'b;']
Line# 7  properties

Identifier:
Identifier:
Identifier:
Identifier:  c
Operator is: Assignment Operator
Identifier:  a
Operator is: Additon Operator
Identifier:  b;
```



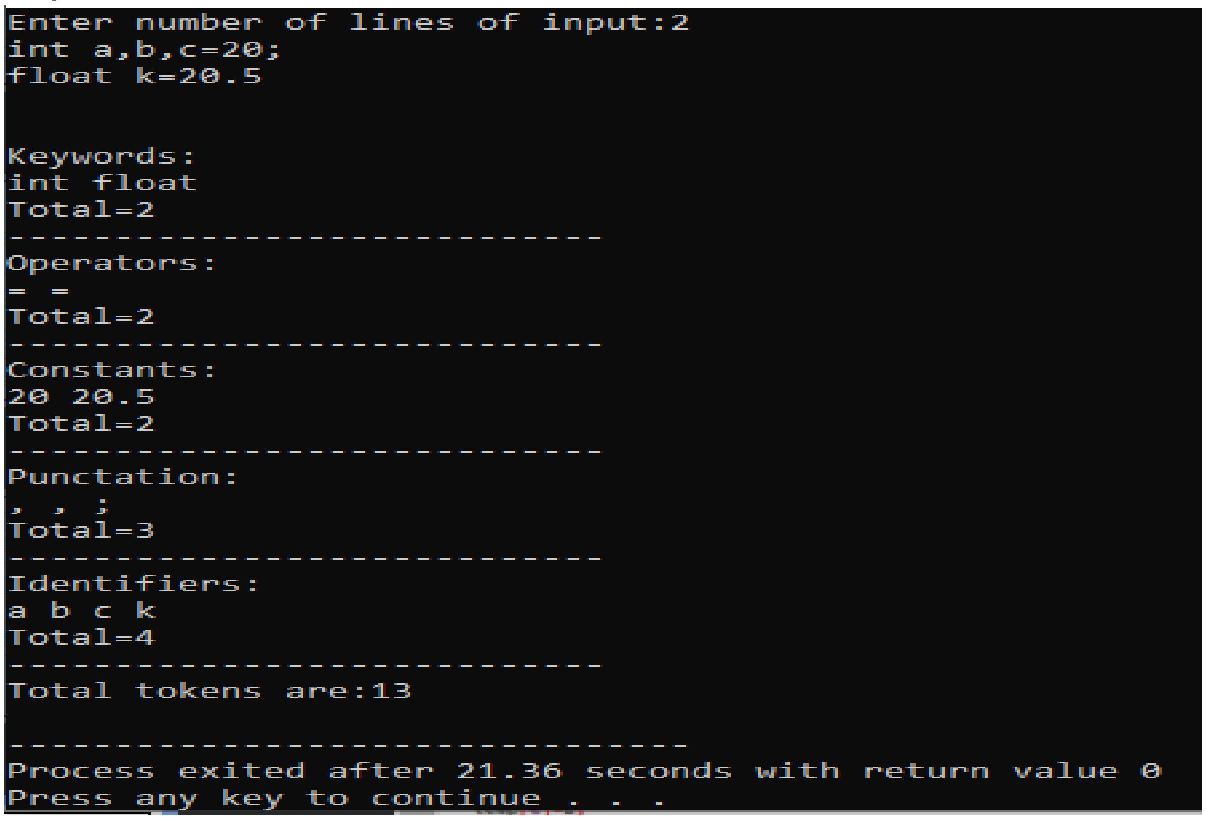
```
-------------------------
Line# 8
     printf("The addition is %d", c);
Tokens are  ['', '', '', '', 'printf("The', 'addition', 'is', '%d",', 'c);']
Line# 8  properties

Identifier:
Identifier:
Identifier:
Identifier:  printf("The
Identifier:  addition
Identifier:  is
Identifier:  %d",
Identifier:  c);

-------------------------
Line# 9
     return0;
Tokens are  ['', '', '', '', 'return0;']
Line# 9  properties

Identifier:
Identifier:
Identifier:
Identifier:  return0;

-------------------------
Line# 10
 }
Tokens are  ['}']
Line# 10  properties

Blocked Statement Body Closed

-------------------------
```

Program-

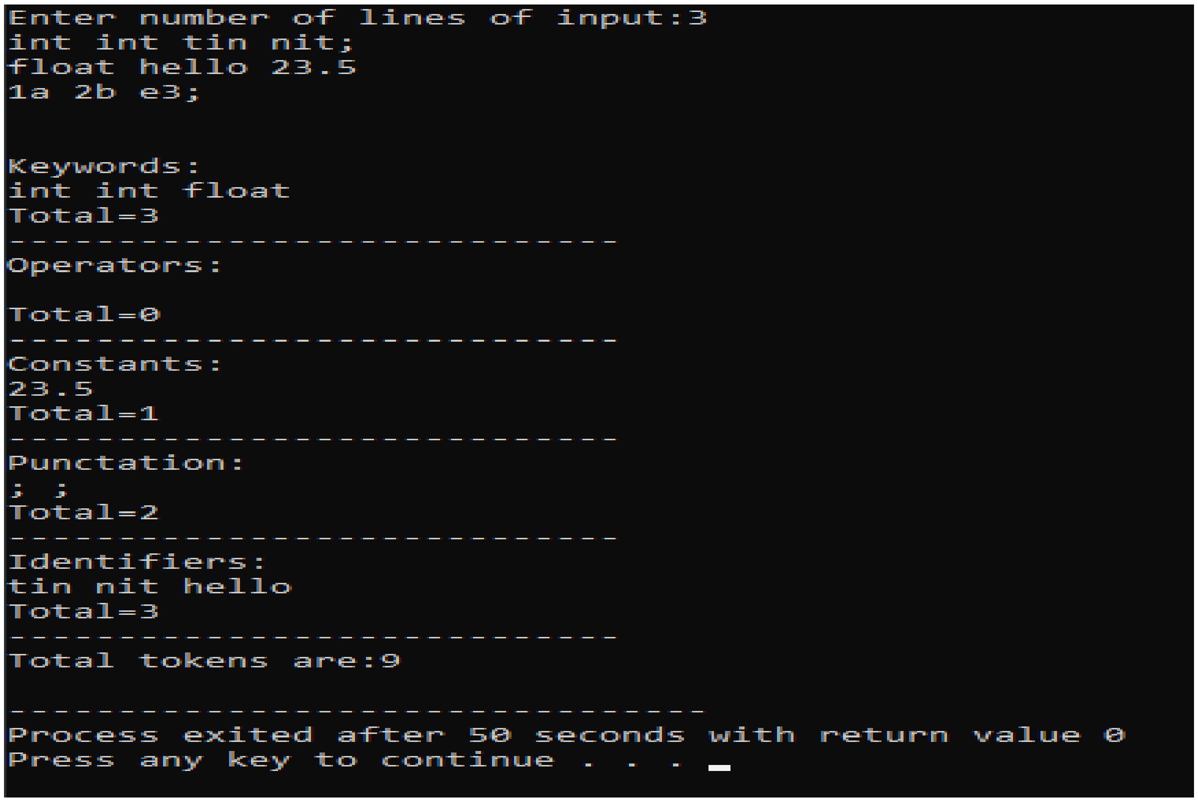
```
#include <bits/stdc++.h>
using namespace std;
string keywords[]={"int","float","if","else","while","for"};
string operators[]={"<",">","<=",">=","==","=","+","-","++","--"};
char punctuation[]={'(',')',',',';','{','}','[',']'};
vector<string> k,o,c,i;
vector<string> p;
void search_for_keywords(string a ,int flag)
{
if(a.size()==0)return;
int size=sizeof(keywords)/sizeof(keywords[0]);
for(int i=0;i<size;i++)
{
if(a==keywords[i])
{
k.push_back(a);
return;
}
}
if(!flag) i.push_back(a);
}
bool search_for_punctuation(char a )
{
//cout<<a;
int size=sizeof(punctuation)/sizeof(punctuation[0]);
for(int i=0;i<size;i++)
{
//cout<<punctuation[i];
if(a==punctuation[i])
{
//cout<<a;
string temp=" ";
temp[0]=a;
//cout<<temp;
p.push_back(temp);
return true;
}
}
return false;
}

void search_for_constants(string a )
{
if(a.size()>0) c.push_back(a);
}

void print(vector<string>a )
{
cout<<"\n";//cout<<"----------------------------------------\n";
for(int i=0;i<a.size();i++)
```

```
{
cout<<a[i]<<" ";
}
cout<<"\nTotal="<<a.size()<<"\n";
cout<<"---------------------------\n";
}

void search_for_operators(string line,int& i)
{
// This is to check the operators which are composed of two characters like '++', '+='.
string temp=line.substr(i,2);
//cout<<temp<<endl;
int size=sizeof(operators)/sizeof(operators[0]);
for(int j=0;j<size;j++)
{
//cout<<punctuation[i];
if(temp==operators[j])
{
o.push_back(temp);i=i+1;
return;
}
}
// This is to check the operators which are composed of only one character like '+', '-'.
temp=line.substr(i,1);
//cout<<temp<<endl;
for(int j=0;j<size;j++)
{
//cout<<punctuation[i];
if(temp==operators[j])
{
o.push_back(temp);
return;
}
}
}
int main()
{
cout<<"Enter number of lines of input:";
int n;
cin>>n;n++;
while(n--)
{
char arr[100];
cin.getline(arr,100,'\n');
string line=arr;
//cout<<line<<line.length();
int i;
string cur="";
string cur_num="";
int flag=0,flag2=0;
int no_of_dots=0;
for(i=0;i<line.length();i++)
{
char now=line[i];
if((now>='a'&&now<='z')||(now>='A'&&now<='Z'))
{
// Check for keywords and identifiers starts.
if(now=='e'&&flag==0&&no_of_dots>0){cur_num+=now;continue;}
if(cur_num.size()>0){flag2=1;}
flag=1;
cur+=line[i];
}
else if(now==' '){
if(flag)search_for_keywords(cur,flag2);
else if(!flag2) search_for_constants(cur_num);
cur="";flag=0;cur_num="";no_of_dots=0;
}
else if(now>='0'&&now<='9'||now=='.')
{

if(now=='.'&&no_of_dots>0)cur_num="";
else if(now=='.')no_of_dots++;
if(flag)cur+=line[i];
else cur_num+=line[i];
}
else{
//cout<<now;
// If none of the above conditions pass, this block of code checks for all of the keywords,
numbers, operators and punctuations.

if((now=='+'||now=='-')&&flag==0&&cur_num.size()>0){cur_num+=now;continue;}
if(flag)search_for_keywords(cur,flag2);
else if(!flag2) search_for_constants(cur_num);
cur="";flag=0;cur_num="";no_of_dots=0;
if(!search_for_punctuation(now))
search_for_operators(line,i);
;//cout<<now;
}
}
if(flag)search_for_keywords(cur,flag2);
else if(!flag2) search_for_constants(cur_num);
cur="";flag=0;cur_num="";}
cout<<"\n\nKeywords:";
print(k);cout<<"Operators:";
print(o);cout<<"Constants:";
print(c);cout<<"Punctation:";
print(p);cout<<"Identifiers:";
print(::i);
cout<<"Total tokens are:"<<k.size()+o.size()+c.size()+p.size()+::i.size()<<"\n";
return 0;
}
```

Output-



```
Enter number of lines of input:2
int a,b,c=20;
float k=20.5


Keywords:
int float
Total=2
----------------------------
Operators:
= =
Total=2
----------------------------
Constants:
20 20.5
Total=2
----------------------------
Punctation:
, , ;
Total=3
----------------------------
Identifiers:
a b c k
Total=4
----------------------------
Total tokens are:13

--------------------------------
Process exited after 21.36 seconds with return value 0
Press any key to continue . . .
```



```
Enter number of lines of input:3
int int tin nit;
float hello 23.5
1a 2b e3;


Keywords:
int int float
Total=3
----------------------------
Operators:

Total=0
----------------------------
Constants:
23.5
Total=1
----------------------------
Punctation:
; ;
Total=2
----------------------------
Identifiers:
tin nit hello
Total=3
----------------------------
Total tokens are:9

--------------------------------
Process exited after 50 seconds with return value 0
Press any key to continue . . .
```

Conclusion

Thus we studied the implementation of Lexical Analyzer.