Lab Manual

Of

Compiler Design Laboratory (CS606)

By

Dhruhi Shah (22000381)

Third Year, Semester 6

Course In-charge: Prof. Vaibhavi Patel

NAVRACHANA UNIVERSITY

a UGC recognized University

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring semester 2025

Sr No.	Experiment Title	Page no.
1	a. Write a program to recognize strings starts with ‘a’ over {a, b}. b. Write a program to recognize strings end with ‘a’. c. Write a program to recognize strings end with ‘ab’. Take the input from text file. d. Write a program to recognize strings contains ‘ab’. Take the input from text file.	4
2	a. Write a program to recognize the valid identifiers. b. Write a program to recognize the valid operators. c. Write a program to recognize the valid number. d. Write a program to recognize the valid comments. e. Write a program to implement Lexical Analyzer.	13
3	To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)	26
4	Implement following programs using Lex. a. Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words. b. Write a Lex program to take input from text file and count number of vowels and consonants. c. Write a Lex program to print out all numbers from the given file. d. Write a Lex program which adds line numbers to the given file and display the same into different file. e. Write a Lex program to printout all markup tags and HTML comments in file.	30
5	a. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file. b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.	38
6	Program to implement Recursive Descent Parsing in C.	44
7	a. To Study about Yet Another Compiler-Compiler(YACC).	49

	b. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / . c. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments. d. Create Yacc and Lex specification files are used to convert infix expression to postfix expression.	

Practical-1

AIM:

a. Write a program to recognize strings starts with ‘a’ over {a, b}.

b. Write a program to recognize strings end with ‘a’.

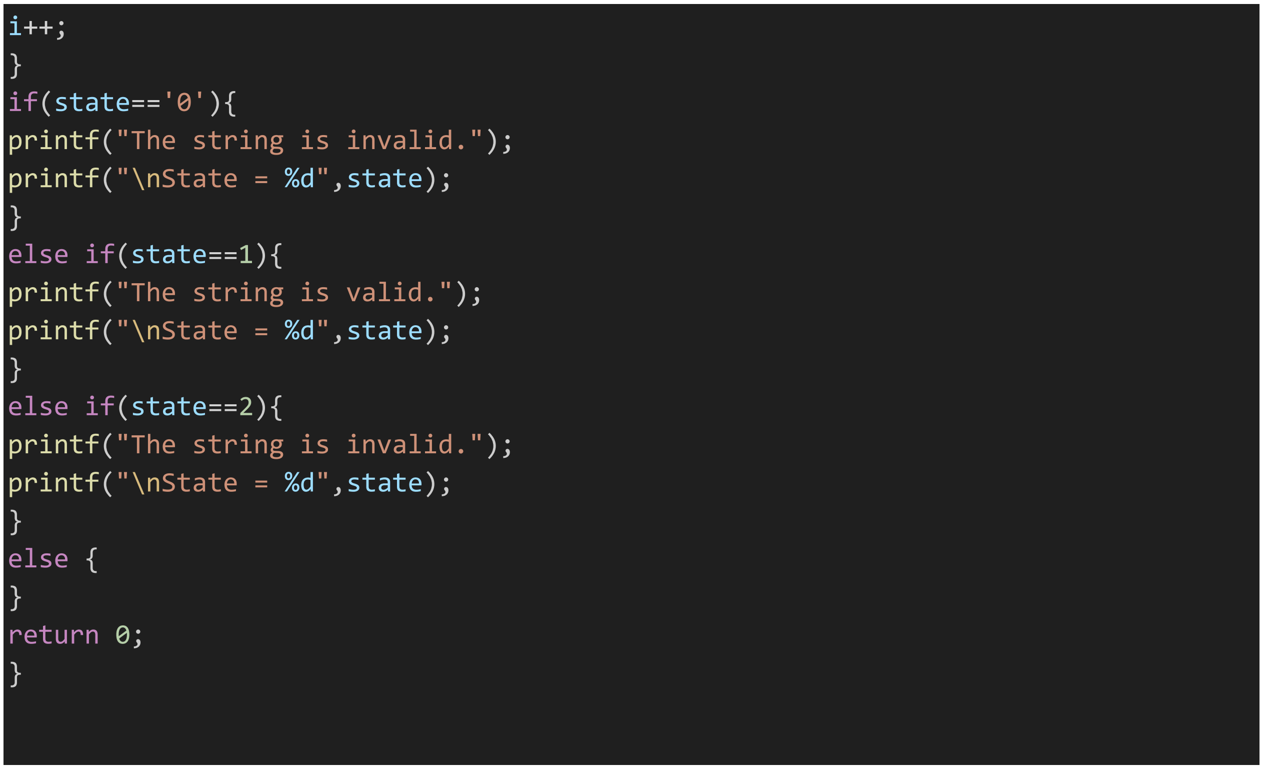
c. Write a program to recognize strings end with ‘ab’. Take the input from text file.

d. Write a program to recognize strings contains ‘ab’. Take the input from text file.

a. Write a program to recognize strings starts with ‘a’ over {a, b}.

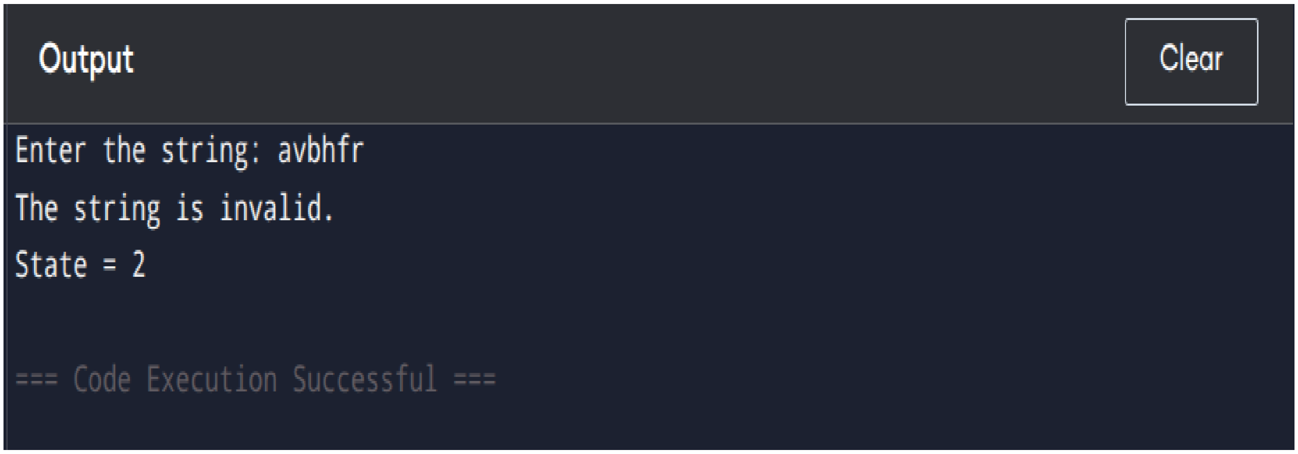
INPUT:

```
//strings starts with 'a' over {a, b}.
#include<stdio.h>
int main(){
char input[100];
int state = 0, i=0;
printf("Enter the string: ");
scanf("%s",input);
while(input[i]!='\0'){
switch(state){
case 0:
if(input[i]=='a') state = 1;
else state = 2;
break;
case 1:
if(input[i]=='a' || input[i]=='b') state=1;
else state =2;
break;
case 2:
state = 2;
break;
    }
```



```
i++;
}
if(state=='0'){
printf("The string is invalid.");
printf("\nState = %d",state);
}
else if(state==1){
printf("The string is valid.");
printf("\nState = %d",state);
}
else if(state==2){
printf("The string is invalid.");
printf("\nState = %d",state);
}
else {
}
return 0;
}
```

OUTPUT:



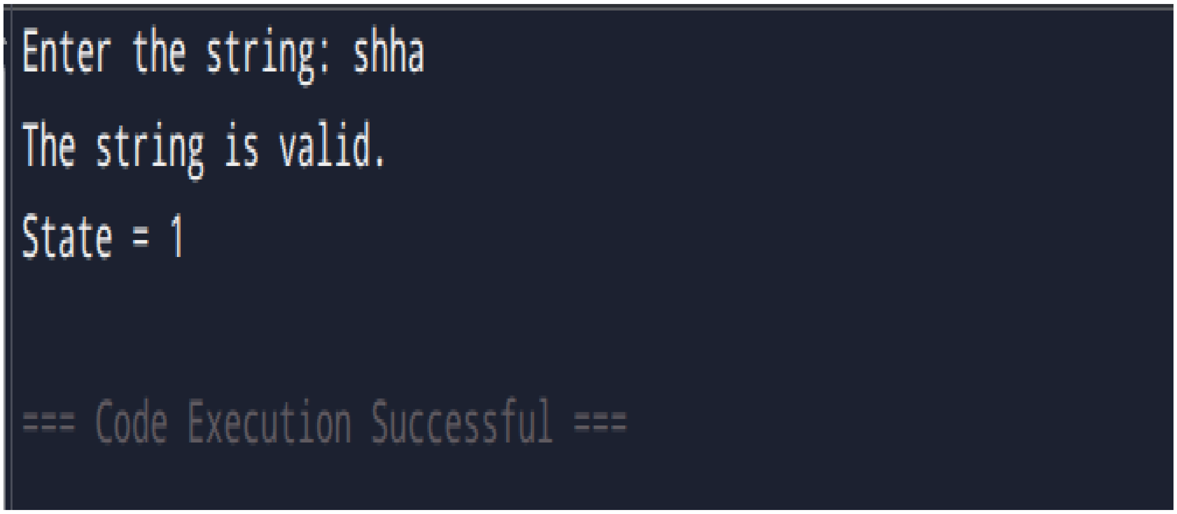
```
Enter the string: avbhfr
The string is invalid.
State = 2


=== Code Execution Successful ===
```

b. Write a program to recognize strings end with ‘a’.

INPUT:

```
//strings end with 'a'.
#include<stdio.h>
int main(){
char input[100];
int state = 0, i=0;
printf("Enter the string: ");
scanf("%s",input);

while(input[i]!='\0'){
switch(state){
case 0:
if(input[i]=='a') state = 1;
else state = 0;
break;
case 1:
if(input[i]=='a') state=1;
else state =0;
break;
}
i++;
}
if(state==0){
printf("The string is invalid.");
printf("\nState = %d",state);
}
else if(state==1){
printf("The string is valid.");
printf("\nState = %d",state);
}
else {
}
return 0;
}
```

OUTPUT:



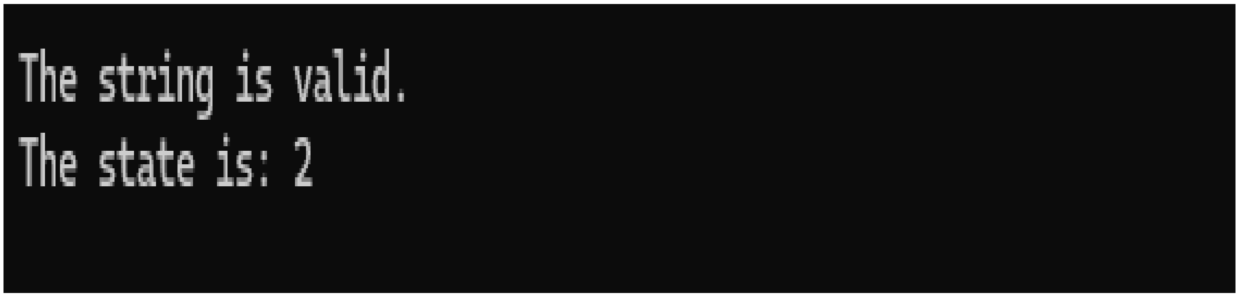
```
Enter the string: shha
The string is valid.
State = 1


=== Code Execution Successful ===
```

c. Write a program to recognize strings end with ‘ab’. Take the input from text file.

INPUT :

```c
//string ends with ab and take input from a file.
#include <stdio.h>

int main() {
    char input[100];
    int state = 0, i = 0;
    FILE *file; // File pointer

    file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Error: Could not open file.\n");
        return 1;
    }

    if (fgets(input, sizeof(input), file) == NULL) {
        printf("Error: Could not read from file or file is empty.\n");
        fclose(file);
        return 1;
    }

    fclose(file);

    // Removing newline character, if present
    for (i = 0; input[i] != '\0'; i++) {
        if (input[i] == '\n') {
            input[i] = '\0';
            break;
        }
    }

    i = 0; // Reset index for processing the string

    while (input[i] != '\0') {
        switch (state) {
            case 0:
                if (input[i] == 'a') {
                    state = 1;
                } else if (input[i] == 'b') {
                    state = 0;
                } else {
                    state = 0;
                }
                break;
            case 1:
                if (input[i] == 'b') {
                    state = 2;
                } else if (input[i] == 'a') {
                    state = 1;
                } else {
                    state = 0;
                }
                break;
            case 2:
                if (input[i] == 'a') {
                    state = 1;
                } else if (input[i] == 'b') {
                    state = 0;
                } else {
                    state = 0;
                }
                break;
        }
        i++;
    }

    if (state == 0) {
        printf("String is invalid.\n");
        printf("The state is: %d\n", state);
    } else if (state == 1) {
        printf("The string is invalid.\n");
        printf("The state is: %d\n", state);
    } else if (state == 2) {
        printf("The string is valid.\n");
        printf("The state is: %d\n", state);
    }

    return 0;
}
```

OUTPUT:

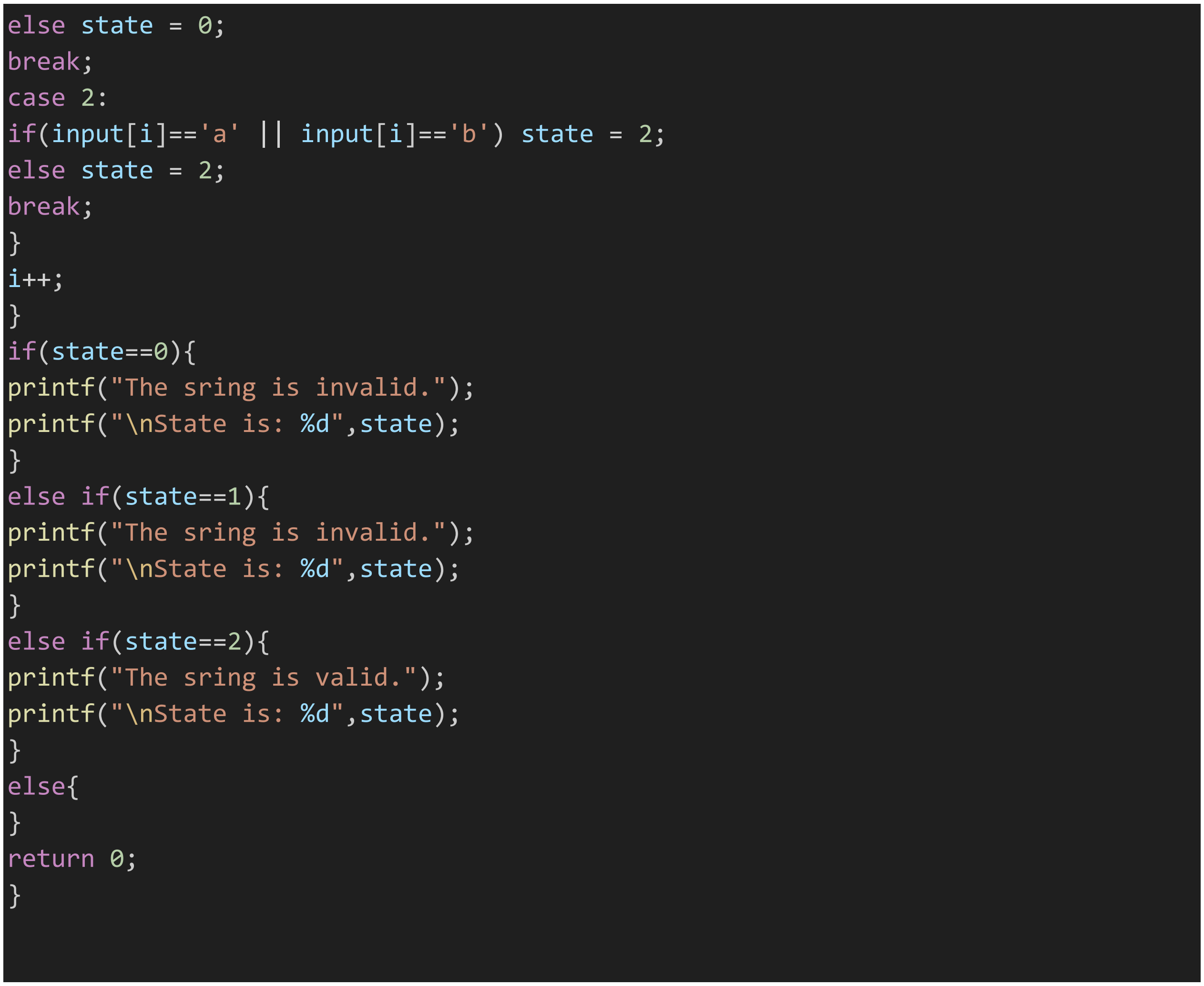


```
The string is valid.
The state is: 2
```

d. Write a program to recognize strings contains ‘ab’. Take the input from text file.

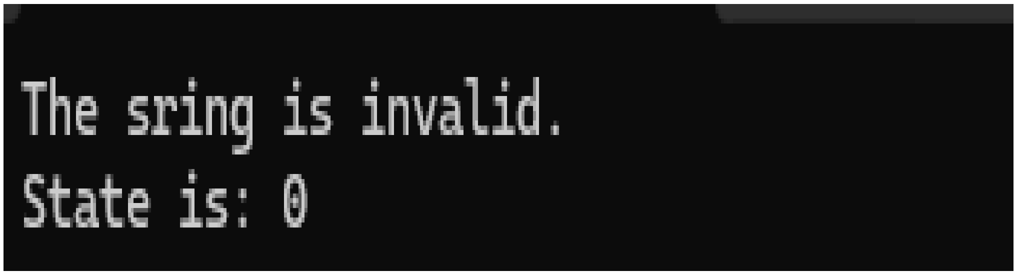
INPUT :

```
//sting contains ab, and takes input from a file.
#include<stdio.h>
int main(){
char input[100];
int state=0,i=0;
FILE *file;
file=fopen("input1.txt","r");
if(file==NULL){
printf("Error: Couldn't open the file.\n");
return 1;
}
if(fgets(input,sizeof(input),file)==NULL){
printf("Error: Could not read from file or file is empty.\n");
        fclose(file);
        return 1;
}
fclose(file);
// Removing newline character, if present
    for (i = 0; input[i] != '\0'; i++) {
        if (input[i] == '\n') {
            input[i] = '\0';
            break;
        }
    }

    i = 0; // Reset index for processing the string
/*printf("Enter the string: ");
scanf("%s",input);*/
while(input[i] != '\0'){
switch(state){
case 0:
if(input[i]=='a') state = 1;
else if(input[i]=='b') state = 0;
else state = 0;
break;
case 1:
if(input[i]=='a') state = 1;
else if(input[i]=='b') state =2;
```



```
else state = 0;
break;
case 2:
if(input[i]=='a' || input[i]=='b') state = 2;
else state = 2;
break;
}
i++;
}
if(state==0){
printf("The sring is invalid.");
printf("\nState is: %d",state);
}
else if(state==1){
printf("The sring is invalid.");
printf("\nState is: %d",state);
}
else if(state==2){
printf("The sring is valid.");
printf("\nState is: %d",state);
}
else{
}
return 0;
}
```

OUTPUT:



```
The sring is invalid.
State is: 0
```

Practical-2

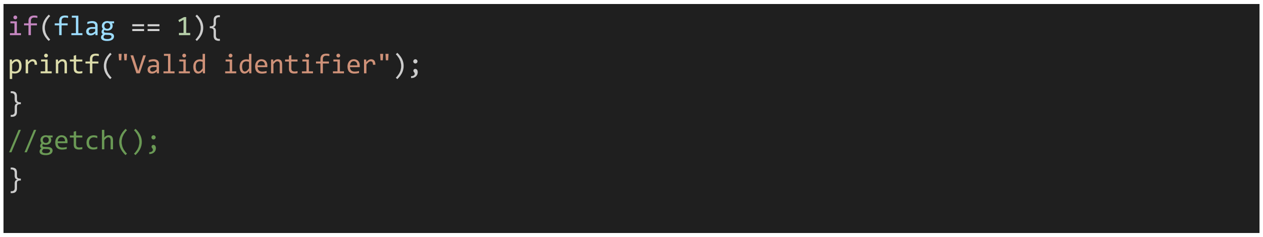
AIM:

a. Write a program to recognize the valid identifiers.
b. Write a program to recognize the valid operators.
c. Write a program to recognize the valid number.
d. Write a program to recognize the valid comments.
e. Write a program to implement Lexical Analyzer.

a. Write a program to recognize the valid identifiers.

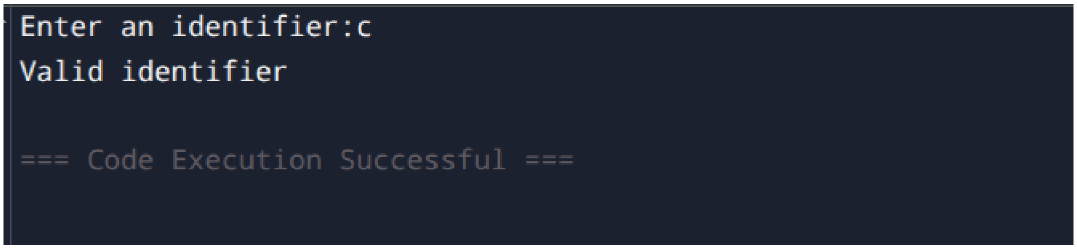
INPUT:

```
#include <stdio.h>
#include <ctype.h>
int main()
{
char a[10];
int flag, i=1;

printf("Enter an identifier:");
scanf("%s",&a);


if(isalpha(a[0])){
flag = 1; // If the first character is an alphabet, set flag = 1 (indicating a valid start).

}
else
printf("invalid identifier");
while (a[i] != '\0') {
        if (!isalnum(a[i]) && a[i] != '_') {
            flag = 0;
            break;
        }
        i++;
    }
```



```
if(flag == 1){
printf("Valid identifier");
}
//getch();
}
```

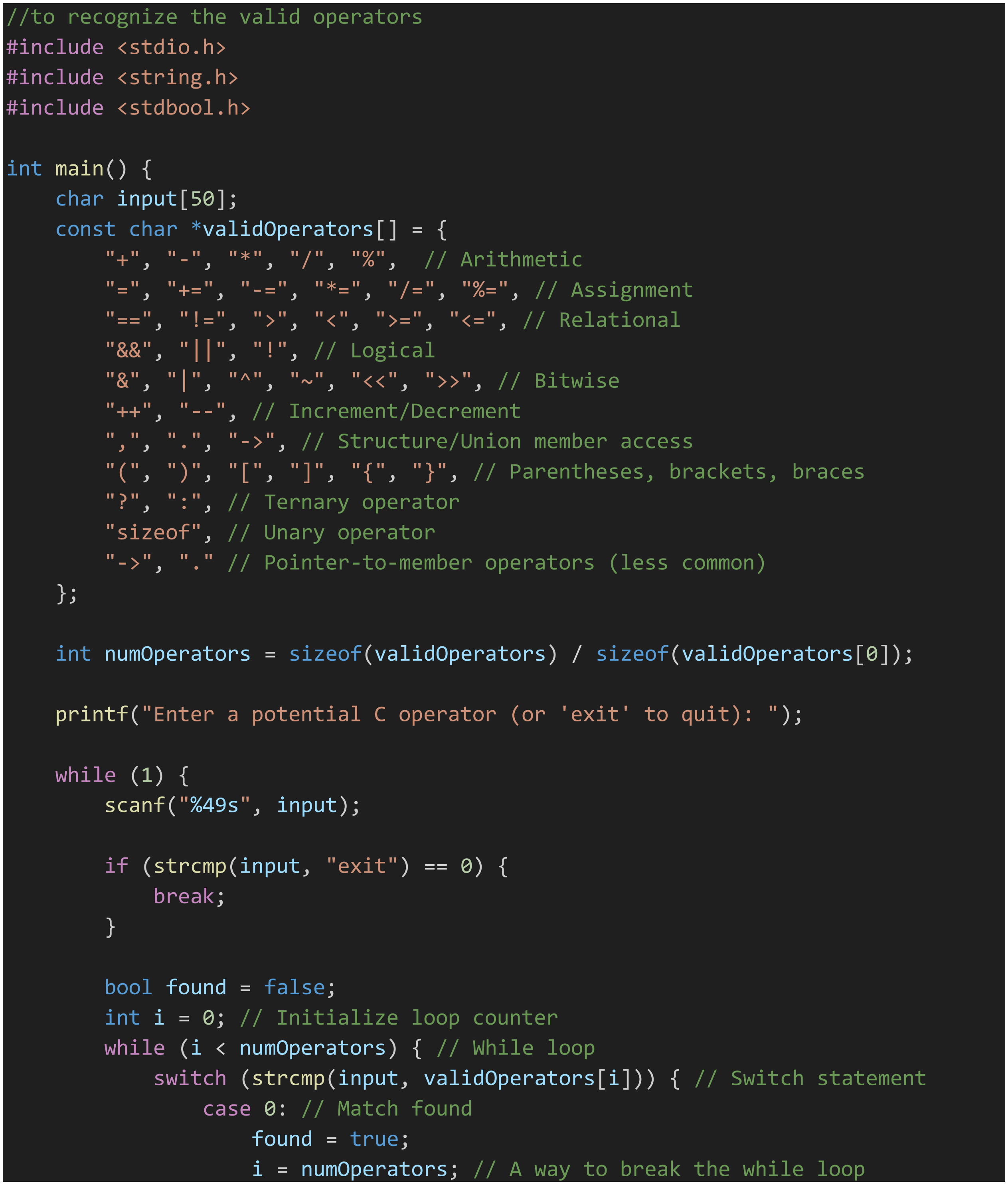
OUTPUT:



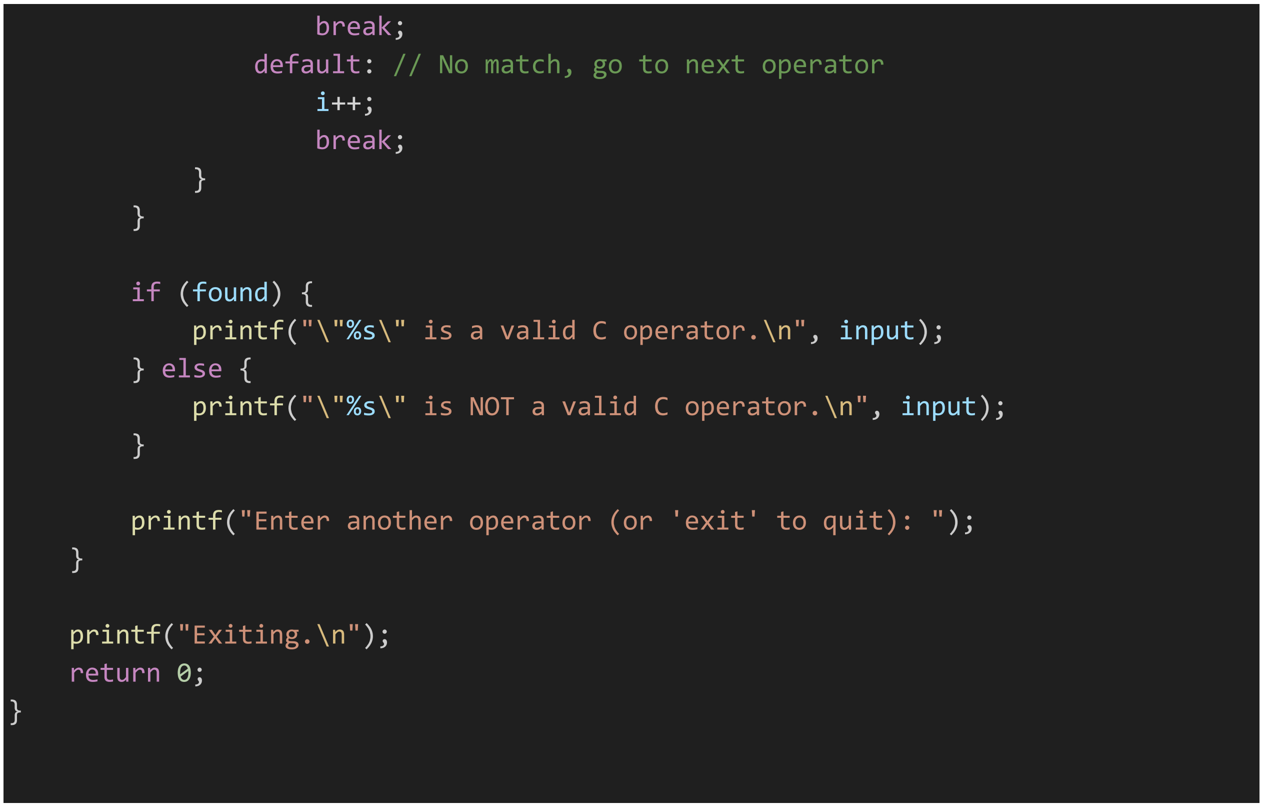
```
Enter an identifier:c
Valid identifier

=== Code Execution Successful ===
```

b. Write a program to recognize the valid operators.

INPUT:

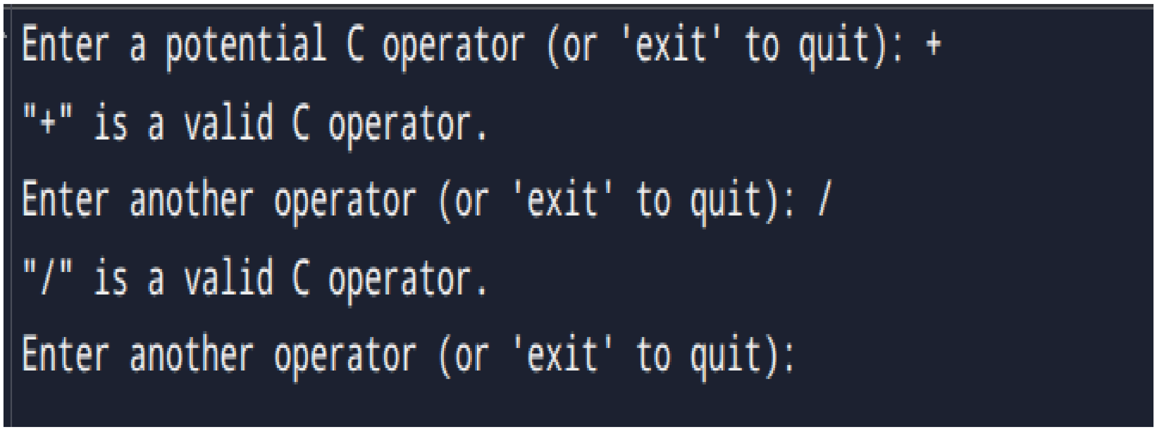


```
//to recognize the valid operators
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

int main() {
    char input[50];
    const char *validOperators[] = {
        "+", "-", "*", "/", "%",  // Arithmetic
        "=", "+=", "-=", "*=", "/=", "%=", // Assignment
        "==", "!=", ">", "<", ">=", "<=", // Relational
        "&&", "||", "!", // Logical
        "&", "|", "^", "~", "<<", ">>", // Bitwise
        "++", "--", // Increment/Decrement
        ",", ".", "->", // Structure/Union member access
        "(", ")", "[", "]", "{", "}", // Parentheses, brackets, braces
        "?", ":", // Ternary operator
        "sizeof", // Unary operator
        "->", "." // Pointer-to-member operators (less common)
    };

    int numOperators = sizeof(validOperators) / sizeof(validOperators[0]);

    printf("Enter a potential C operator (or 'exit' to quit): ");

    while (1) {
        scanf("%49s", input);

        if (strcmp(input, "exit") == 0) {
            break;
        }

        bool found = false;
        int i = 0; // Initialize loop counter
        while (i < numOperators) { // While loop
            switch (strcmp(input, validOperators[i])) { // Switch statement
                case 0: // Match found
                    found = true;
                    i = numOperators; // A way to break the while loop
```



```
                    break;
                default: // No match, go to next operator
                    i++;
                    break;
            }
        }

        if (found) {
            printf("\"%s\" is a valid C operator.\n", input);
        } else {
            printf("\"%s\" is NOT a valid C operator.\n", input);
        }

        printf("Enter another operator (or 'exit' to quit): ");
    }

    printf("Exiting.\n");
    return 0;
}
```

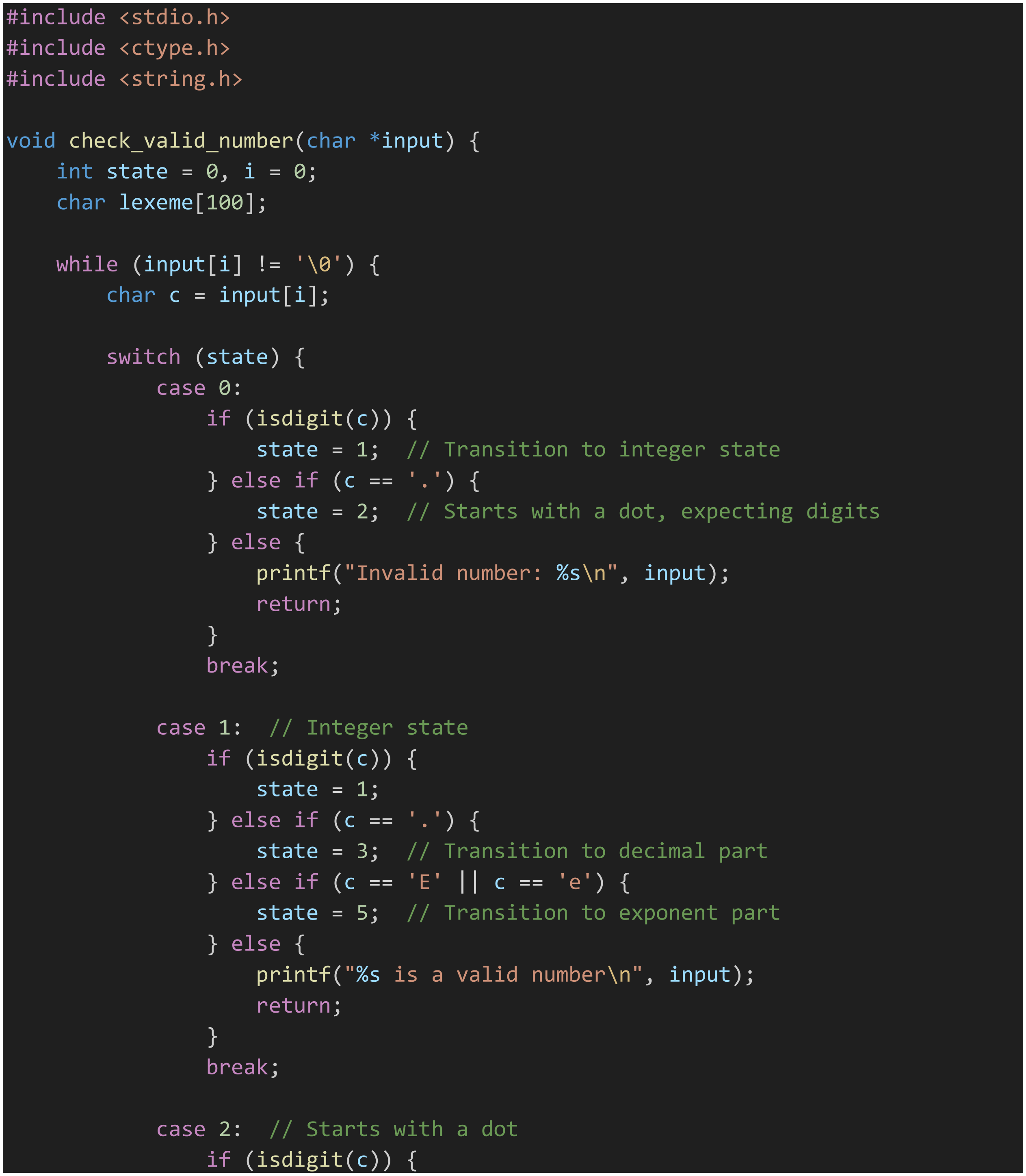
OUTPUT:



```
Enter a potential C operator (or 'exit' to quit): +
"+" is a valid C operator.
Enter another operator (or 'exit' to quit): /
"/" is a valid C operator.
Enter another operator (or 'exit' to quit):
```

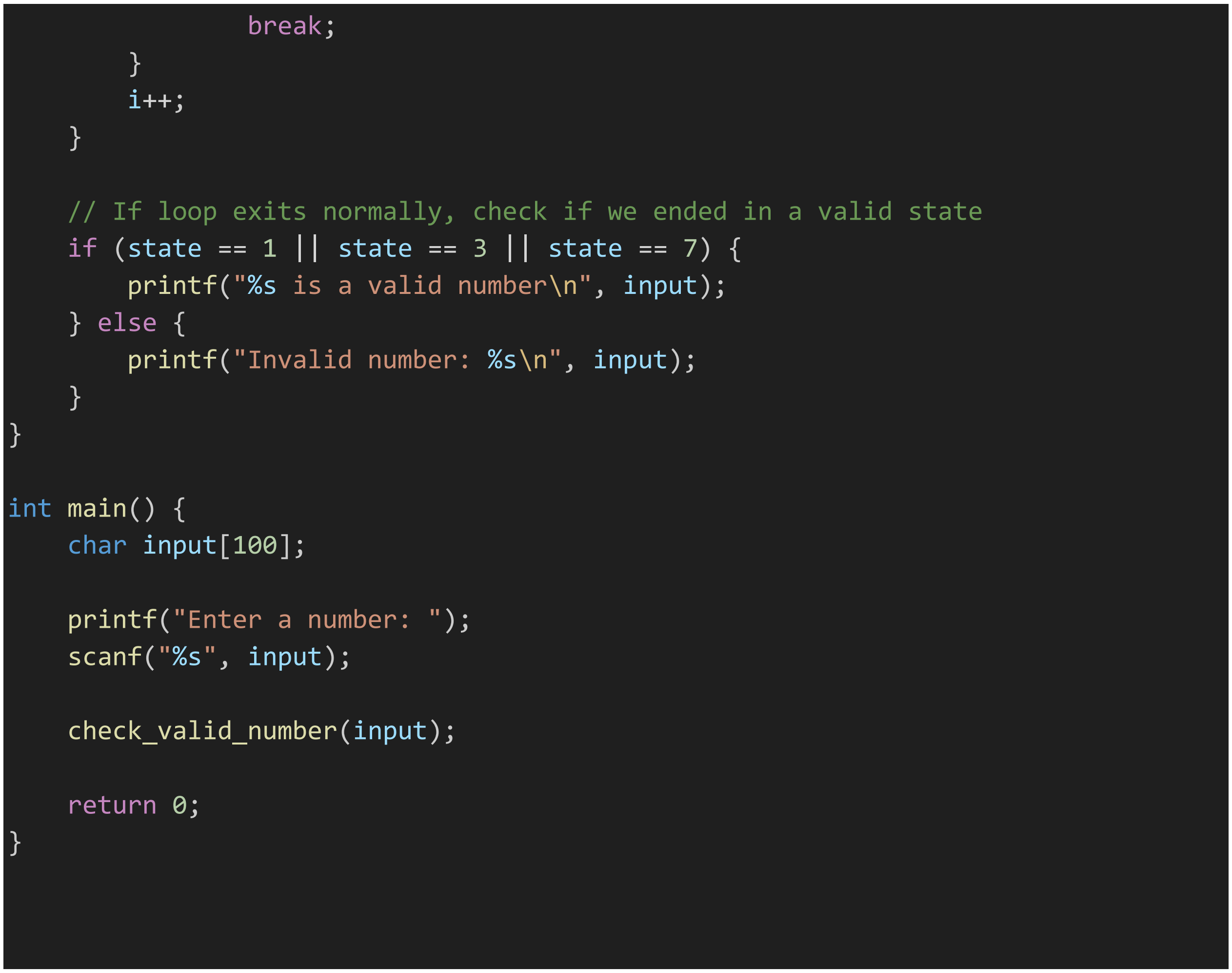
c. Write a program to recognize the valid number.

INPUT:



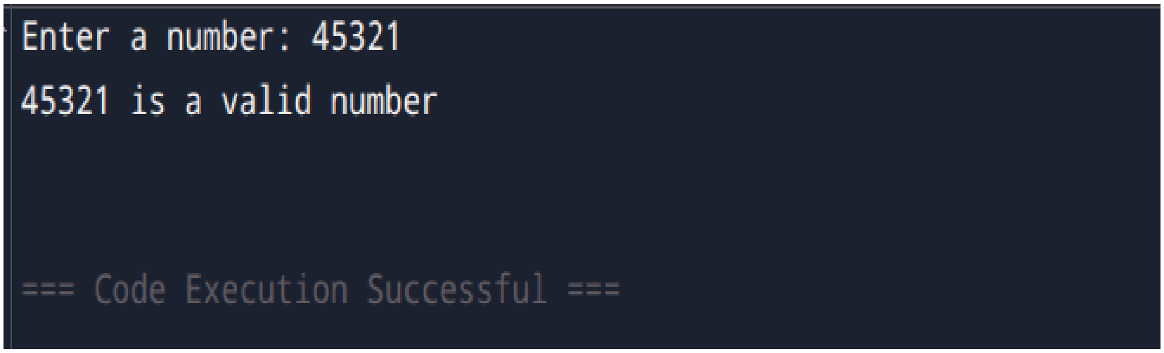
```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void check_valid_number(char *input) {
    int state = 0, i = 0;
    char lexeme[100];

    while (input[i] != '\0') {
        char c = input[i];

        switch (state) {
            case 0:
                if (isdigit(c)) {
                    state = 1;  // Transition to integer state
                } else if (c == '.') {
                    state = 2;  // Starts with a dot, expecting digits
                } else {
                    printf("Invalid number: %s\n", input);
                    return;
                }
                break;

            case 1:  // Integer state
                if (isdigit(c)) {
                    state = 1;
                } else if (c == '.') {
                    state = 3;  // Transition to decimal part
                } else if (c == 'E' || c == 'e') {
                    state = 5;  // Transition to exponent part
                } else {
                    printf("%s is a valid number\n", input);
                    return;
                }
                break;

            case 2:  // Starts with a dot
                if (isdigit(c)) {
```

```
                state = 3;
            } else {
                printf("Invalid number: %s\n", input);
                return;
            }
            break;

        case 3:  // Decimal part
            if (isdigit(c)) {
                state = 3;
            } else if (c == 'E' || c == 'e') {
                state = 5;
            } else {
                printf("%s is a valid number\n", input);
                return;
            }
            break;

        case 5:  // Exponent part
            if (c == '+' || c == '-') {
                state = 6;
            } else if (isdigit(c)) {
                state = 7;
            } else {
                printf("Invalid number: %s\n", input);
                return;
            }
            break;

        case 6:  // Sign after exponent
            if (isdigit(c)) {
                state = 7;
            } else {
                printf("Invalid number: %s\n", input);
                return;
            }
            break;

        case 7:  // Digits after exponent
            if (isdigit(c)) {
                state = 7;
            } else {
                printf("%s is a valid number\n", input);
                return;
            }
```



```
                break;
        }
        i++;
    }

    // If loop exits normally, check if we ended in a valid state
    if (state == 1 || state == 3 || state == 7) {
        printf("%s is a valid number\n", input);
    } else {
        printf("Invalid number: %s\n", input);
    }
}

int main() {
    char input[100];

    printf("Enter a number: ");
    scanf("%s", input);

    check_valid_number(input);

    return 0;
}
```

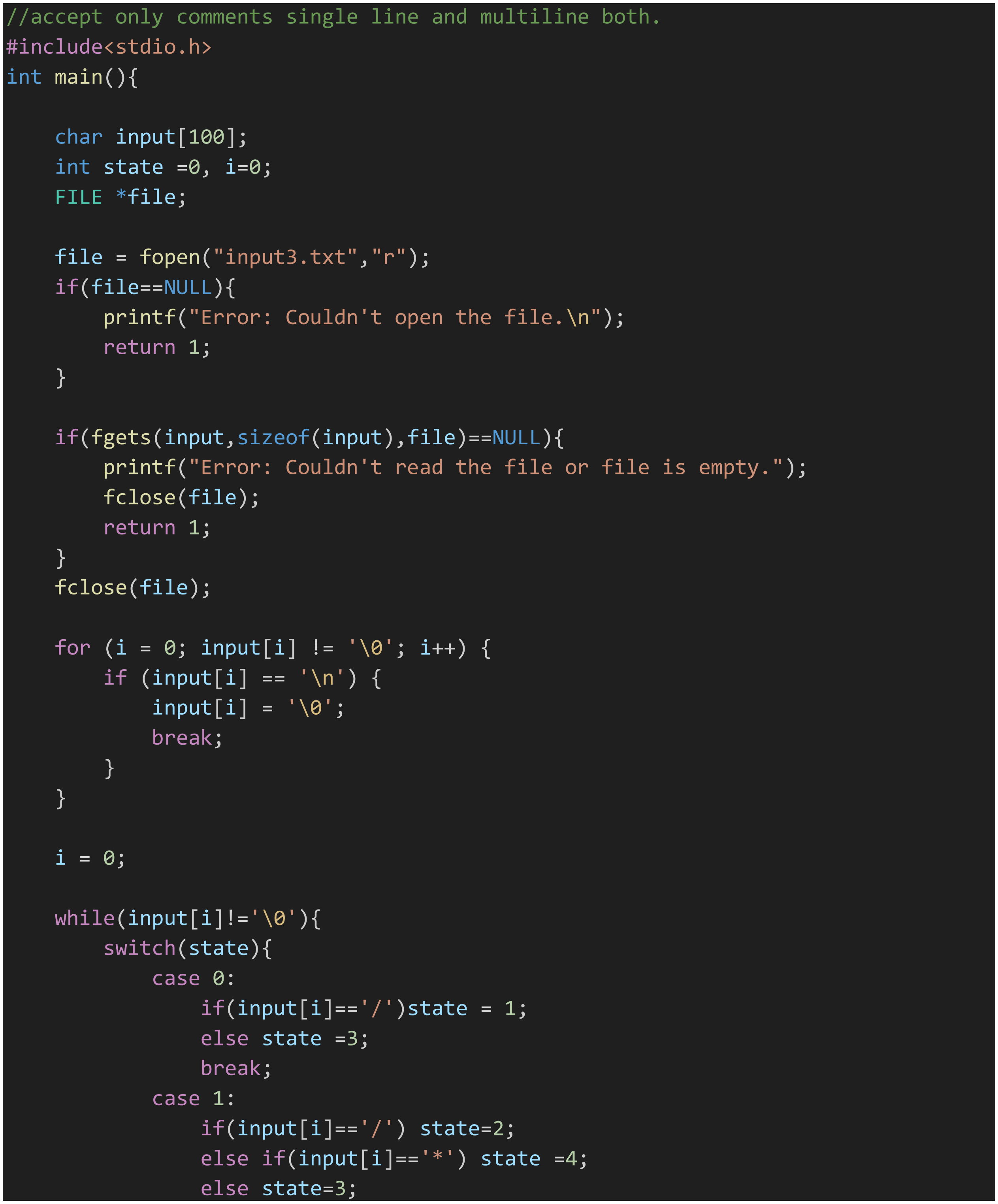
OUTPUT:



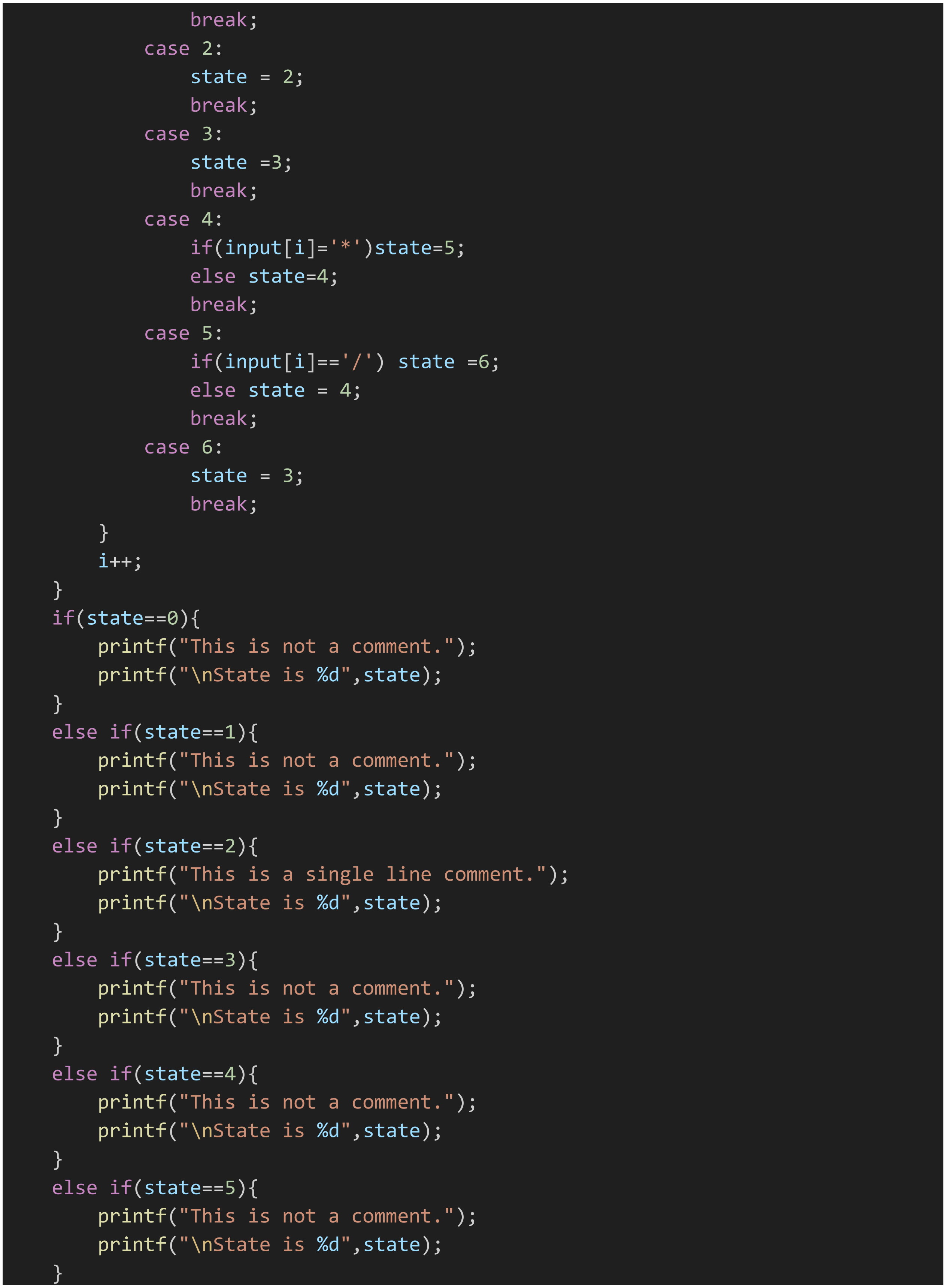
```
Enter a number: 45321
45321 is a valid number


=== Code Execution Successful ===
```

d. Write a program to recognize the valid comments.

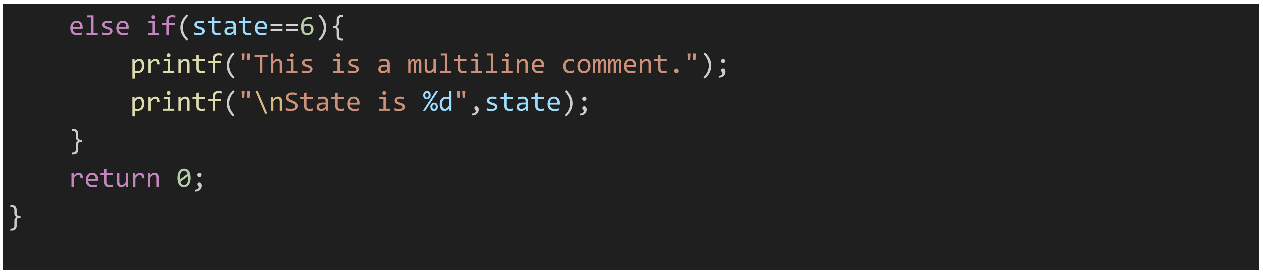
INPUT:



```
//accept only comments single line and multiline both.
#include<stdio.h>
int main(){

    char input[100];
    int state =0, i=0;
    FILE *file;

    file = fopen("input3.txt","r");
    if(file==NULL){
        printf("Error: Couldn't open the file.\n");
        return 1;
    }

    if(fgets(input,sizeof(input),file)==NULL){
        printf("Error: Couldn't read the file or file is empty.");
        fclose(file);
        return 1;
    }
    fclose(file);

    for (i = 0; input[i] != '\0'; i++) {
        if (input[i] == '\n') {
            input[i] = '\0';
            break;
        }
    }

    i = 0;

    while(input[i]!='\0'){
        switch(state){
            case 0:
                if(input[i]=='/')state = 1;
                else state =3;
                break;
            case 1:
                if(input[i]=='/') state=2;
                else if(input[i]=='*') state =4;
                else state=3;
```



```
                break;
            case 2:
                state = 2;
                break;
            case 3:
                state =3;
                break;
            case 4:
                if(input[i]='*')state=5;
                else state=4;
                break;
            case 5:
                if(input[i]=='/') state =6;
                else state = 4;
                break;
            case 6:
                state = 3;
                break;
        }
        i++;
    }
    if(state==0){
        printf("This is not a comment.");
        printf("\nState is %d",state);
    }
    else if(state==1){
        printf("This is not a comment.");
        printf("\nState is %d",state);
    }
    else if(state==2){
        printf("This is a single line comment.");
        printf("\nState is %d",state);
    }
    else if(state==3){
        printf("This is not a comment.");
        printf("\nState is %d",state);
    }
    else if(state==4){
        printf("This is not a comment.");
        printf("\nState is %d",state);
    }
    else if(state==5){
        printf("This is not a comment.");
        printf("\nState is %d",state);
    }
```

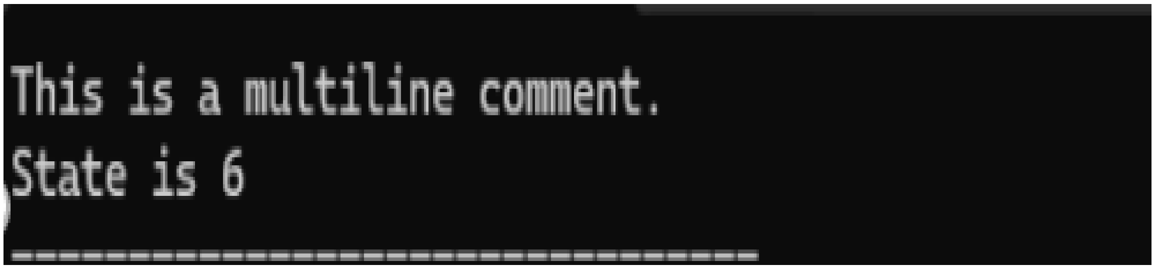


```
    else if(state==6){
        printf("This is a multiline comment.");
        printf("\nState is %d",state);
    }
    return 0;
}
```

input3.txt:

/*dsjdbhsdbf *gdgsdg *dfd */

OUTPUT:

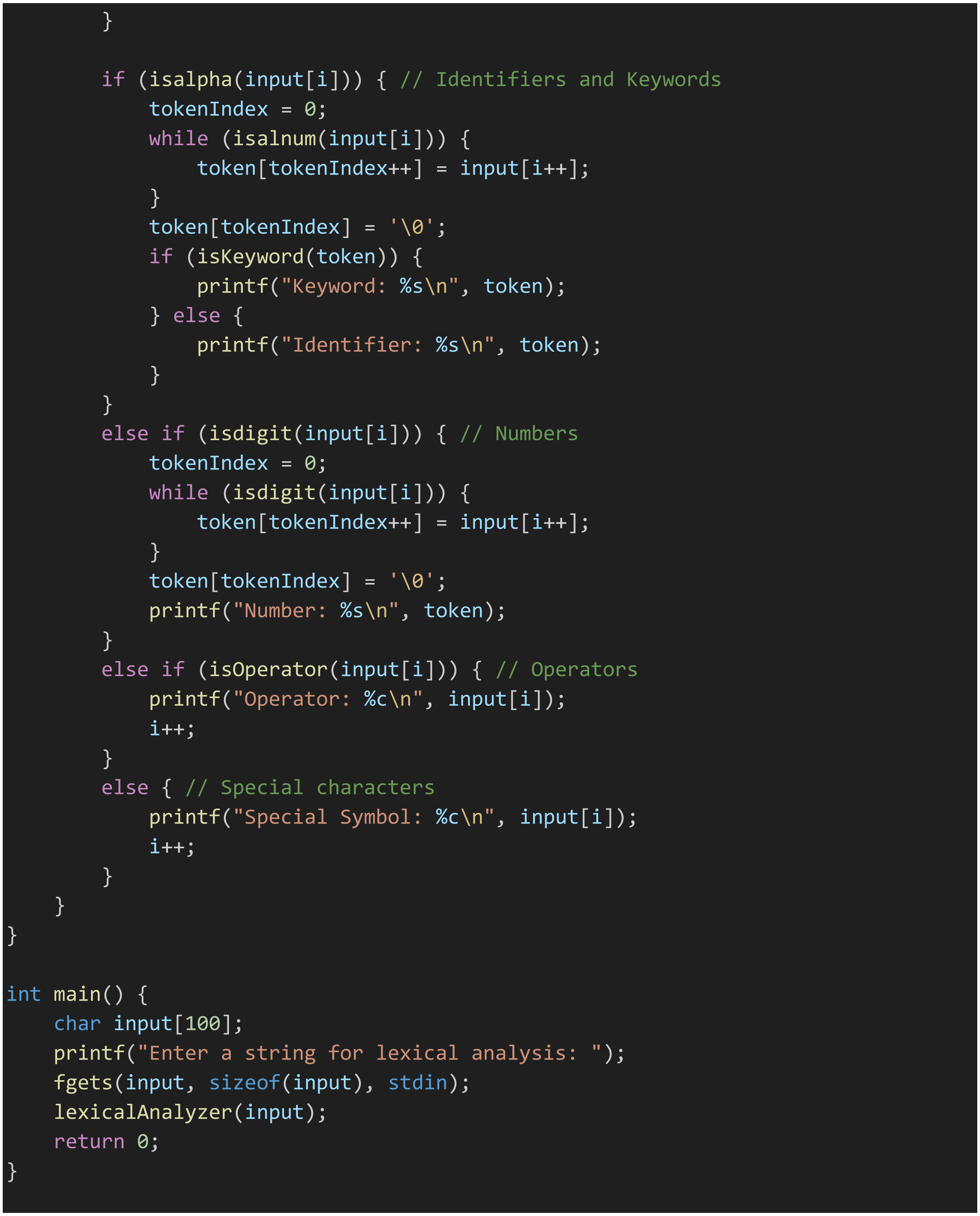


```
This is a multiline comment.
State is 6
--------------------------------
```

e. Write a program to implement Lexical Analyzer.

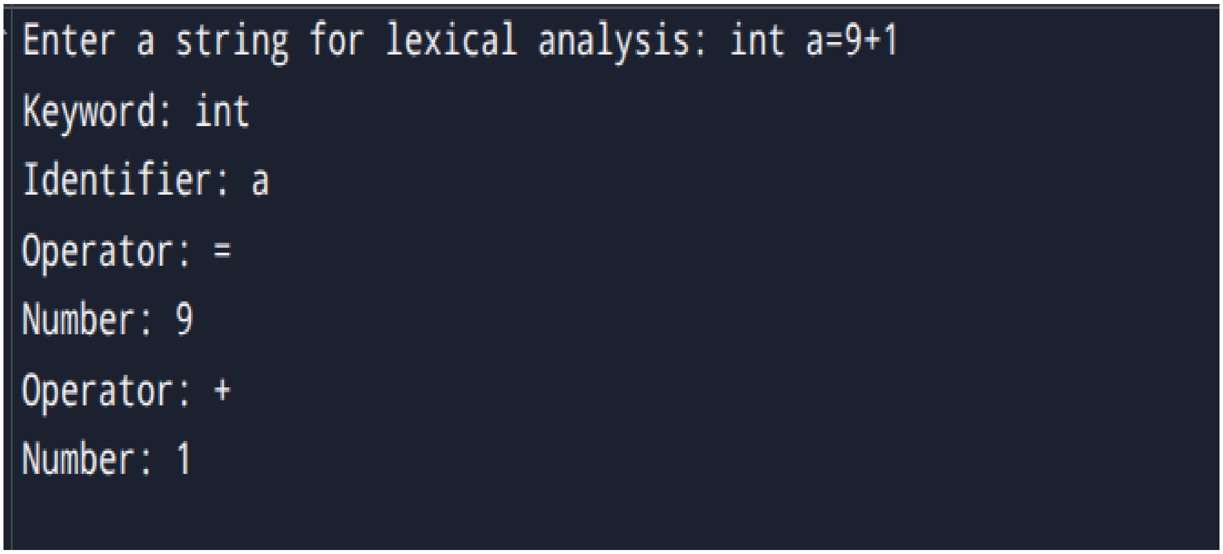
INPUT:

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

// List of keywords
const char *keywords[] = {"int", "float", "if", "else", "while", "return", "for",
"do", "switch", "case"};
#define NUM_KEYWORDS (sizeof(keywords) / sizeof(keywords[0]))

// Function to check if a string is a keyword
int isKeyword(char *str) {
int i;
    for (i = 0; i < NUM_KEYWORDS; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1;
    }
    return 0;
}

// Function to check if a character is an operator
int isOperator(char ch) {
    char operators[] = "+-*/=<>!&|";
    int i;
    for (i = 0; operators[i] != '\0'; i++) {
        if (ch == operators[i])
            return 1;
    }
    return 0;
}

void lexicalAnalyzer(char *input) {
    int i = 0;
    char token[50];
    int tokenIndex = 0;

    while (input[i] != '\0') {
        if (isspace(input[i])) {
            i++;
            continue;
```



```c
        }

        if (isalpha(input[i])) { // Identifiers and Keywords
            tokenIndex = 0;
            while (isalnum(input[i])) {
                token[tokenIndex++] = input[i++];
            }
            token[tokenIndex] = '\0';
            if (isKeyword(token)) {
                printf("Keyword: %s\n", token);
            } else {
                printf("Identifier: %s\n", token);
            }
        }
        else if (isdigit(input[i])) { // Numbers
            tokenIndex = 0;
            while (isdigit(input[i])) {
                token[tokenIndex++] = input[i++];
            }
            token[tokenIndex] = '\0';
            printf("Number: %s\n", token);
        }
        else if (isOperator(input[i])) { // Operators
            printf("Operator: %c\n", input[i]);
            i++;
        }
        else { // Special characters
            printf("Special Symbol: %c\n", input[i]);
            i++;
        }
    }
}

int main() {
    char input[100];
    printf("Enter a string for lexical analysis: ");
    fgets(input, sizeof(input), stdin);
    lexicalAnalyzer(input);
    return 0;
}
```

OUTPUT:



```
Enter a string for lexical analysis: int a=9+1
Keyword: int
Identifier: a
Operator: =
Number: 9
Operator: +
Number: 1
```

Practical-3

AIM:

To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)

Introduction:

A Lexical Analyzer converts an input stream (source code) into a sequence of tokens, which are then used by the parser in a compiler. Lex and Flex are tools designed for this purpose.

1. Lexical Analyzer Generator (LEX)

LEX is a tool used to generate lexical analyzers. It takes a set of **regular expressions** (token patterns) as input and produces a C program that can identify these tokens.

Working of LEX:

1. **Specification File:**
   A LEX program consists of three sections:

a. **Definition Section:** Declare header files and global variables.
b. **Rules Section:** Define token patterns using regular expressions.
c. **C Code Section:** Additional helper functions (optional).

2. Compilation Process:

a. The **LEX file (.l)** is compiled using lex to generate lex.yy.c.
b. The lex.yy.c file is compiled with a C compiler (gcc lex.yy.c -o output).
c. The executable processes input and tokenizes it.

Example of LEX Program:

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


%%
[0-9]+    { printf("Number: %s\n", yytext); }
[a-zA-Z]+ { printf("Identifier: %s\n", yytext); }
.         { printf("Special Symbol: %s\n", yytext); }
%%
int main() {
   yylex();
   return 0;
}
int yywrap() { return 1; }
```

Commands to Run:

1. lex filename.l
2. gcc lex.yy.c -o output
3. ./output < input.txt

2. Fast Lexical Analyzer (FLEX)

Flex is an improved and faster version of **Lex**. It provides better performance and extended functionality.

Key Features of FLEX:

- Works similarly to **Lex**, but faster.
- Generates a more optimized lex.yy.c.
- Supports additional options like debugging and performance tuning.

Example of FLEX Program:

(Same structure as LEX)

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

%%
[0-9]+    { printf("Number: %s\n", yytext); }
[a-zA-Z]+ { printf("Identifier: %s\n", yytext); }
.         { printf("Special Symbol: %s\n", yytext); }
%%

int main() {
    yylex();
    return 0;
}

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

Commands to Run:

1. flex filename.l
2. gcc lex.yy.c -o output
3. ./output < input.txt

Comparison: LEX vs FLEX

Feature	LEX	FLEX
Speed	**Slower**	**Faster**
Compatibility	**Traditional UNIX tool**	**GNU version, supports more platforms**
Debugging	**Limited**	**More debugging options**
Performance	**Basic optimization**	**Highly optimized DFA**

Conclusion:

- Lex and Flex automate the creation of lexical analyzers.
- Flex is an enhanced version of Lex and is more commonly used today.
- These tools simplify token generation in compiler design.

Practical- 4

AIM:

Implement following programs using Lex.

- **e) Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.**
- **e) Write a Lex program to take input from text file and count number of vowels and consonants.**
- **e) Write a Lex program to print out all numbers from the given file.**
- **e) Write a Lex program which adds line numbers to the given file and display the same into different file.**
- **e) Write a Lex program to printout all markup tags and HTML comments in file.**

a) **Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.**

Lex CODE: (count_lex.l)

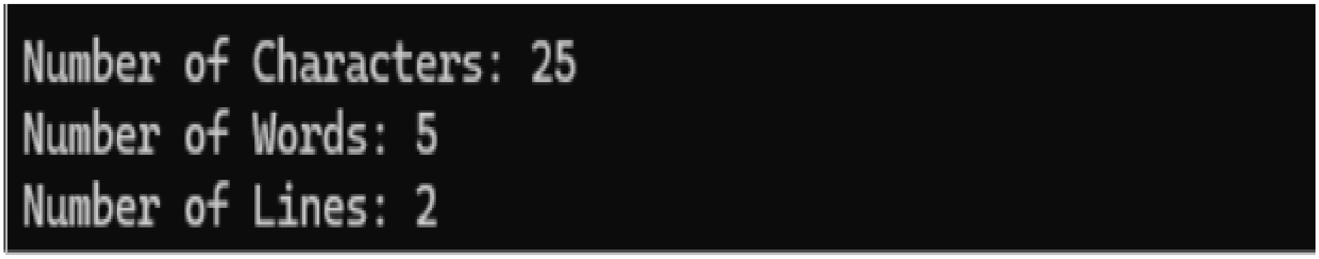
```
%{
#include <stdio.h>
int char_count = 0, word_count = 0, line_count = 0;
%}

%%

\n          { line_count++; char_count++; }
[^\n\t ]+   { word_count++; char_count += yyleng; }
.           { char_count++; }
```

```
%%

int main() {
    yylex();
    printf("\nNumber of Characters: %d", char_count);
    printf("\nNumber of Words: %d", word_count);
    printf("\nNumber of Lines: %d\n", line_count);
    return 0;
}

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

Input.txt CODE:

```
Hello World!
Lex is fun.
```

Compile and run:



```
Number of Characters: 25
Number of Words: 5
Number of Lines: 2
```

b. Write a Lex program to take input from text file and count number of vowels and consonants.

Lex Code [count1_lex.l]

```
%{
    int vowels = 0;
    int consonants = 0;
    FILE *yyin;
%}

%%

[aeiouAEIOU]    { vowels++; }
[a-zA-Z]        { consonants++; }
.|\n            { /* Ignore other characters */ }

%%
int yywrap() {
    return 1;
}



int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s input2.txt\n", argv[0]);
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (!file) {
        printf("Cannot open file %s\n", argv[1]);
        return 1;
    }

    yyin = file;
    yylex();

    printf("Number of vowels: %d\n", vowels);
    printf("Number of consonants: %d\n", consonants);

    fclose(file);
    return 0;
}
```

Input1.txt CODE:

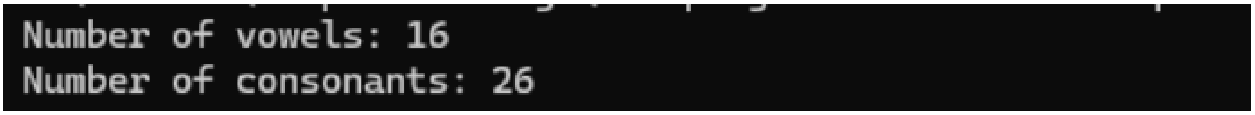
Hello World!

Lex is fun.

123

Dhruhi Shah born on 4th sep. 2003

Compile and run:



```
Number of vowels: 16
Number of consonants: 26
```

c. Write a Lex program to print out all numbers from the given file.

Lex Code [number.l]

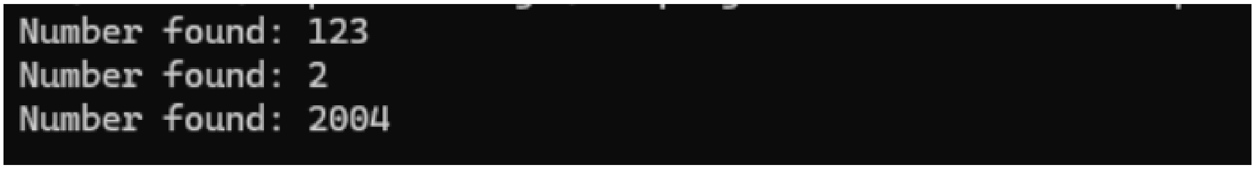
```
%{
#include <stdio.h>
%}

%%

[0-9]+(\.[0-9]+)?    { printf("Number found: %s\n", yytext); }
.|\n                 { /* Ignore all other characters */ }

%%

int yywrap() {
    return 1;
}

int main() {
    yylex();  // Start the lexical analysis
    return 0;
}
```

Input2.txt CODE:

```
Hello World!
Lex is fun.
123
```

Storm hit village on 2JAN 2004

Compile and run:



```
Number found: 123
Number found: 2
Number found: 2004
```

d. Write a Lex program to printout all markup/open tags and HTML comments in file.

Lex Code [tagscomments_lex.l]

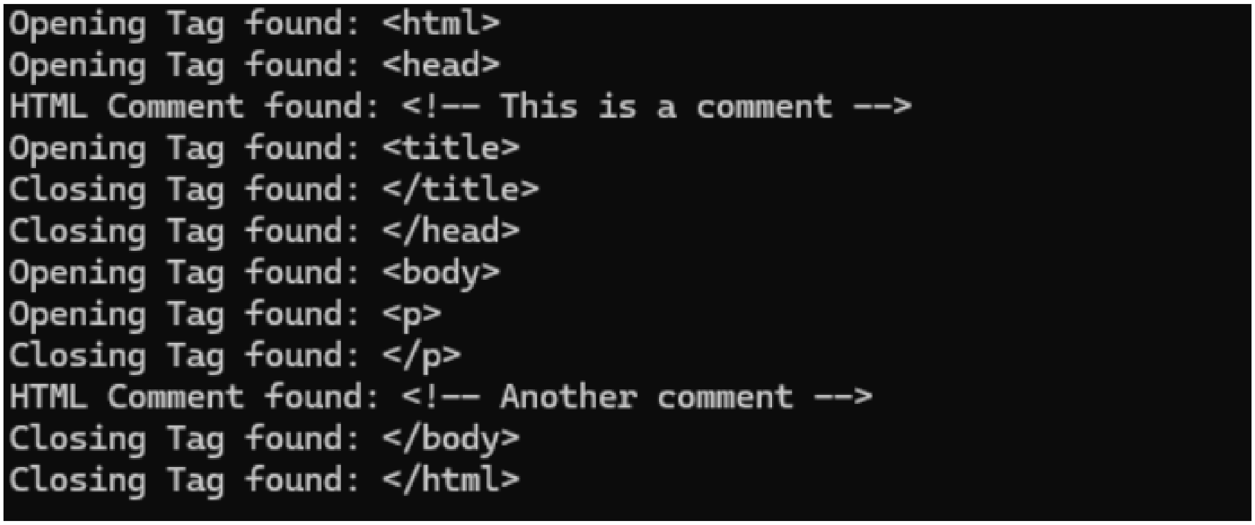
```
%{
#include <stdio.h>
%}

%%

"<!--"([^>]|[\n])*"-->"           { printf("HTML Comment found: %s\n", yytext); }
"<"[a-zA-Z][a-zA-Z0-9]*">"        { printf("Opening Tag found: %s\n", yytext); }
"</"[a-zA-Z][a-zA-Z0-9]*">"       { printf("Closing Tag found: %s\n", yytext); }
"<"[a-zA-Z][^>]*"/>"              { printf("Self-closing Tag found: %s\n", yytext); }

.|\n                              { /* Ignore other content */ }

%%

int yywrap() { return 1; }

int main() {
   yylex();
   return 0;
}
```

input3.html CODE:

```
<html>
<head>
```

```
<!-- This is a comment -->
<title>Page Title</title>
</head>
<body>
<p>Welcome to the page!</p>
<!-- Another comment -->
</body>
</html>
```

Compile and run:



```
Opening Tag found: <html>
Opening Tag found: <head>
HTML Comment found: <!-- This is a comment -->
Opening Tag found: <title>
Closing Tag found: </title>
Closing Tag found: </head>
Opening Tag found: <body>
Opening Tag found: <p>
Closing Tag found: </p>
HTML Comment found: <!-- Another comment -->
Closing Tag found: </body>
Closing Tag found: </html>
```

Practical-5

AIM:

a. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.
b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

a. **Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.**

INPUT comment_lex.l:

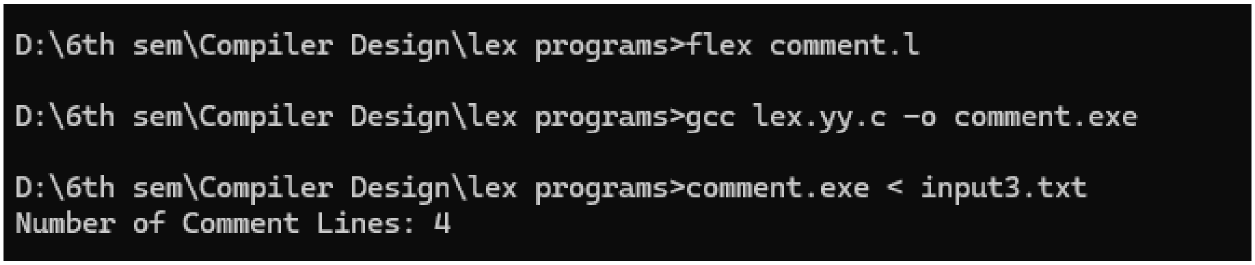
```
%{
#include <stdio.h>
#include <stdlib.h>
int comment_count = 0;
%}

%%
\/\/.*      { comment_count++; }  // Single-line comments
\/\*[^*]*\*+([^/*][^*]*\*+)*\/  { comment_count++; } // Multi-line comments
.|\n       { /* Ignore all characters, since we are not writing to a file */ }
%%

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

```
int main() {
    yyin = stdin;  // Read input from standard input (CMD)
    yylex();

    printf("Number of Comment Lines: %d\n", comment_count);
    return 0;
}
```

Input4.txt

```
#include <stdio.h>

/* This is a multi-line comment
   explaining the main function */
int main() {
    // This is a single-line comment
    printf("Hello, World!\n"); // Print statement
    return 0; /* Return statement */
}
```

Compile and run:



```
D:\6th sem\Compiler Design\lex programs>flex comment.l

D:\6th sem\Compiler Design\lex programs>gcc lex.yy.c -o comment.exe

D:\6th sem\Compiler Design\lex programs>comment.exe < input3.txt
Number of Comment Lines: 4
```

b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

INPUT tokenizer_lex.l:

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

DIGIT      [0-9]
LETTER     [a-zA-Z]
IDENTIFIER  {LETTER}({LETTER}|{DIGIT})*
NUMBER     {DIGIT}+(\.{DIGIT}+)?
OPERATOR    [+\-*/%=><|&!]
SPECIAL    [(){}[\];,]
LITERAL    \"(\\.|[^"\\])*\"

%%

"auto"      { printf("Keyword: %s\n", yytext); }
"break"     { printf("Keyword: %s\n", yytext); }
"case"      { printf("Keyword: %s\n", yytext); }
"char"      { printf("Keyword: %s\n", yytext); }
"const"     { printf("Keyword: %s\n", yytext); }
"continue"  { printf("Keyword: %s\n", yytext); }
"default"   { printf("Keyword: %s\n", yytext); }
"do"        { printf("Keyword: %s\n", yytext); }
```

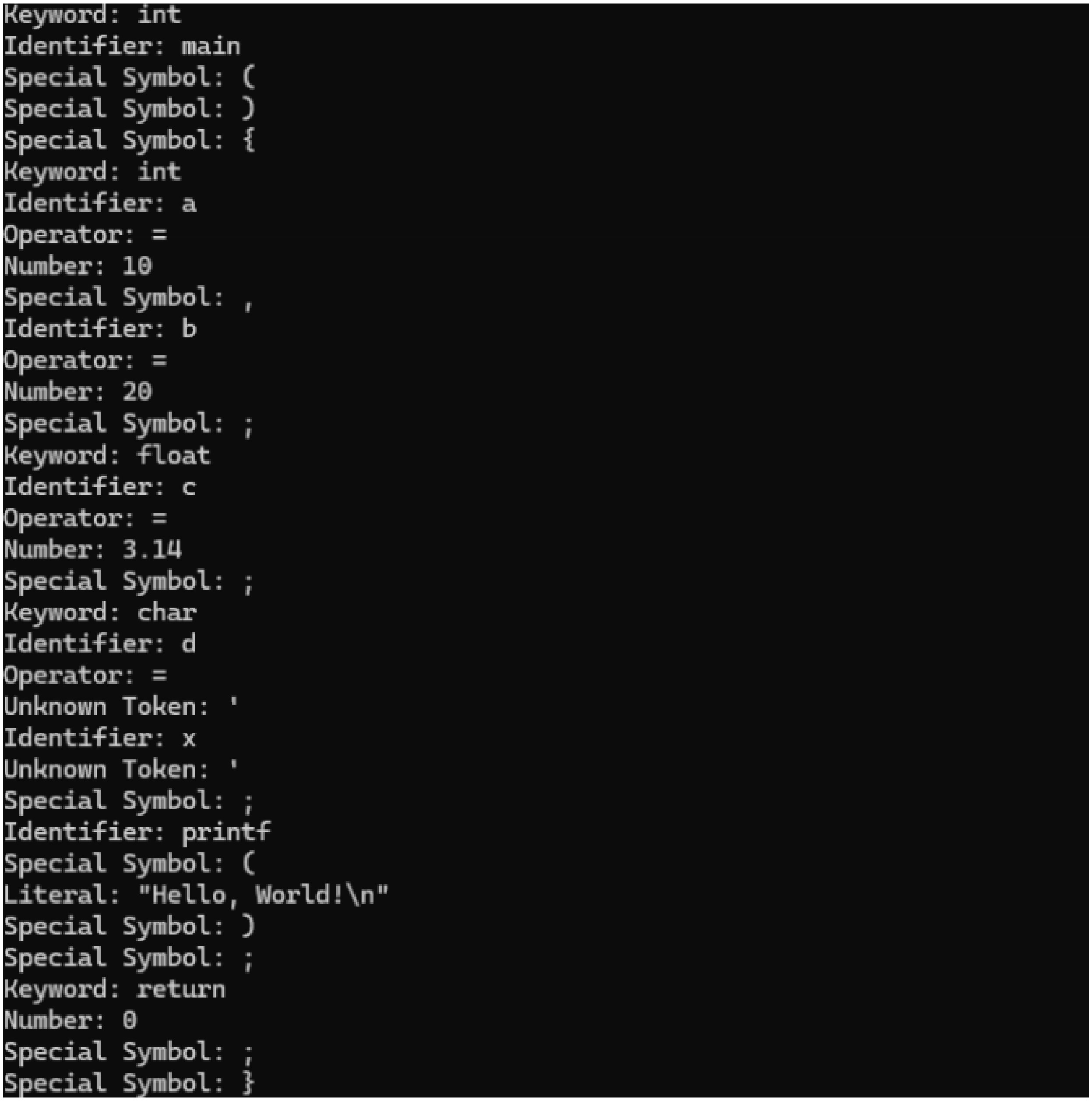
```
"double"    { printf("Keyword: %s\n", yytext); }
"else"      { printf("Keyword: %s\n", yytext); }
"enum"      { printf("Keyword: %s\n", yytext); }
"extern"    { printf("Keyword: %s\n", yytext); }
"float"     { printf("Keyword: %s\n", yytext); }
"for"       { printf("Keyword: %s\n", yytext); }
"goto"      { printf("Keyword: %s\n", yytext); }
"if"        { printf("Keyword: %s\n", yytext); }
"int"       { printf("Keyword: %s\n", yytext); }
"long"      { printf("Keyword: %s\n", yytext); }
"register"  { printf("Keyword: %s\n", yytext); }
"return"    { printf("Keyword: %s\n", yytext); }
"short"     { printf("Keyword: %s\n", yytext); }
"signed"    { printf("Keyword: %s\n", yytext); }
"sizeof"    { printf("Keyword: %s\n", yytext); }
"static"    { printf("Keyword: %s\n", yytext); }
"struct"    { printf("Keyword: %s\n", yytext); }
"switch"    { printf("Keyword: %s\n", yytext); }
"typedef"   { printf("Keyword: %s\n", yytext); }
"union"     { printf("Keyword: %s\n", yytext); }
"unsigned"  { printf("Keyword: %s\n", yytext); }
"void"      { printf("Keyword: %s\n", yytext); }
"volatile"  { printf("Keyword: %s\n", yytext); }
"while"     { printf("Keyword: %s\n", yytext); }

{IDENTIFIER}   { printf("Identifier: %s\n", yytext); }
{NUMBER}       { printf("Number: %s\n", yytext); }
{OPERATOR}     { printf("Operator: %s\n", yytext); }
```

```
{SPECIAL}      { printf("Special Symbol: %s\n", yytext); }
{LITERAL}      { printf("Literal: %s\n", yytext); }

[ \t\n]        { /* Ignore whitespace and newlines */ }

.              { printf("Unknown Token: %s\n", yytext); }

%%

int yywrap() {
    return 1;
}

int main() {
    yylex();
    return 0;
}
```

Input5.txt

```
int main() {
    int a = 10, b = 20;
    float c = 3.14;
    char d = 'x';
    printf("Hello, World!\n");

    return 0;
}
```

Compile and run:



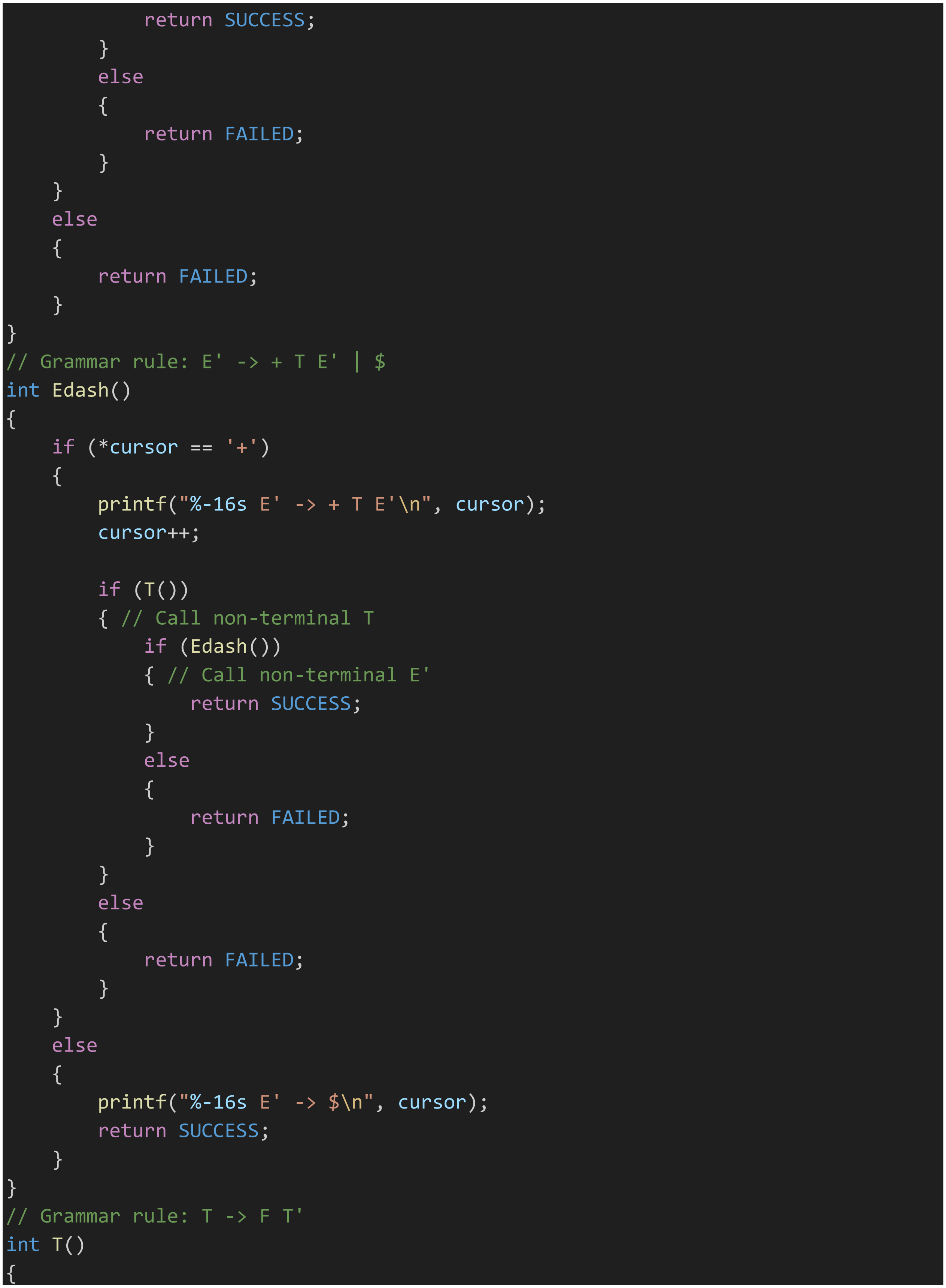
```
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Keyword: int
Identifier: a
Operator: =
Number: 10
Special Symbol: ,
Identifier: b
Operator: =
Number: 20
Special Symbol: ;
Keyword: float
Identifier: c
Operator: =
Number: 3.14
Special Symbol: ;
Keyword: char
Identifier: d
Operator: =
Unknown Token: '
Identifier: x
Unknown Token: '
Special Symbol: ;
Identifier: printf
Special Symbol: (
Literal: "Hello, World!\n"
Special Symbol: )
Special Symbol: ;
Keyword: return
Number: 0
Special Symbol: ;
Special Symbol: }
```

Practical- 6

AIM: Program to implement Recursive Descent Parsing in C.

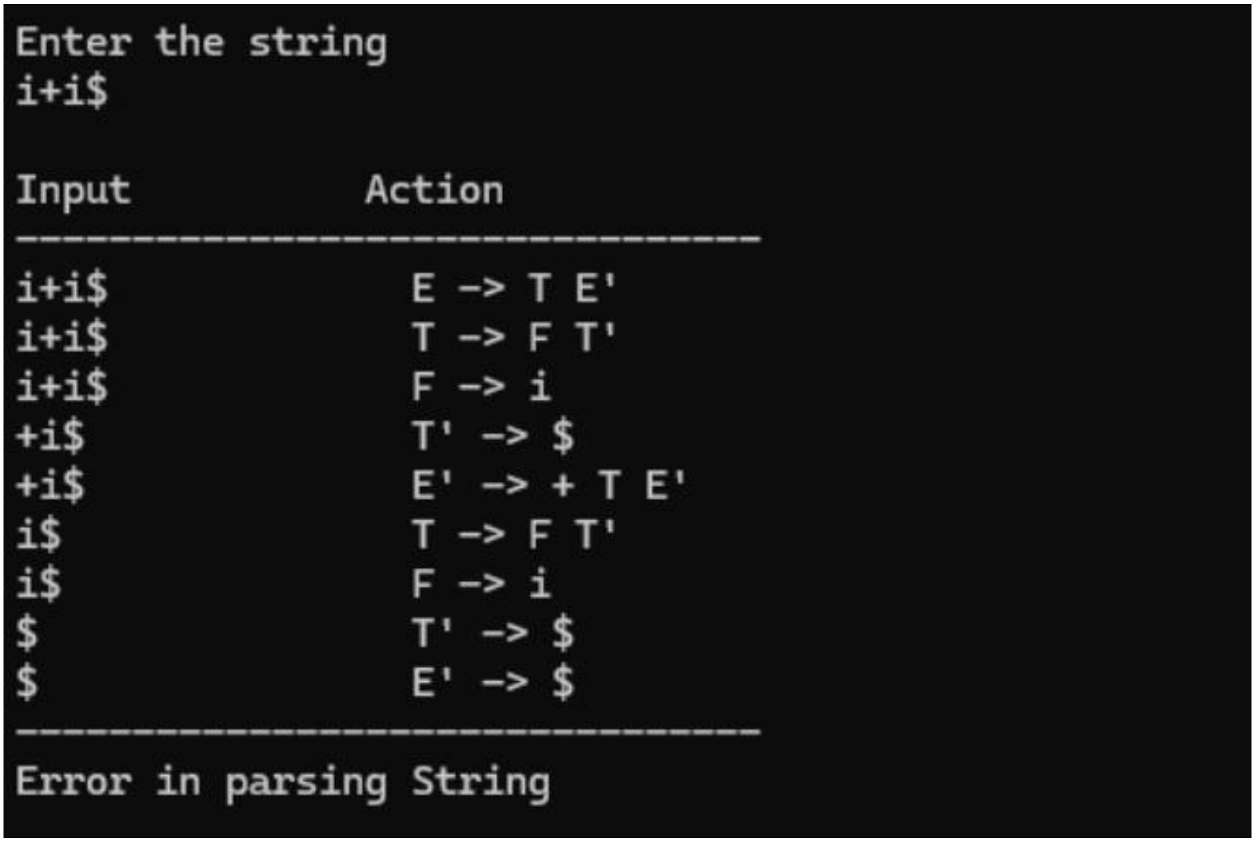
INPUT:

```c
#include <stdio.h>
#include <string.h>
#define SUCCESS 1
#define FAILED 0
// Function prototypes
int E(), Edash(), T(), Tdash(), F();
const char *cursor;
char string[64];
int main()
{
    puts("Enter the string");
    scanf("%s", string); // Read input from the user
    cursor = string;
    puts("");
    puts("Input          Action");
    puts("--------------------------------");
    // Call the starting non-terminal E
    if (E() && *cursor == '\0')
    { // If parsing is successful and the cursor has reached the end
        puts("--------------------------------");
        puts("String is successfully parsed");
        return 0;
    }
    else
    {
        puts("--------------------------------");
        puts("Error in parsing String");
        return 1;
    }
}
// Grammar rule: E -> T E'
int E()
{
    printf("%-16s E -> T E'\n", cursor);
    if (T())
    { // Call non-terminal T
        if (Edash())
        { // Call non-terminal E'
```



```
            return SUCCESS;
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        return FAILED;
    }
}
// Grammar rule: E' -> + T E' | $
int Edash()
{
    if (*cursor == '+')
    {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;

        if (T())
        { // Call non-terminal T
            if (Edash())
            { // Call non-terminal E'
                return SUCCESS;
            }
            else
            {
                return FAILED;
            }
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        printf("%-16s E' -> $\n", cursor);
        return SUCCESS;
    }
}
// Grammar rule: T -> F T'
int T()
{
    printf("%-16s T -> F T'\n", cursor);
    if (F())
    { // Call non-terminal F
        if (Tdash())
        { // Call non-terminal T'
            return SUCCESS;
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        return FAILED;
    }
}

// Grammar rule: T' -> * F T' | $
int Tdash()
{
    if (*cursor == '*')
    {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;

        if (F())
        { // Call non-terminal F
            if (Tdash())
            { // Call non-terminal T'
                return SUCCESS;
            }
            else
            {
                return FAILED;
            }
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}
// Grammar rule: F -> ( E ) | i
int F()
{
    if (*cursor == '(')
    {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E())
        { // Call non-terminal E
            if (*cursor == ')')
            {
                cursor++;
                return SUCCESS;
            }
            else
            {
                return FAILED;
            }
        }
        else
        {
            return FAILED;
        }
    }
    else if (*cursor == 'i')
    {
        printf("%-16s F -> i\n", cursor);
        cursor++;
        return SUCCESS;
    }
    else
    {
        return FAILED;
    }
}
```

OUTPUT:



```
Enter the string
i+i$

Input          Action
--------------------------------
i+i$             E -> T E'
i+i$             T -> F T'
i+i$             F -> i
+i$              T' -> $
+i$              E' -> + T E'
i$               T -> F T'
i$               F -> i
$                T' -> $
$                E' -> $
--------------------------------
Error in parsing String
```

Practical-7

AIM:

a. **To Study about Yet Another Compiler-Compiler(YACC).**

b. **Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .**

c. **Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.**

d. **Create Yacc and Lex specification files are used to convert infix expression to postfix expression.**

a. To Study about Yet Another Compiler-Compiler(YACC).

What is YACC?

- YACC (Yet Another Compiler-Compiler) is a tool used in compiler design to generate parsers. It helps you build the syntax analysis part of a compiler.
- It was developed by Stephen C. Johnson at AT&T Bell Labs.

Why is YACC used?

- Writing a parser manually (like recursive descent) is complex and error-prone.
- YACC automates this by generating C code for the parser from a grammar specification.
- It works well with lex, the lexical analyzer generator.

How does YACC work?

- You write a grammar using BNF (Backus-Naur Form) or similar syntax.
- You assign semantic actions to grammar rules (using C code).
- YACC generates a parser in C that uses a bottom-up parsing algorithm (usually LALR(1)).
- The parser works with lex to analyze tokens.

Structure of a YACC file:

A YACC source file has three sections, separated by %%:

```
%{
   // Declarations (C code, headers)
%}

%token ID NUM // Token definitions

%%

E : E '+' T   { printf("Adding\n"); }
  | T         { /* do nothing */ }
  ;

T : T '*' F   { printf("Multiplying\n"); }
  | F         { /* do nothing */ }
  ;

F : '(' E ')'
  | ID
  | NUM
  ;

%%

// Additional C code (main function etc.)
```

YACC and LEX Integration

- LEX handles scanning/tokenizing (splits input into tokens).
- YACC handles parsing (checks if token sequence is valid as per grammar).
- They work together to build front ends for compilers.

Advantages of YACC

- Speeds up parser development.
- Helps build robust parsers for programming languages.
- Well-suited for formal language processing tasks.

b. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .

Expr_lexer.l CODE:

```
%{
   #include "expr.tab.h"
   #include <stdlib.h>
%}

%%

[0-9]+       { yylval.ival = atoi(yytext); return NUMBER; }
[a-zA-Z]+    { yylval.ival = 0; return ID; }
[ \t]+       ; // skip whitespace
\n           { return '\n'; }
.            { return yytext[0]; }

%%

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

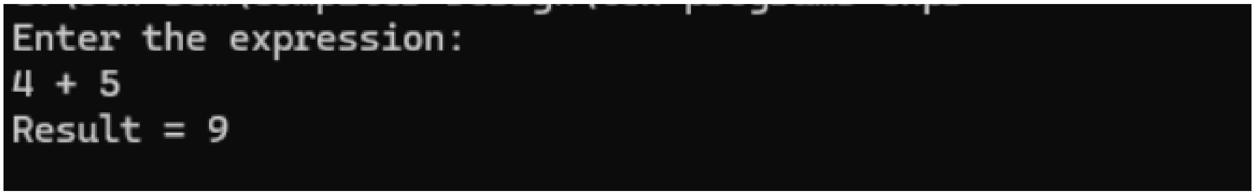
Expr_parser.y CODE:

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

   void yyerror(const char *s);
```

```
    int yylex(void);
%}

%union {
    int ival;
}

%token <ival> NUMBER
%token <ival> ID
%type <ival> E

%left '+' '-'
%left '*' '/'

%%

input:
    E '\n'      { printf("Result = %d\n", $1); }
    ;

E:
    E '+' E     { $$ = $1 + $3; }
  | E '-' E     { $$ = $1 - $3; }
  | E '*' E     { $$ = $1 * $3; }
  | E '/' E     { $$ = $1 / $3; }
  | '-' E       { $$ = -$2; }
  | '(' E ')'   { $$ = $2; }
  | NUMBER      { $$ = $1; }
  | ID          { $$ = $1; }
  ;
```

```
%%

int main(void) {
    printf("Enter the expression:\n");
    yyparse();
    return 0;
}

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

OUTPUT:



```
Enter the expression:
4 + 5
Result = 9
```

c. **Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.**

calculator.l CODE:

```
%{
   #include "calc.tab.h"
   #include <stdlib.h>
%}

%%

[0-9]+       { yylval.ival = atoi(yytext); return NUMBER; }
[ \t]+       ; // skip whitespace
\n           { return '\n'; }
.            { return yytext[0]; }

%%

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

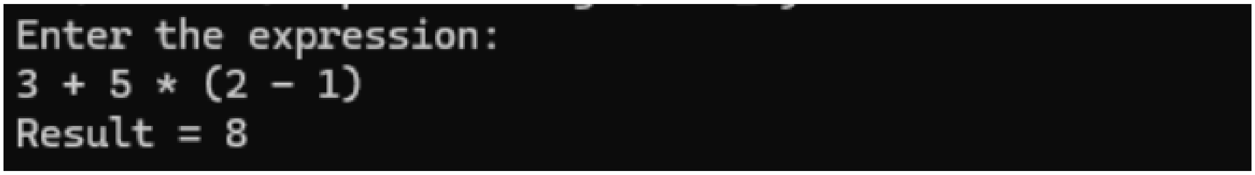
calculator.y CODE:

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

void yyerror(const char *s);
```

```
int yylex(void);
%}

%union {
   int ival;
}

%token <ival> NUMBER
%type <ival> expr

%left '+' '-'
%left '*' '/'
%start input

%%

input:
   expr '\n'      { printf("Result = %d\n", $1); }
   ;

expr:
   expr '+' expr   { $$ = $1 + $3; }
 | expr '-' expr   { $$ = $1 - $3; }
 | expr '*' expr   { $$ = $1 * $3; }
 | expr '/' expr   {
                  if ($3 == 0) {
                    yyerror("Division by zero");
                    YYABORT; // Exit the parsing process immediately
                  } else {
                    $$ = $1 / $3;
                  }
```

```
                }
  | '(' expr ')'    { $$ = $2; }
  | NUMBER          { $$ = $1; }
  ;

%%

int main() {
    printf("Enter the expression:\n");
    return yyparse();
}

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

OUTPUT:



```
Enter the expression:
3 + 5 * (2 - 1)
Result = 8
```

d. Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

infix_to_postfix.l CODE:

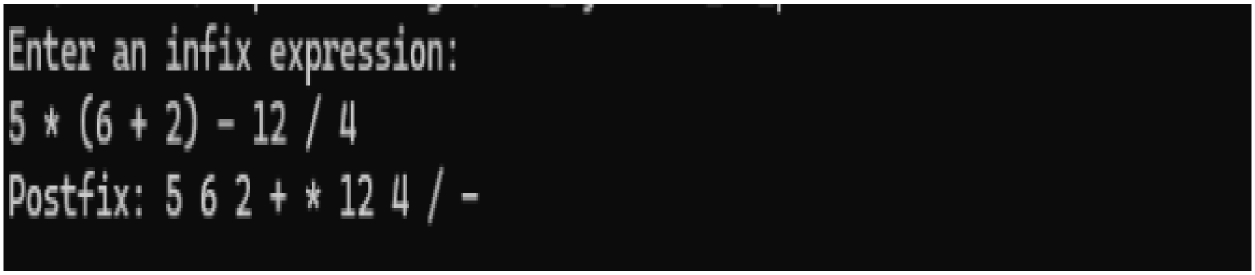
```
%{
#include "infix_to_postfix.tab.h"
#include <stdlib.h>
#include <string.h>
%}

DIGIT   [0-9]
WS      [ \t\r]+

%%

{DIGIT}+    {
                yylval.str = strdup(yytext);
                return NUMBER;
            }
"("        { return '('; }
")"        { return ')'; }
"+"        { return '+'; }
"-"        { return '-'; }
"*"        { return '*'; }
"/"        { return '/'; }
{WS}        { /* skip whitespace */ }
\n         { return '\n'; }
.          { return yytext[0]; }
```

```
%%

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

infix_to_postfix.y CODE:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

// custom asprintf implementation for Windows
int asprintf(char **strp, const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    int size = vsnprintf(NULL, 0, fmt, args);
    va_end(args);

    if (size < 0) return -1;

    *strp = (char *)malloc(size + 1);
    if (!*strp) return -1;

    va_start(args, fmt);
    vsnprintf(*strp, size + 1, fmt, args);
    va_end(args);
    return size;
}

void yyerror(const char *s);
int yylex(void);
%}

%union {
    char *str;
}

%token <str> NUMBER
%left '+' '-'
%left '*' '/'
%token '(' ')'

%type <str> expr

%%

input:
    /* empty */
    | input expr '\n' {
        printf("Postfix: %s\n", $2);
        free($2);
    }
    ;

expr:
      NUMBER              { $$ = strdup($1); free($1); }
    | expr '+' expr       { asprintf(&$$, "%s %s +", $1, $3); free($1); free($3); }
    | expr '-' expr        { asprintf(&$$, "%s %s -", $1, $3); free($1); free($3); }
    | expr '*' expr        { asprintf(&$$, "%s %s *", $1, $3); free($1); free($3); }
    | expr '/' expr        { asprintf(&$$, "%s %s /", $1, $3); free($1); free($3); }
    | '(' expr ')'         { $$ = $2; }
    ;

%%

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

int main() {
    printf ("Enter an infix expression:\n");
    yyparse();
    return 0;
}
```

OUTPUT:



```
Enter an infix expression:
5 * (6 + 2) - 12 / 4
Postfix: 5 6 2 + * 12 4 / -
```

!!THANKS!!