Date : 03/02/2022

CSPC62 : COMPILER DESIGN
LAB-1

Roll no. : **106119100**

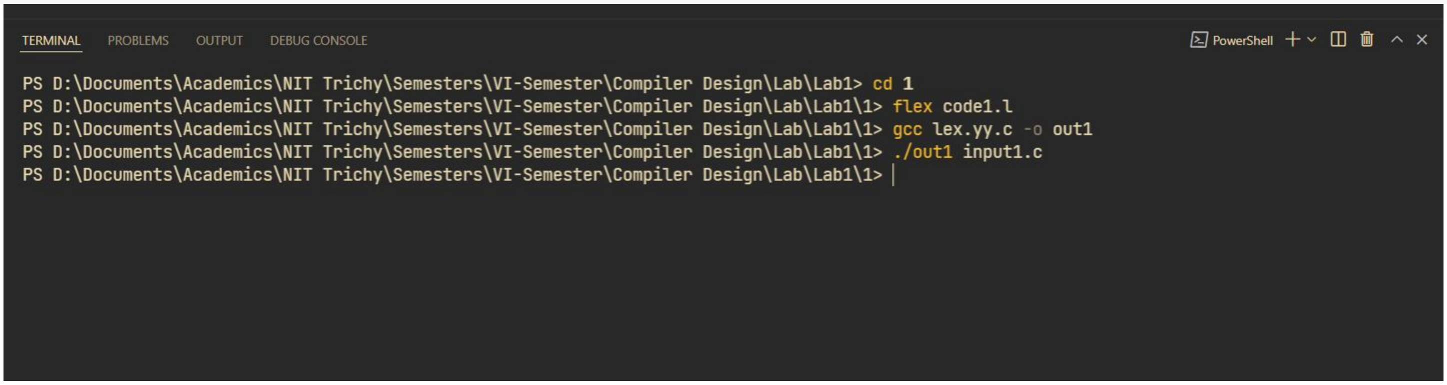
Name : **Rajneesh Pandey**

Section : **CSE-B**

1. Design a lexical analyzer that could ignore redundant spaces, tabs, new lines, and comments in a source program (C language).

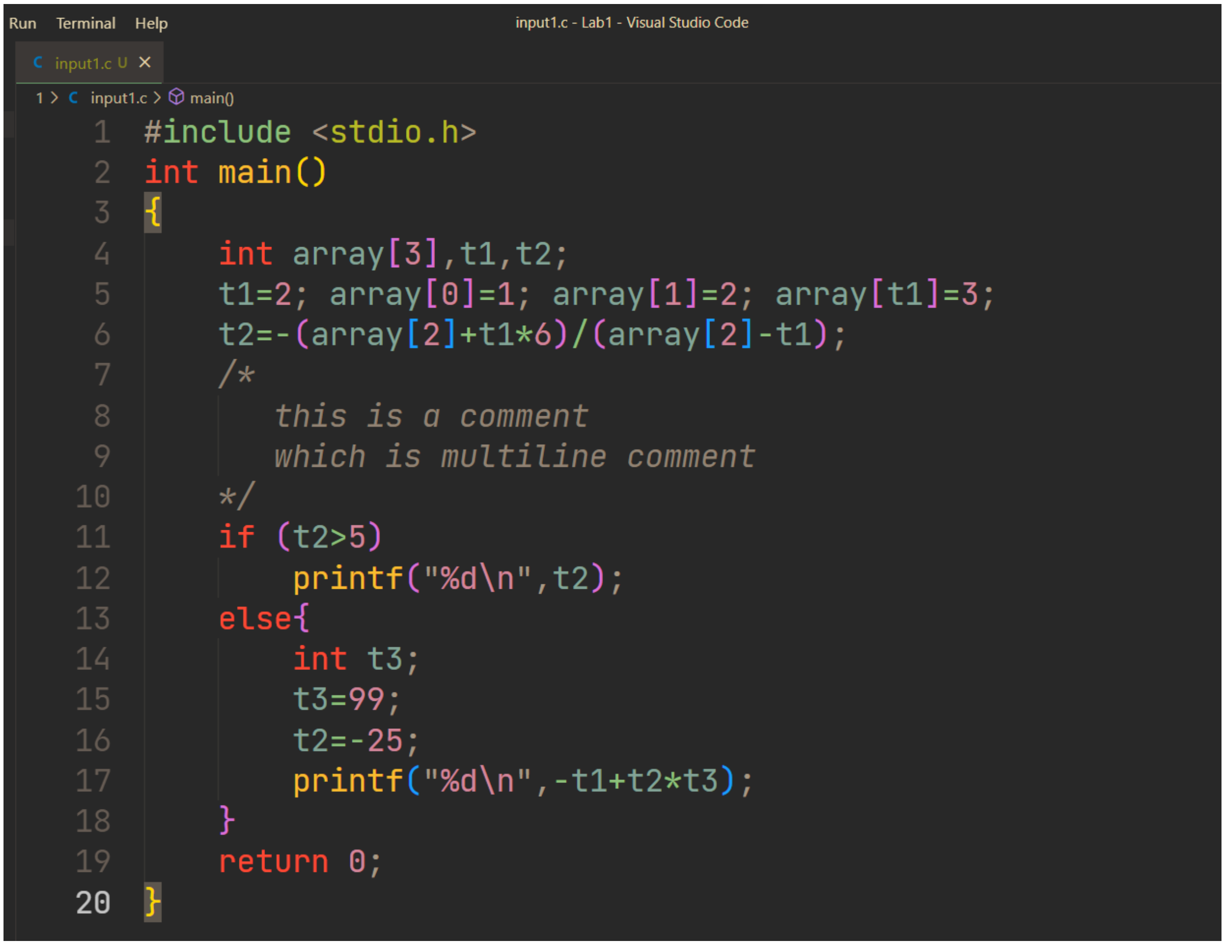
Code:

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


%%


\/\/(.*)\n ;
\/\*[^*/]*\*\/\n ;
([\t]|" ")*\n+ {fprintf(yyout,"\n");}
\t ;
" "+ {fprintf(yyout," ");}


%%


int yywrap(){return 1;}



int main(int k,char **argcv)
{
  yyin=fopen(argcv[1],"r");
  yyout=fopen("out1.c","w");
  yylex();
  return 0;
}
```



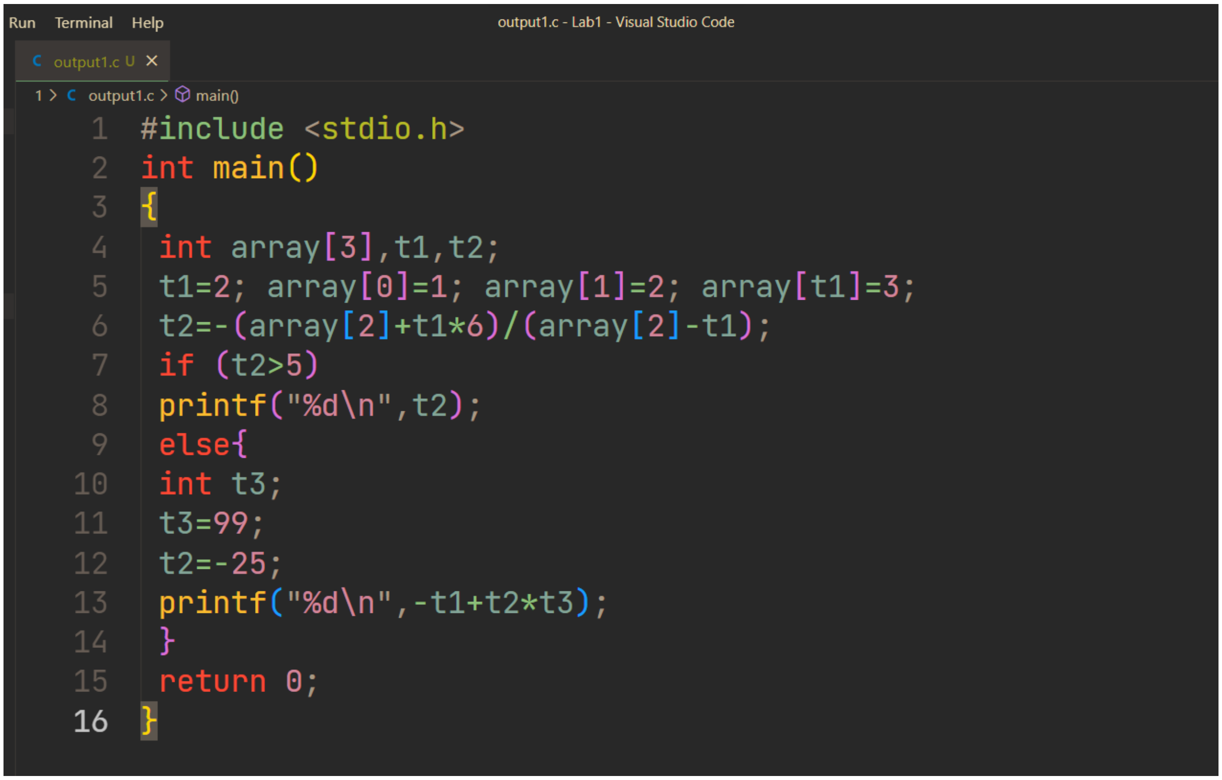
```
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1> cd 1
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\1> flex code1.l
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\1> gcc lex.yy.c -o out1
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\1> ./out1 input1.c
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\1> |
```

Input:



```c
#include <stdio.h>
int main()
{
    int array[3],t1,t2;
    t1=2; array[0]=1; array[1]=2; array[t1]=3;
    t2=-(array[2]+t1*6)/(array[2]-t1);
    /*
        this is a comment
        which is multiline comment
    */
    if (t2>5)
        printf("%d\n",t2);
    else{
        int t3;
        t3=99;
        t2=-25;
        printf("%d\n",-t1+t2*t3);
    }
    return 0;
}
```

Output:



```c
#include <stdio.h>
int main()
{
 int array[3],t1,t2;
 t1=2; array[0]=1; array[1]=2; array[t1]=3;
 t2=-(array[2]+t1*6)/(array[2]-t1);
 if (t2>5)
 printf("%d\n",t2);
 else{
 int t3;
 t3=99;
 t2=-25;
 printf("%d\n",-t1+t2*t3);
 }
 return 0;
}
```

2. Read an input C file. Design a lexical analyzer that could recognize keywords, identifiers and numeric data which is valid in C language. You may restrict the length of identifiers to some reasonable value (like 32). Display appropriate message if identifiers are not valid or it is too lengthy. List out the token names along with the recognized lexemes. Construct a symbol table which holds information (name, datatype, offset, size, scope) on valid identifiers.

Code

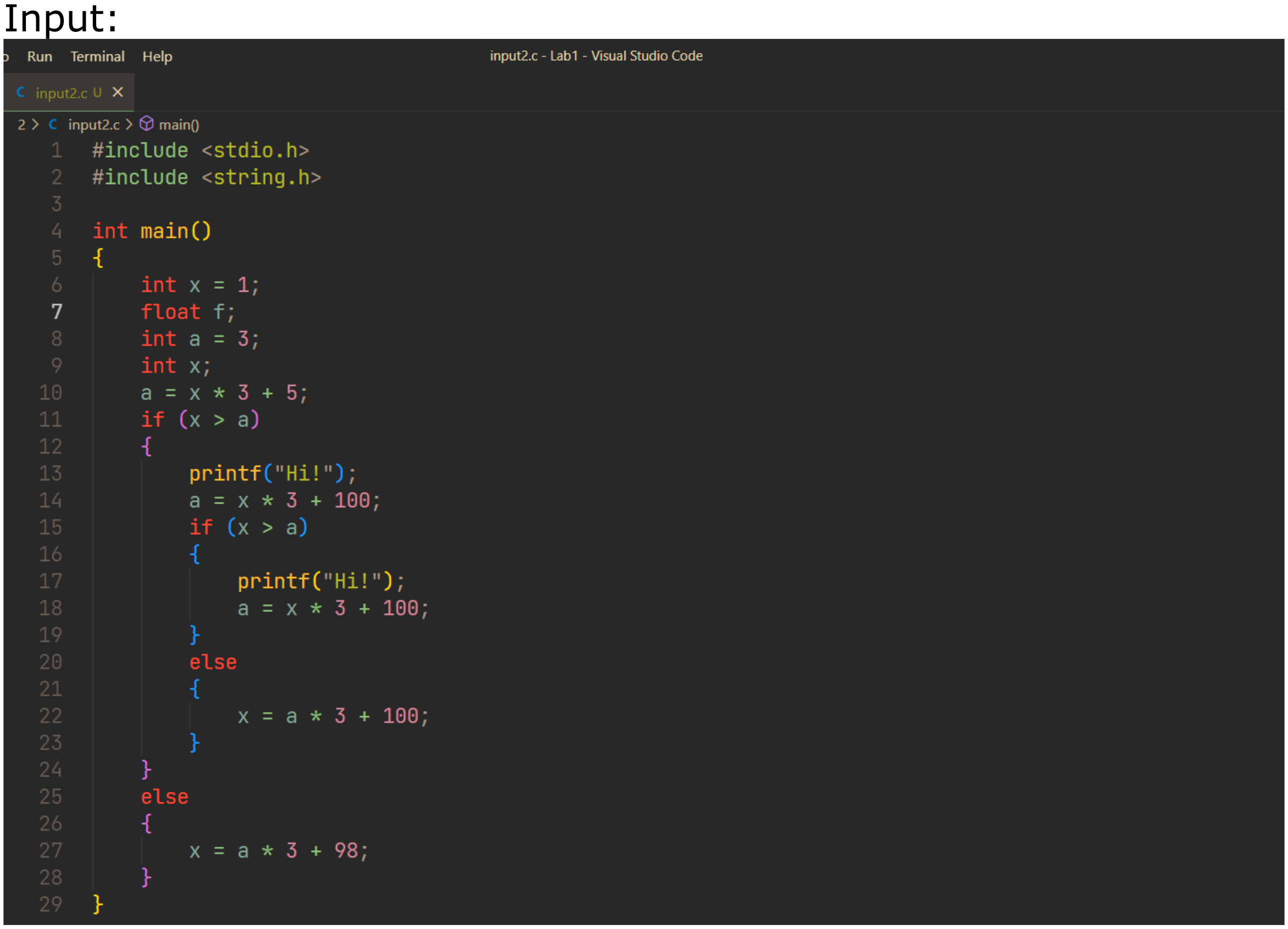
```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


char  keywords[22][10] =
{"if","else","while","do","break","continue","int","double","float
","return","char", "case", "sizeof", "long",
"short","typedef","switch","unsigned","void", "static", "struct",
"goto"};

// Parsing the input STRING.
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    //delimitors
    char delimiter[] = {" +-*/,;><=()[]{}\n\""};
    char operators[] = {"+-*/><=&"};
    int operlen = strlen(operators);
    int dellen = strlen(delimiter);
    while (right <= len && left <= right) {
        if(str[left] == '/' && str[left + 1] == '/'){
          break;
        }
        // check for string right to be delimiter
        bool isEndDel = false;
        for(int i=0;i<dellen;i++){
            if(delimiter[i] == str[right]){
                isEndDel = true;
                break;
            }
        }
        if(isEndDel == false)
            right++;
        isEndDel = false;
        for(int i=0;i<dellen;i++){
            if(delimiter[i] == str[right]){
                isEndDel = true;
                break;
            }
        }
        if (str[left] == '\"') {
          if(left == right)
          right++;
          while (str[right] != '\"') {
            right++;
          }
          right++;
          char* word = (char *) malloc((right - left +
2)*sizeof(char));
          strncpy(word, str + left , right - left);
          printf("'%s' IS A STRING literal\n", word);
          left = right;
          continue;
        }
        if (isEndDel && (left == right)) {
            // check for operators
            for(int i=0;i<operlen;i++){
                if(str[right] == operators[i]){
                    printf("'%c' IS AN OPERATOR\n", str[right]);
                    break;
                }
            }
```

```
      right++;
      left = right;
    } else if ((isEndDel == true && left != right) || (right
== len && left != right)) {
      char* word = (char *) malloc((right - left +
2)*sizeof(char));
      strncpy(word, str + left, right - left);
      word[right-left ] = '\0';
      int lengt = right-left;
      bool isDone = false;
      for(int i=0;i<22;i++){
        if(!strcmp(keywords[i], word) && isDone == false){
          printf("'%s' IS A KEYWORD\n", word);
          isDone = true;
        }
      }
      if(isDone == false) {
        int i = 0;
        if (word[0] == '-') {
          i = 1;
        }
        bool isint = true;
        for (; i < lengt; i++) {
          if (word[i] < '0' || word[i] > '9') {
            isint = false;
            break;
          }
        }
        if (isint) {
          printf("'%s' IS AN INTEGER\n", word);
          isDone = true;
        }
      }
      if(isDone == false) {
        int i = 0;
        bool isfloat = false;
        if (word[0] == '-')
```

```
                    i++;
                for (; i < lengt; i++) {
                    if ((word[i] < '0' || word[i] > '9') &&
word[i] != '.')
                        break;
                    if (word[i] == '.')
                        isfloat = true;
                }
                if (isfloat) {
                    printf("'%s' IS A REAL NUMBER\n", word);
                    isDone = true;
                }
            }
            if(isDone == false){
                bool isDelEnd = false;
                for(int i=0;i<dellen;i++){
                    if(delimiter[i] == str[right - 1]){
                        isDelEnd = true;
                        break;
                    }
                }
                bool isIdenti = true;
                for(int i=0;i<dellen;i++){
                    if(delimiter[i] == word[0] || (word[i] >= '0'
&& word[i] <= '9')){
                        isIdenti = false;
                        break;
                    }
                }
                if (isIdenti && !isDelEnd  && strlen(word) != 0)
                    printf("'%s' IS A VALID IDENTIFIER\n", word);

                else if (isIdenti == false && !isDelEnd)
                    printf("'%s' IS NOT A VALID IDENTIFIER\n",
word);
            }
            left = right;
```

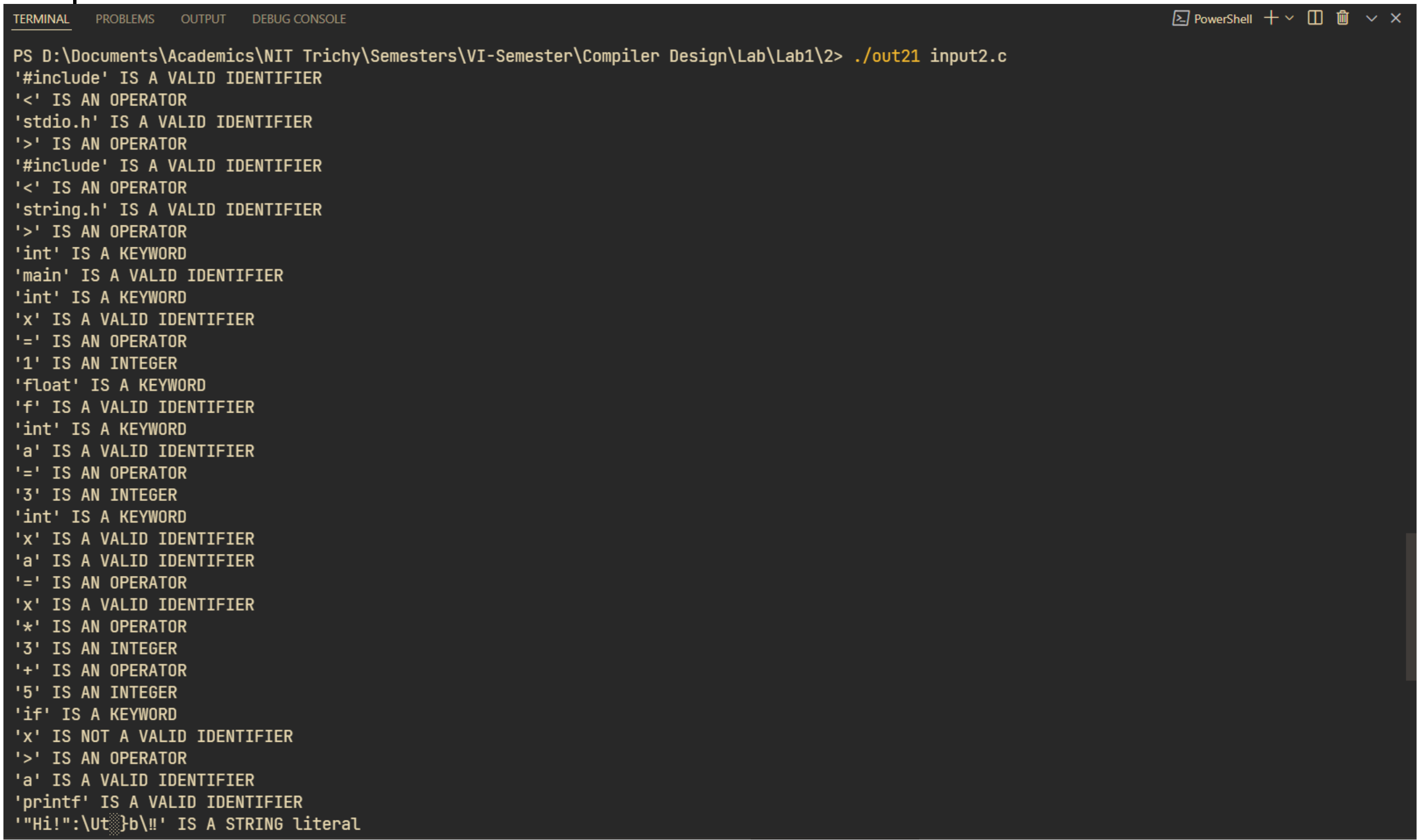
```
        }
    }
}

// DRIVER FUNCTION
int main(int argc, char** argv)
{
  char fle[100];
  strcpy(fle,argv[1]);
  char str[10000];
  FILE *fp;
  fp =fopen(fle,"r");
  while (fgets(str,1000,fp)!=NULL){
    parse(str);
  }
  return 0;
}
```

Input:

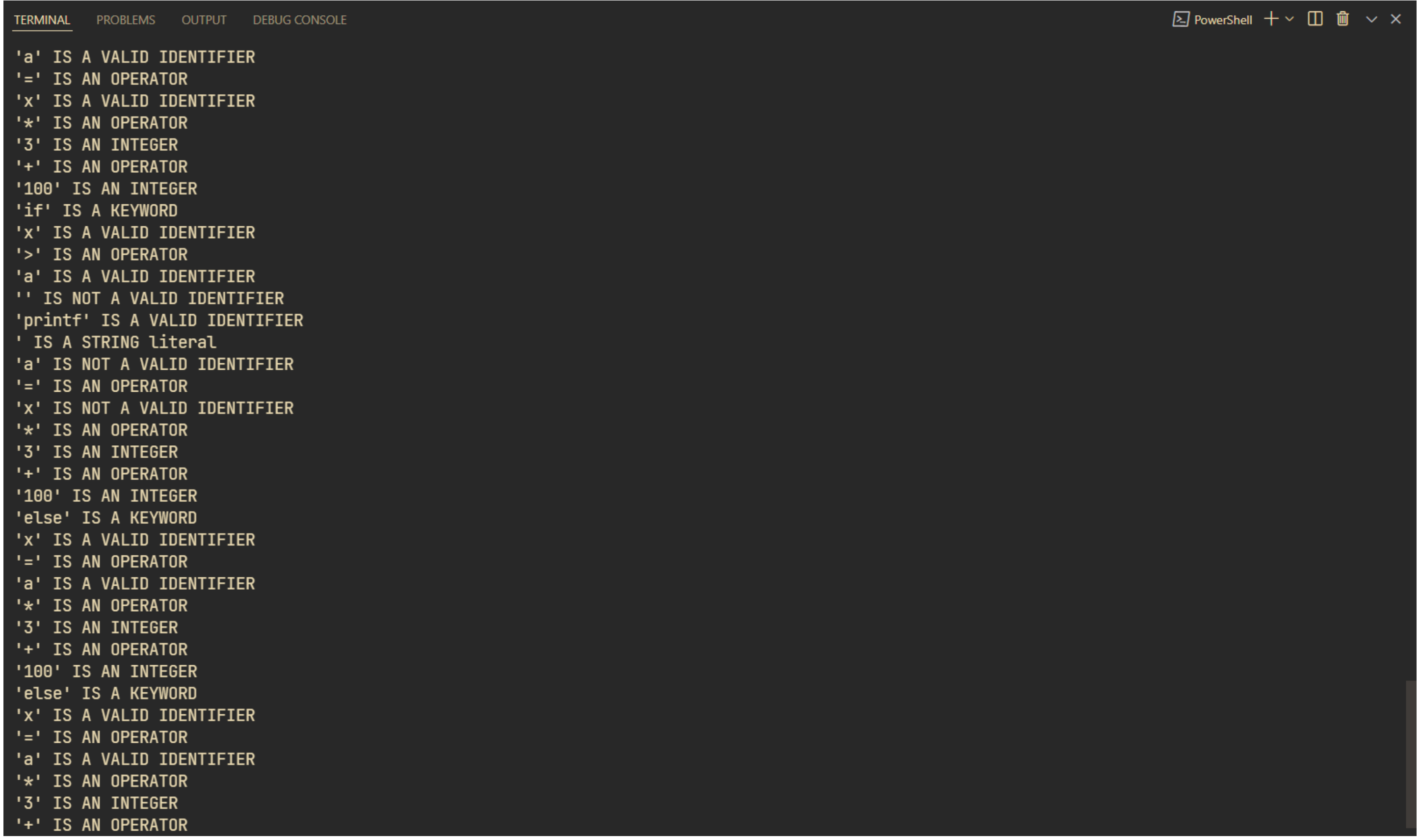


```
#include <stdio.h>
#include <string.h>

int main()
{
    int x = 1;
    float f;
    int a = 3;
    int x;
    a = x * 3 + 5;
    if (x > a)
    {
        printf("Hi!");
        a = x * 3 + 100;
        if (x > a)
        {
            printf("Hi!");
            a = x * 3 + 100;
        }
        else
        {
            x = a * 3 + 100;
        }
    }
    else
    {
        x = a * 3 + 98;
    }
}
```

Output1:



```
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\2> ./out21 input2.c
'#include' IS A VALID IDENTIFIER
'<' IS AN OPERATOR
'stdio.h' IS A VALID IDENTIFIER
'>' IS AN OPERATOR
'#include' IS A VALID IDENTIFIER
'<' IS AN OPERATOR
'string.h' IS A VALID IDENTIFIER
'>' IS AN OPERATOR
'int' IS A KEYWORD
'main' IS A VALID IDENTIFIER
'int' IS A KEYWORD
'x' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'1' IS AN INTEGER
'float' IS A KEYWORD
'f' IS A VALID IDENTIFIER
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'3' IS AN INTEGER
'int' IS A KEYWORD
'x' IS A VALID IDENTIFIER
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'x' IS A VALID IDENTIFIER
'*' IS AN OPERATOR
'3' IS AN INTEGER
'+' IS AN OPERATOR
'5' IS AN INTEGER
'if' IS A KEYWORD
'x' IS NOT A VALID IDENTIFIER
'>' IS AN OPERATOR
'a' IS A VALID IDENTIFIER
'printf' IS A VALID IDENTIFIER
'"Hi!":\Ut░}b\‼' IS A STRING literal
```



```
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'x' IS A VALID IDENTIFIER
'*' IS AN OPERATOR
'3' IS AN INTEGER
'+' IS AN OPERATOR
'100' IS AN INTEGER
'if' IS A KEYWORD
'x' IS A VALID IDENTIFIER
'>' IS AN OPERATOR
'a' IS A VALID IDENTIFIER
'' IS NOT A VALID IDENTIFIER
'printf' IS A VALID IDENTIFIER
' IS A STRING literal
'a' IS NOT A VALID IDENTIFIER
'=' IS AN OPERATOR
'x' IS NOT A VALID IDENTIFIER
'*' IS AN OPERATOR
'3' IS AN INTEGER
'+' IS AN OPERATOR
'100' IS AN INTEGER
'else' IS A KEYWORD
'x' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'a' IS A VALID IDENTIFIER
'*' IS AN OPERATOR
'3' IS AN INTEGER
'+' IS AN OPERATOR
'100' IS AN INTEGER
'else' IS A KEYWORD
'x' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'a' IS A VALID IDENTIFIER
'*' IS AN OPERATOR
'3' IS AN INTEGER
'+' IS AN OPERATOR
```

Code1.y

```
%{
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>
    #include<ctype.h>
    #include"lex.yy.c"

    void yyerror(const char *s);
    int yylex();
    int yywrap();
    void add(char);
    void insert_type();
    int search(char *);
    void insert_type();

    struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        int line_no;
    } symbol_table[40];

    int count=0;
    int q;
    char type[10];
    extern int countn;
%}

%token VOID CHARACTER PRINTFF SCANFF INT FLOAT CHAR FOR IF ELSE
TRUE FALSE NUMBER FLOAT_NUM ID LE GE EQ NE GT LT AND OR STR ADD
MULTIPLY DIVIDE SUBTRACT UNARY INCLUDE RETURN

%%
```

```
program: headers main '(' ')' '{' body return '}'
;

headers: headers headers
| INCLUDE { add('H'); }
;

main: datatype ID { add('F'); }
;

datatype: INT { insert_type(); }
| FLOAT { insert_type(); }
| CHAR { insert_type(); }
| VOID { insert_type(); }
;

body: FOR { add('K'); } '(' statement ';' condition ';' statement ')' '{' body '}'
| IF { add('K'); } '(' condition ')' '{' body '}' else
| statement ';'
| body body
| PRINTFF { add('K'); } '(' STR ')' ';'
| SCANFF { add('K'); } '(' STR ',' '&' ID ')' ';'
;

else: ELSE { add('K'); } '{' body '}'
|
;

condition: value relop value
| TRUE { add('K'); }
| FALSE { add('K'); }
|
;

statement: datatype ID { add('V'); } init
| ID '=' expression
| ID relop expression
| ID UNARY
| UNARY ID
;

init: '=' value
|
;

expression: expression arithmetic expression
| value
;

arithmetic: ADD
| SUBTRACT
| MULTIPLY
| DIVIDE
;

relop: LT
| GT
| LE
| GE
| EQ
| NE
;

value: NUMBER { add('C'); }
| FLOAT_NUM { add('C'); }
| CHARACTER { add('C'); }
| ID
;

return: RETURN { add('K'); } value ';'
|
;
```

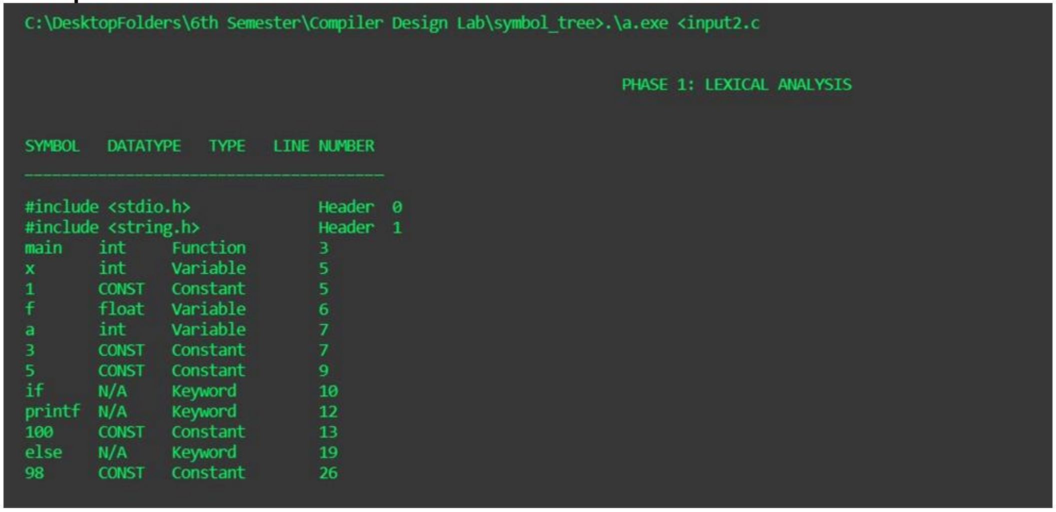
```
%%

int main() {
  yyparse();
  printf("\n\n");
    printf("\t\t\t\t\t\t\t\t PHASE 1: LEXICAL ANALYSIS \n\n");
    printf("\nSYMBOL   DATATYPE   TYPE   LINE NUMBER \n");
    printf("_______________________________________\n\n");
    int i=0;
    for(i=0; i<count; i++) {
        printf("%s\t%s\t%s\t%d\t\n", symbol_table[i].id_name, 
symbol_table[i].data_type, symbol_table[i].type, 
symbol_table[i].line_no);
    }
    for(i=0;i<count;i++) {
        free(symbol_table[i].id_name);
        free(symbol_table[i].type);
    }
    printf("\n\n");
}

int search(char *type) {
    int i;
    for(i=count-1; i>=0; i--) {
        if(strcmp(symbol_table[i].id_name, type)==0) {
            return -1;
            break;
        }
    }
    return 0;
}

void add(char c) {
  q=search(yytext);
  if(!q) {
    if(c == 'H') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup(type);
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Header");
            count++;
        }
        else if(c == 'K') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup("N/A");
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Keyword\t");
            count++;
        }
        else if(c == 'V') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup(type);
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Variable");
            count++;
        }
        else if(c == 'C') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup("CONST");
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Constant");
            count++;
        }
        else if(c == 'F') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup(type);
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Function");
            count++;
        }
    }
}

void insert_type() {
    strcpy(type, yytext);
}

void yyerror(const char* msg) {
  fprintf(stderr, "%s\n", msg);
}
```

Code2.l

```
%{
    #include "code2.tab.h"
    int countn=0;
%}

%option yylineno

alpha [a-zA-Z]
digit [0-9]
unary "++"|"--"

%%

"printf"                     { return PRINTFF; }
"scanf"                      { return SCANFF; }
"int"                        { return INT; }
"float"                      { return FLOAT; }
"char"                       { return CHAR; }
"void"                       { return VOID; }
"return"                     { return RETURN; }
"for"                        { return FOR; }
"if"                         { return IF; }
"else"                       { return ELSE; }
^"#include"[ ]*<.+\.h>       { return INCLUDE; }
"true"                       { return TRUE; }
"false"                      { return FALSE; }
[-]?{digit}+                 { return NUMBER; }
[-]?{digit}+\.{digit}{1,6}  { return FLOAT_NUM; }
{alpha}({alpha}|{digit})*    { return ID; }
{unary}                      { return UNARY; }
```

```
"<="                            { return LE; }
">="                            { return GE; }
"=="                            { return EQ; }
"!="                            { return NE; }
">"                         { return GT; }
"<"                         { return LT; }
"&&"                            { return AND; }
"||"                            { return OR; }
"+"                             { return ADD; }
"-"                             { return SUBTRACT; }
"/"                             { return DIVIDE; }
"*"                             { return MULTIPLY; }
\/\/.*                          { ; }
\/\*(.*\n)*.*\*\/               { ; }
[ \t]*                          { ; }
[\n]                            { countn++; }
.                           { return *yytext; }
["].*["]                        { return STR; }
['].[']                         { return CHARACTER; }

%%

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

Output21:

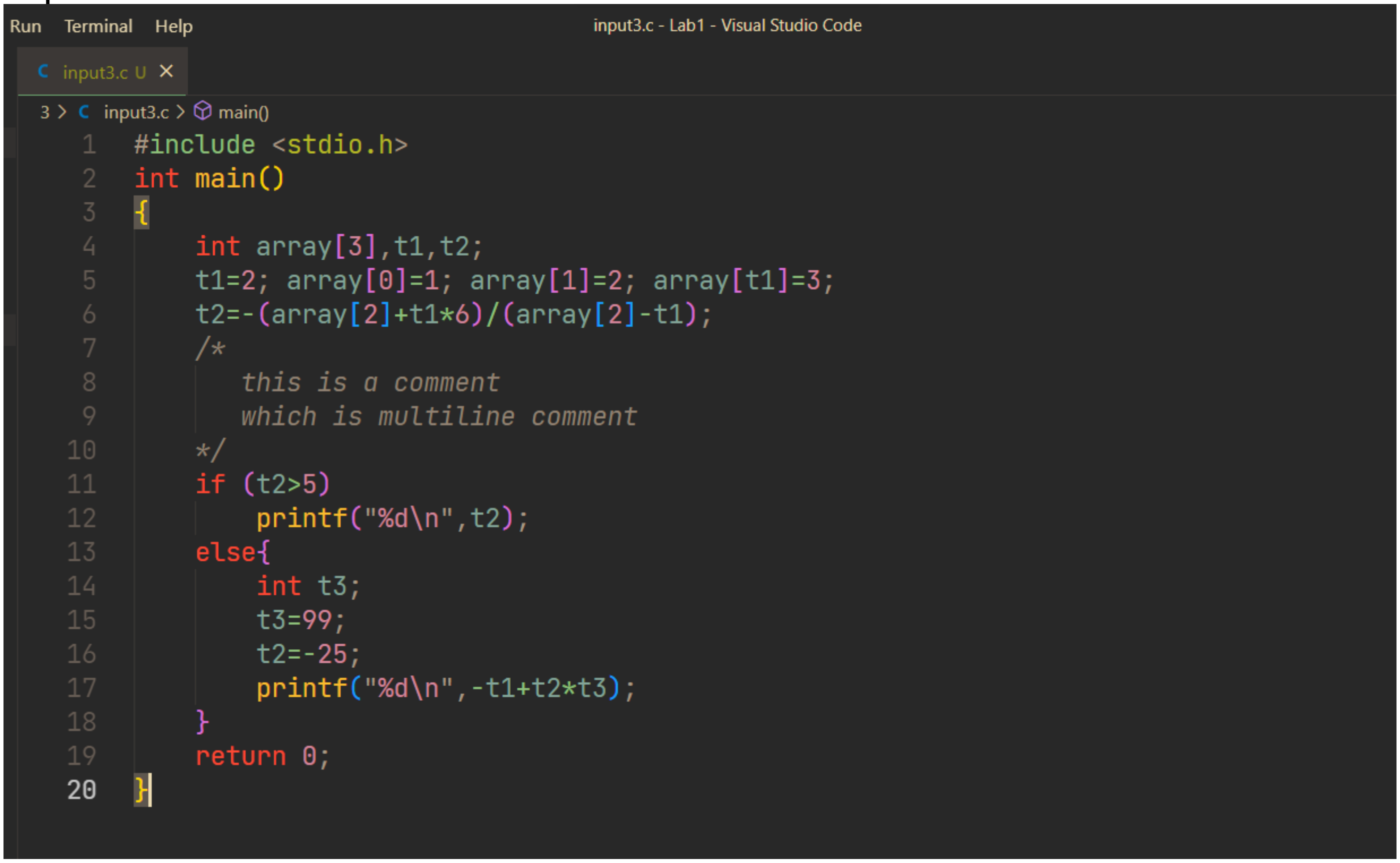


```
C:\DesktopFolders\6th Semester\Compiler Design Lab\symbol_tree>.\a.exe <input2.c


                                                PHASE 1: LEXICAL ANALYSIS


SYMBOL   DATATYPE   TYPE   LINE NUMBER
_______________________________________

#include <stdio.h>              Header  0
#include <string.h>             Header  1
main     int      Function      3
x        int      Variable      5
1        CONST    Constant      5
f        float    Variable      6
a        int      Variable      7
3        CONST    Constant      7
5        CONST    Constant      9
if       N/A      Keyword       10
printf   N/A      Keyword       12
100      CONST    Constant      13
else     N/A      Keyword       19
98       CONST    Constant      26
```

3. Design a lexical analyzer that could recognize the operators in C language. Display the name of the operation along with the recognized operator symbol.

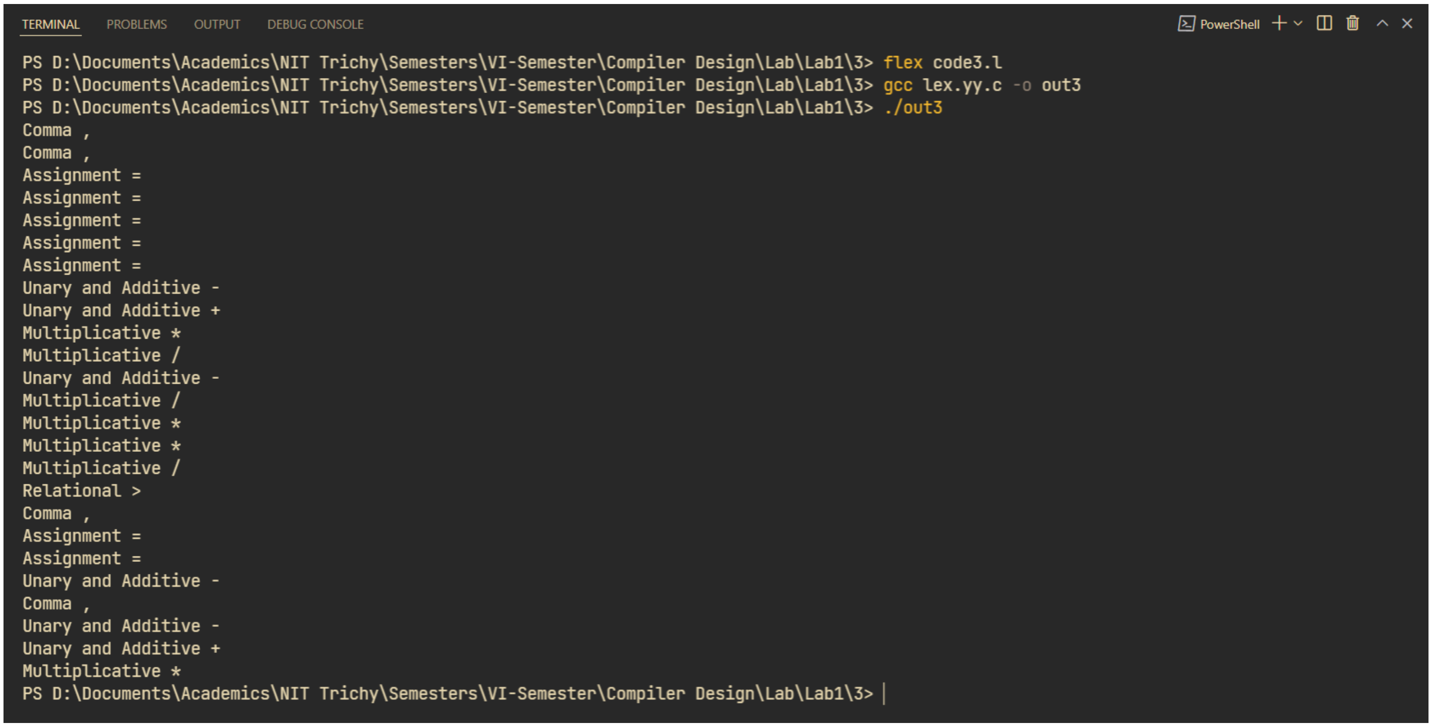
```
%{
#include<stdio.h>
%}


%%

\#.*\n ;
"++"|"--" printf("Unary and Postfix ");ECHO;printf("\n");
"+"|"-"    printf("Unary and Additive ");ECHO;printf("\n");
"!"|"~"|"sizeof()" printf("Unary ");ECHO;printf("\n");
"*"|"/"|"%"  printf("Multiplicative ");ECHO;printf("\n");
"<<"|">>" printf("Shift ");ECHO;printf("\n");
"<"|"<="|">"|">=" printf("Relational ");ECHO;printf("\n");
"=="|"!=" printf("Equality ");ECHO;printf("\n");
"&" printf("Bitwise AND ");ECHO;printf("\n");
"|" printf("Bitwise OR ");ECHO;printf("\n");
"^" printf("Bitwise XOR ");ECHO;printf("\n");
"&&" printf("Logical AND ");ECHO;printf("\n");
"||" printf("Logical OR ");ECHO;printf("\n");
"?:" printf("Conditional ");ECHO;printf("\n");
"="|"+="|"-="|"*="|"/="|"%="|">>="|"<<="|"&="|"^="|"|=" printf("Assignment ");ECHO;printf("\n");
"," printf("Comma ");ECHO;printf("\n");
\".*\" ;
\n ;
. ;

%%

int yywrap(){return 1;}

int main()
{
   yyin = fopen("input.c", "r");
   yylex();
return 0;
}
```

Input:



```c
#include <stdio.h>
int main()
{
    int array[3],t1,t2;
    t1=2; array[0]=1; array[1]=2; array[t1]=3;
    t2=-(array[2]+t1*6)/(array[2]-t1);
    /*
       this is a comment
       which is multiline comment
    */
    if (t2>5)
        printf("%d\n",t2);
    else{
        int t3;
        t3=99;
        t2=-25;
        printf("%d\n",-t1+t2*t3);
    }
    return 0;
}
```

Output:



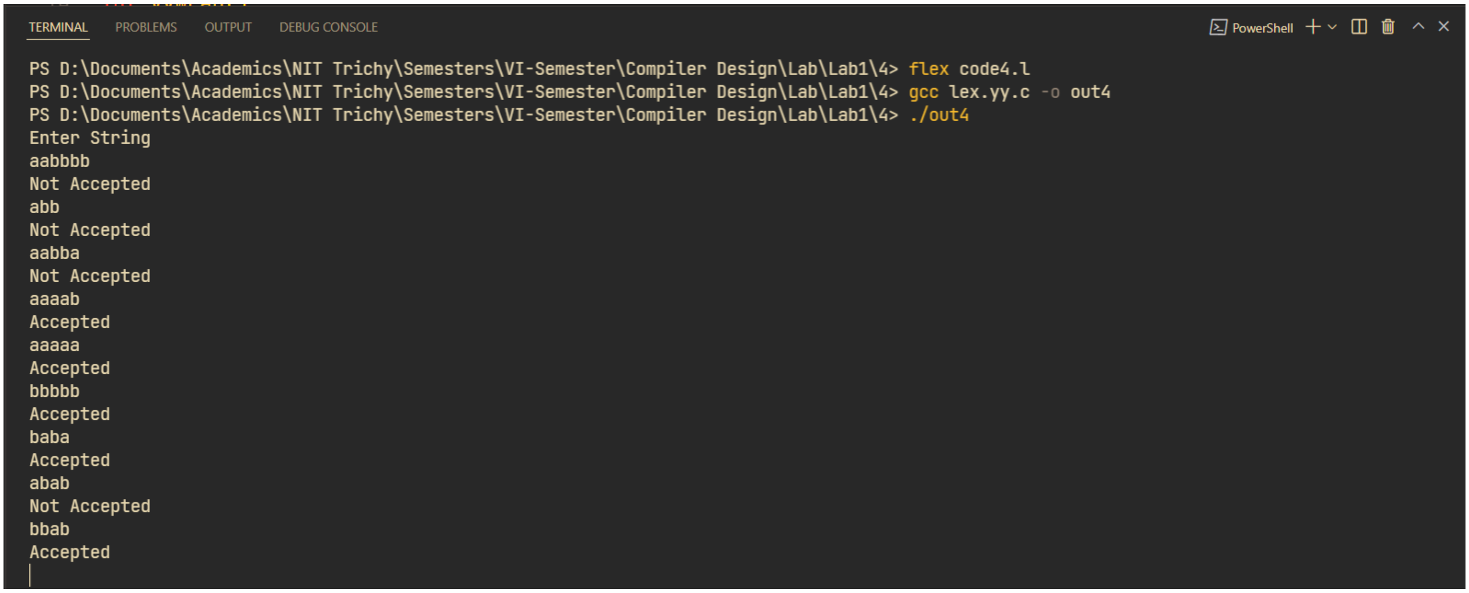
```
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\3> flex code3.l
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\3> gcc lex.yy.c -o out3
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\3> ./out3
Comma ,
Comma ,
Assignment =
Assignment =
Assignment =
Assignment =
Assignment =
Unary and Additive -
Unary and Additive +
Multiplicative *
Multiplicative /
Unary and Additive -
Multiplicative /
Multiplicative *
Multiplicative *
Multiplicative /
Relational >
Comma ,
Assignment =
Assignment =
Unary and Additive -
Comma ,
Unary and Additive -
Unary and Additive +
Multiplicative *
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\3>
```

4. Write a Lex program that accepts all strings of a's and b's that do not contain the subsequence abb.

Code:

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



%%

b*aa*ba*b(a|b)* printf("Not Accepted") ;
[\^b*aa*ba*b(a|b)*]+   printf("Accepted");

%%

int yywrap()
{
return 1;
}

int main()
{
   printf("Enter String\n");
   // called yylex
   yylex();
return 0;
}
```

Input/Output:

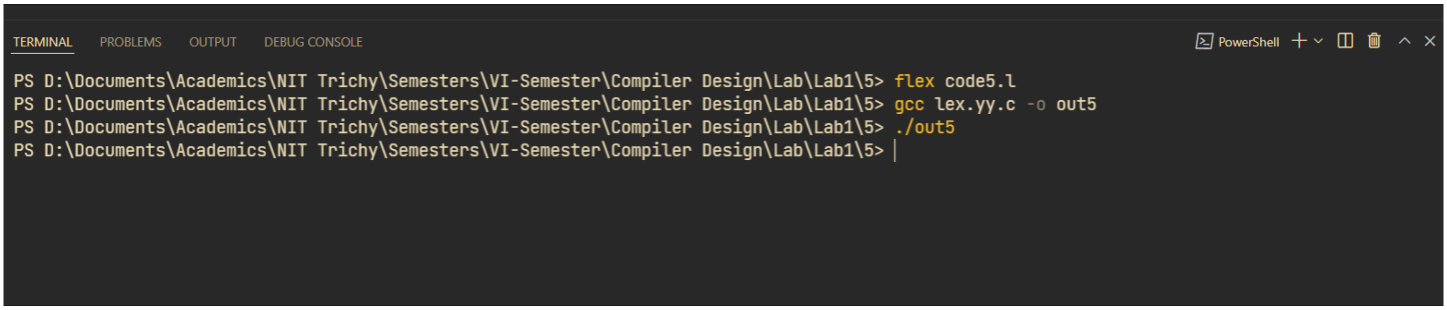


```
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\4> flex code4.l
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\4> gcc lex.yy.c -o out4
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\4> ./out4
Enter String
aabbbb
Not Accepted
abb
Not Accepted
aabba
Not Accepted
aaaab
Accepted
aaaaa
Accepted
bbbbb
Accepted
baba
Accepted
abab
Not Accepted
bbab
Accepted
```

5. Write a Lex program that copies a C program, replacing each instance of the keyword float by double.

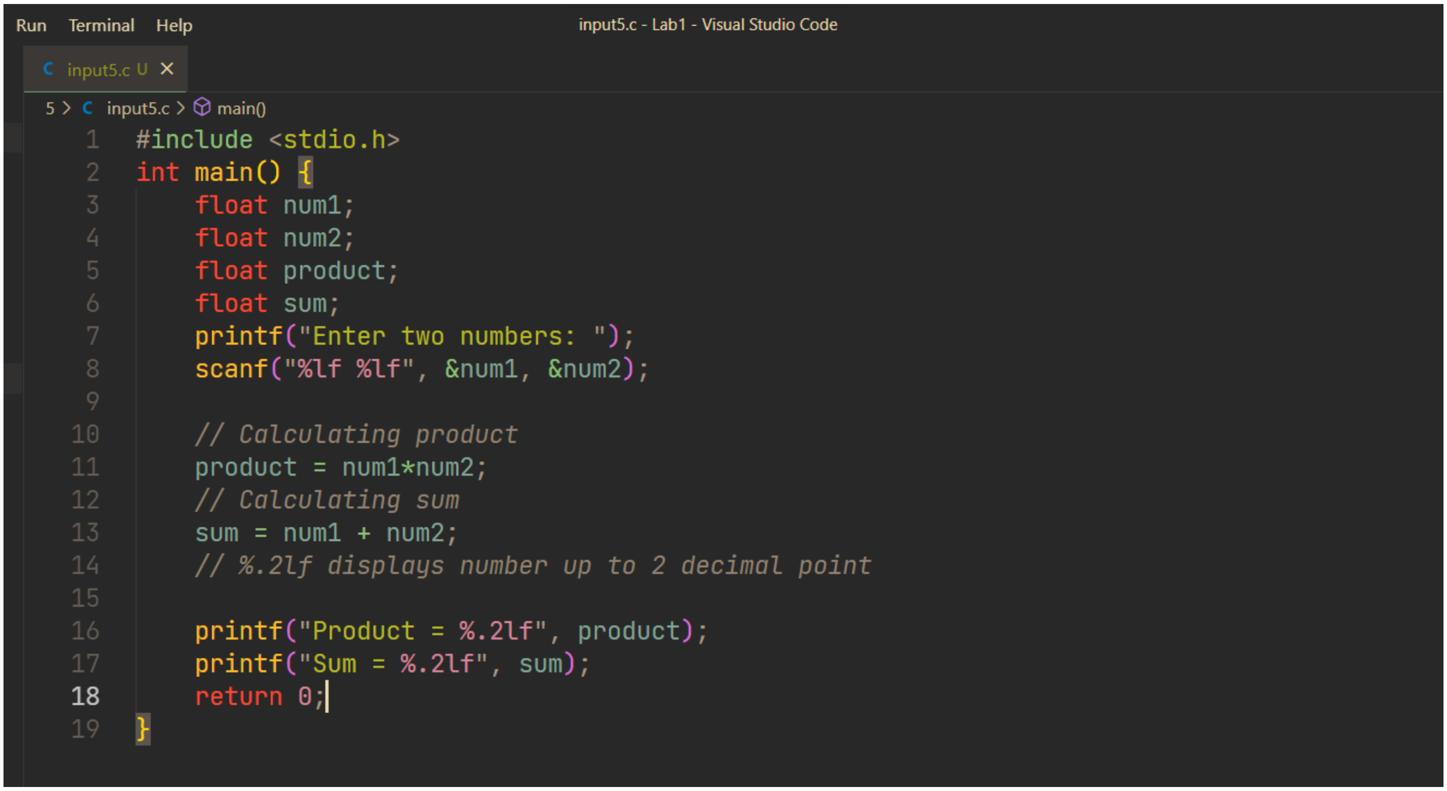
Code:

```
%{
#include<stdio.h>
//#include<string.h>
%}

%%

"float" fprintf(yyout,"double");
.|\n fprintf(yyout,"%s",yytext);

%%

int yywrap()
{
    return 1;
}


int main()
{
  yyin = fopen("input5.c", "r");
    yyout = fopen("output5.c", "w");

    yylex();
    return 0;
}
```



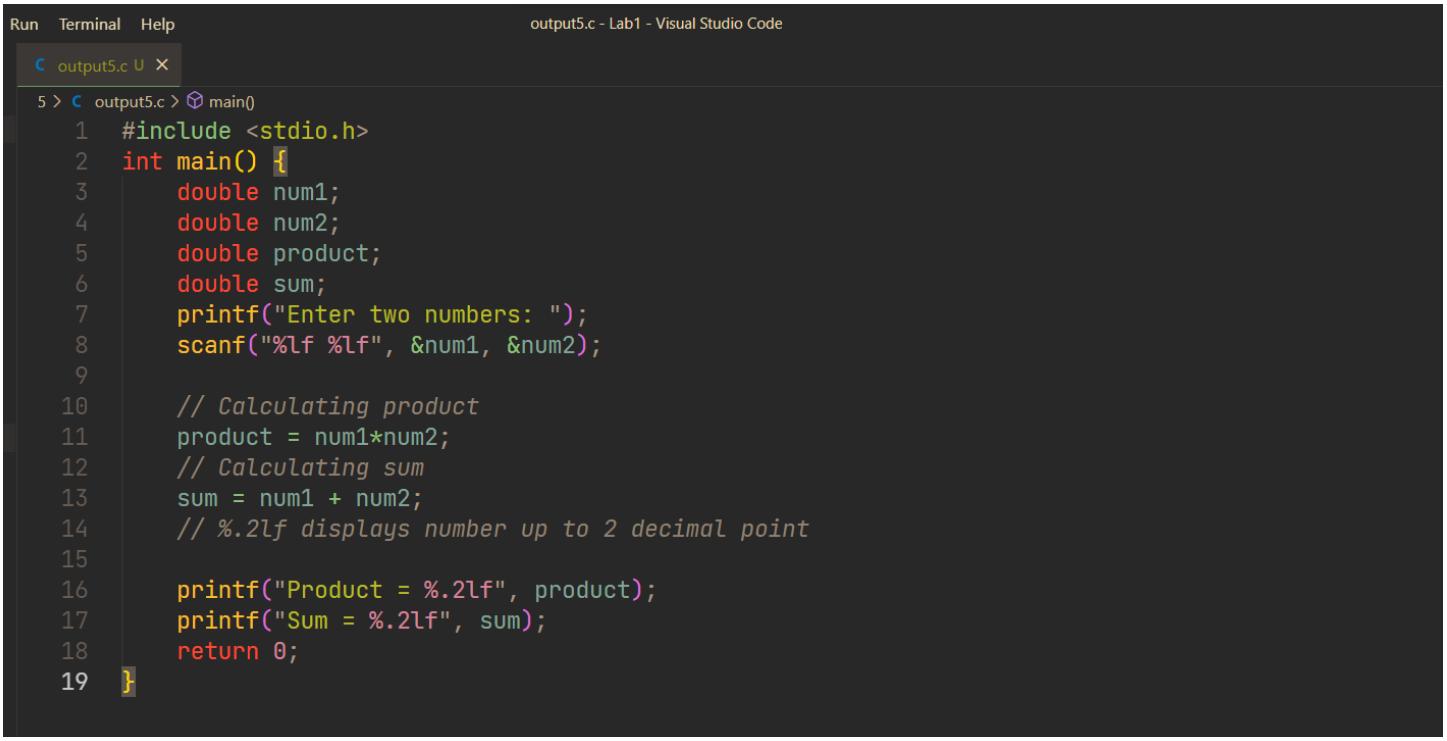
```
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\5> flex code5.l
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\5> gcc lex.yy.c -o out5
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\5> ./out5
PS D:\Documents\Academics\NIT Trichy\Semesters\VI-Semester\Compiler Design\Lab\Lab1\5>
```

Input:



```c
#include <stdio.h>
int main() {
    float num1;
    float num2;
    float product;
    float sum;
    printf("Enter two numbers: ");
    scanf("%lf %lf", &num1, &num2);

    // Calculating product
    product = num1*num2;
    // Calculating sum
    sum = num1 + num2;
    // %.2lf displays number up to 2 decimal point

    printf("Product = %.2lf", product);
    printf("Sum = %.2lf", sum);
    return 0;
}
```

Output:



```c
#include <stdio.h>
int main() {
    double num1;
    double num2;
    double product;
    double sum;
    printf("Enter two numbers: ");
    scanf("%lf %lf", &num1, &num2);

    // Calculating product
    product = num1*num2;
    // Calculating sum
    sum = num1 + num2;
    // %.2lf displays number up to 2 decimal point

    printf("Product = %.2lf", product);
    printf("Sum = %.2lf", sum);
    return 0;
}
```