LAB MANUAL

of

COMPILER DESIGN LABORATORY (CSE605)

Bachelor of Technology (CSE)

By

Vanraj Jhala (22000419)

Third Year, Semester 6

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring Semester

(2024-2025)

List of Experiments

Experiment - 1

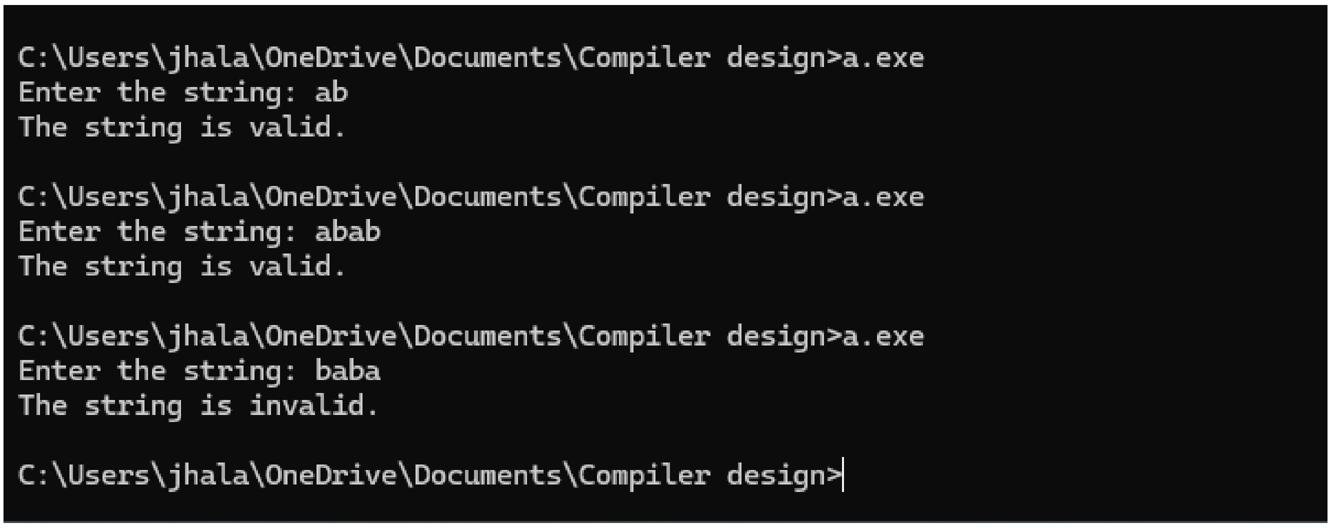
Aim : Write a C program for the following :

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

Code :

```c
#include <stdio.h>

int main() {
    char input[100];  // Array to store the input string
    int state = 0, i = 0;  // `state` keeps track of the DFA state, `i` is the index
for traversal

    // Prompt user for input
    printf("Enter the string: ");
    scanf("%s", input);  // Read the string from the user

    // Process each character in the string
    while (input[i] != '\0') {
        switch (state) {
            case 0:  // Initial state
                if (input[i] == 'a')
                    state = 1;  // Move to state 1 if the first character is 'a'
                else
                    state = 2;  // Move to state 2 if the first character is 'b'
                break;

            case 1:  // Valid state (Once 'a' is found, remain in state 1)
                if (input[i] != 'a' && input[i] != 'b')
                    state = 2;  // If an invalid character appears, move to state 2
                break;

            case 2:  // Invalid state (Once in state 2, remain there)
                break;  // No need to process further, remains invalid
        }
        i++;  // Move to the next character
    }

// Determine if the string is valid based on the final state
if (state == 1)
    printf("The string is valid.\n");  // If final state is 1, the string is valid

else
    printf("The string is invalid.\n");  // Otherwise, it's invalid
    return 0;
}
```

Output :



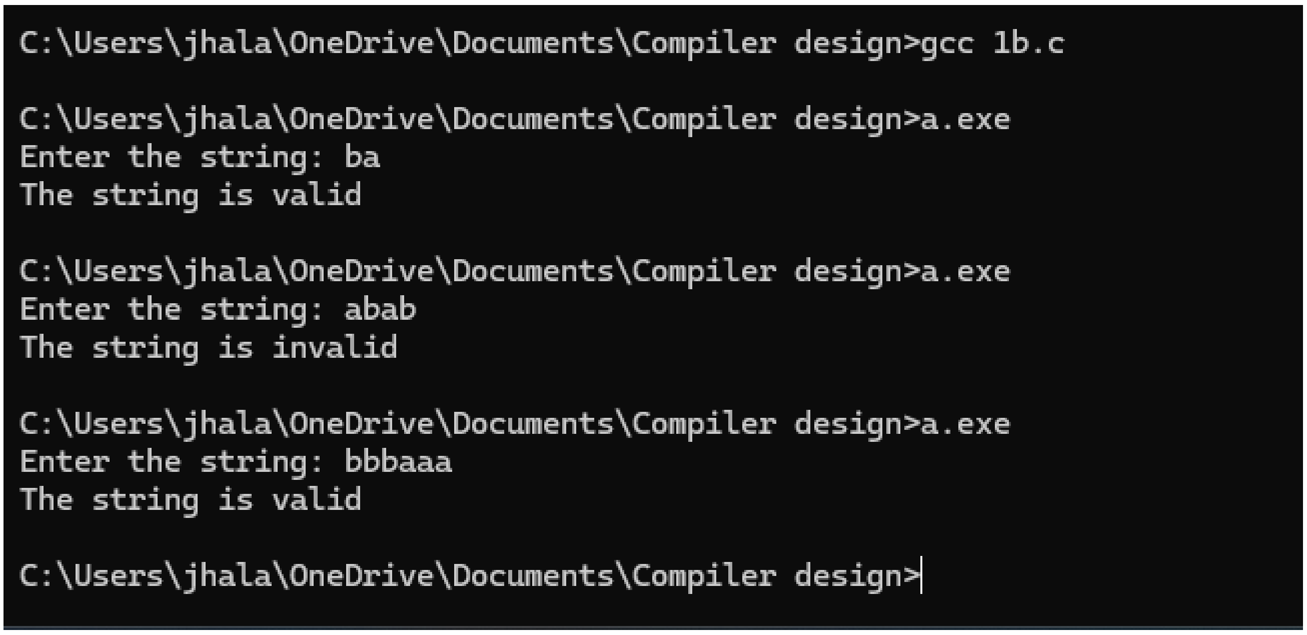
```
C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter the string: ab
The string is valid.

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter the string: abab
The string is valid.

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter the string: baba
The string is invalid.

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

B. Write a program to recognise strings end with ‘a’.

Code :

```c
#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 == 1) {
        printf("The string is valid\n");
    } else {
        printf("The string is invalid\n");
    }

    return 0;
}
```

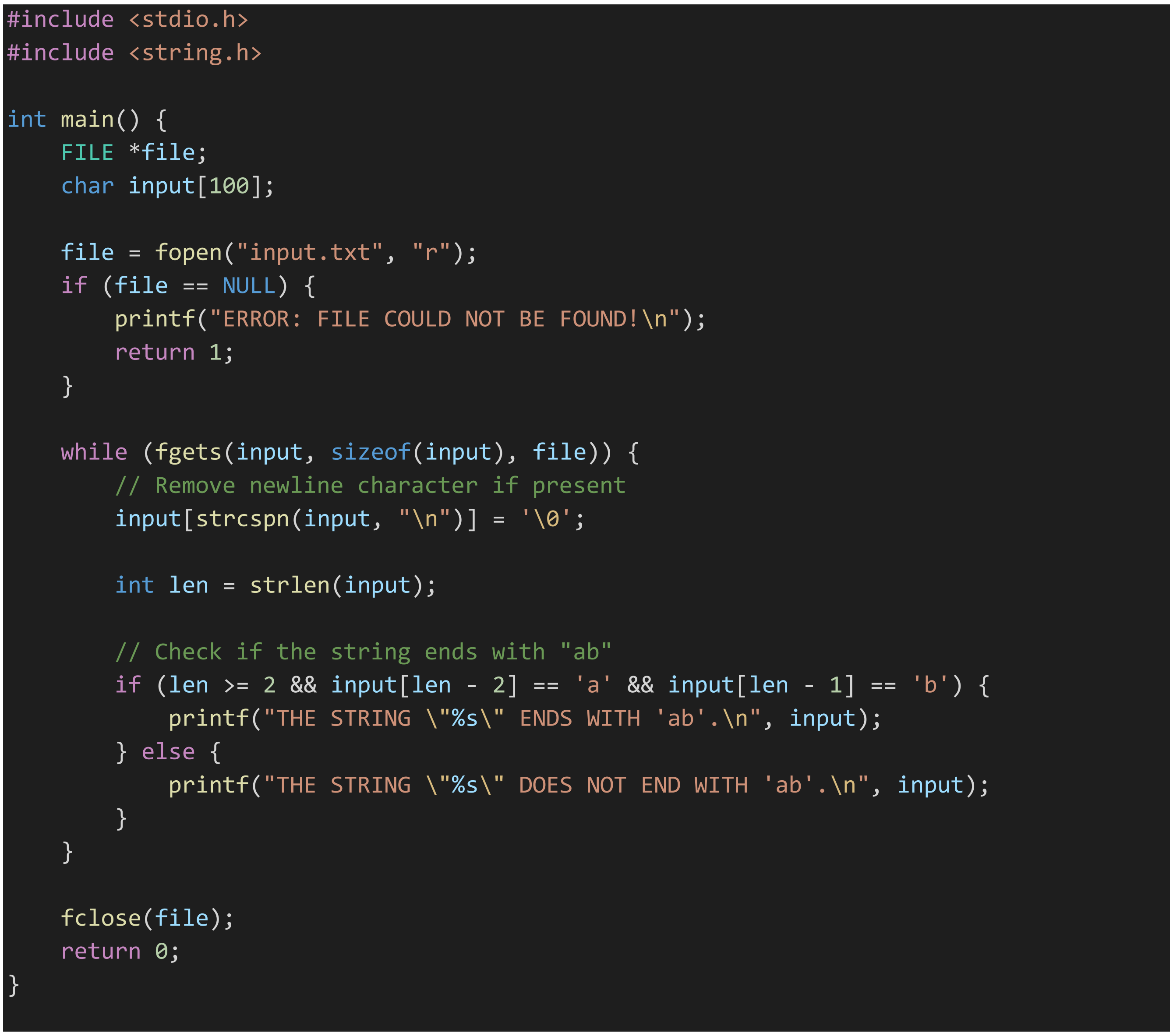
Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>gcc 1b.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter the string: ba
The string is valid

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter the string: abab
The string is invalid

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter the string: bbbaaa
The string is valid

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

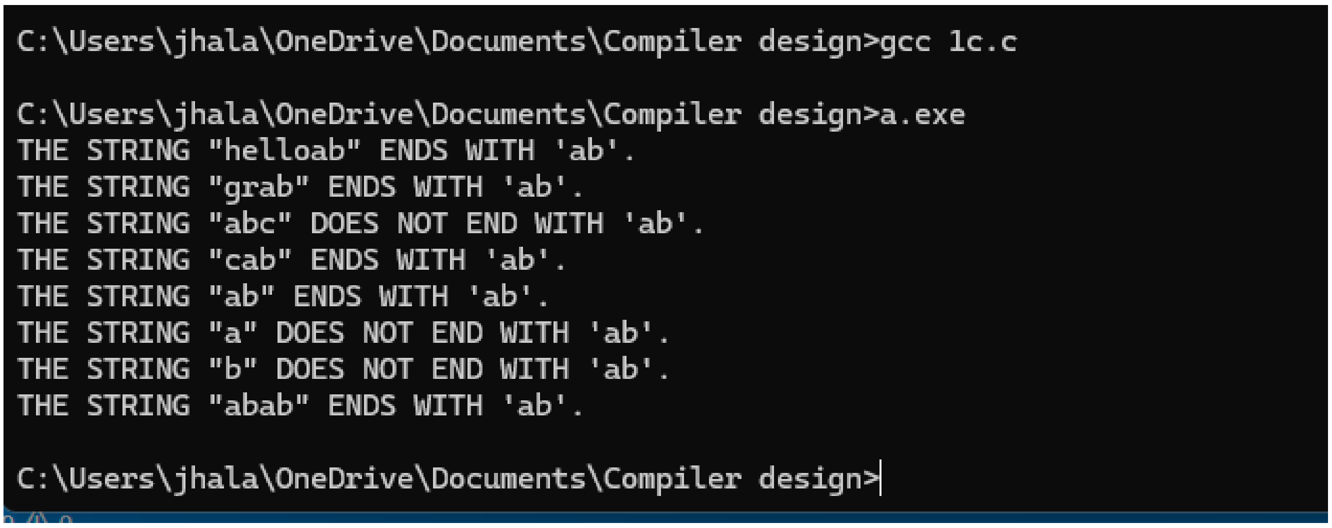
C. Write a program to recognise strings end with ‘ab’. Take the input from text file.

Code :



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

int main() {
    FILE *file;
    char input[100];

    file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("ERROR: FILE COULD NOT BE FOUND!\n");
        return 1;
    }

    while (fgets(input, sizeof(input), file)) {
        // Remove newline character if present
        input[strcspn(input, "\n")] = '\0';

        int len = strlen(input);

        // Check if the string ends with "ab"
        if (len >= 2 && input[len - 2] == 'a' && input[len - 1] == 'b') {
            printf("THE STRING \"%s\" ENDS WITH 'ab'.\n", input);
        } else {
            printf("THE STRING \"%s\" DOES NOT END WITH 'ab'.\n", input);
        }
    }

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

Output :

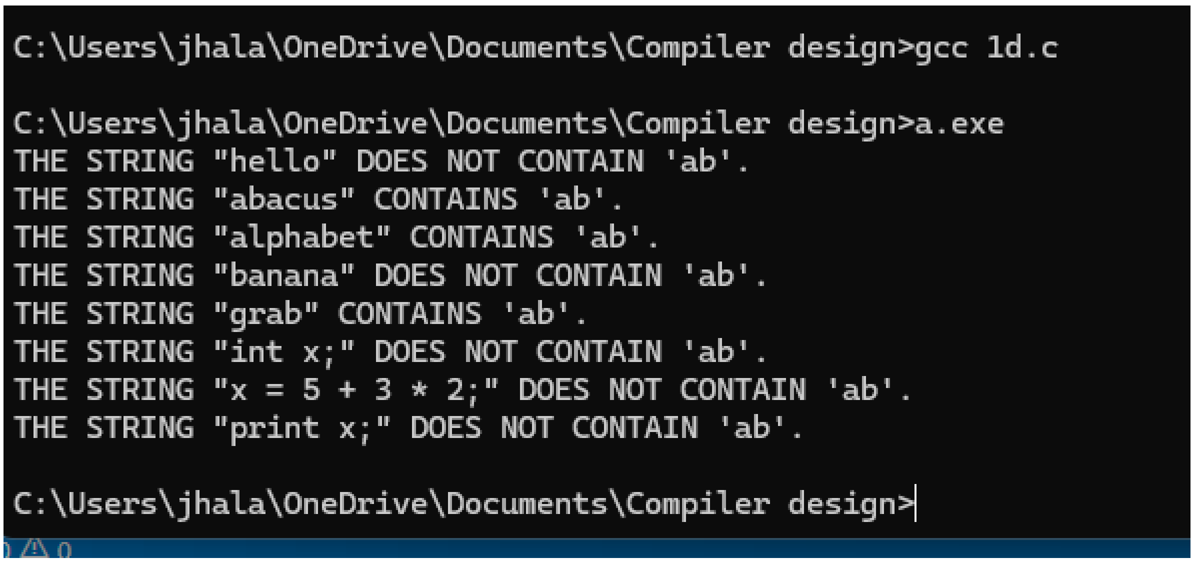


```
C:\Users\jhala\OneDrive\Documents\Compiler design>gcc 1c.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
THE STRING "helloab" ENDS WITH 'ab'.
THE STRING "grab" ENDS WITH 'ab'.
THE STRING "abc" DOES NOT END WITH 'ab'.
THE STRING "cab" ENDS WITH 'ab'.
THE STRING "ab" ENDS WITH 'ab'.
THE STRING "a" DOES NOT END WITH 'ab'.
THE STRING "b" DOES NOT END WITH 'ab'.
THE STRING "abab" ENDS WITH 'ab'.

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

D. Write a program to recognise strings contains ‘ab’. Take the input from text file.
Code :

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

int main() {
    FILE *file;
    char input[100];
    int i, state;

    file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("ERROR: FILE COULD NOT BE FOUND!\n");
        return 1;
    }

    while (fgets(input, sizeof(input), file)) {
        input[strcspn(input, "\n")] = 0;  // Remove trailing newline
        i = 0;
        state = 0;

        // Finite automaton to detect substring "ab"
        while (input[i] != '\0') {
            switch (state) {
                case 0:
                    if (input[i] == 'a') {
                        state = 1;
                    }
                    break;
                case 1:
                    if (input[i] == 'b') {
                        state = 2;
                    } else if (input[i] == 'a') {
                        state = 1;
                    } else {
                        state = 0;
                    }
                    break;
                case 2:
                    break; // Found "ab", no need to continue
            }
            i++;
        }

        if (state == 2)
            printf("THE STRING \"%s\" CONTAINS 'ab'.\n", input);
        else
            printf("THE STRING \"%s\" DOES NOT CONTAIN 'ab'.\n", input);
    }
    fclose(file);
    return 0;
}
```

Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>gcc 1d.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
THE STRING "hello" DOES NOT CONTAIN 'ab'.
THE STRING "abacus" CONTAINS 'ab'.
THE STRING "alphabet" CONTAINS 'ab'.
THE STRING "banana" DOES NOT CONTAIN 'ab'.
THE STRING "grab" CONTAINS 'ab'.
THE STRING "int x;" DOES NOT CONTAIN 'ab'.
THE STRING "x = 5 + 3 * 2;" DOES NOT CONTAIN 'ab'.
THE STRING "print x;" DOES NOT CONTAIN 'ab'.

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

Experiment - 2

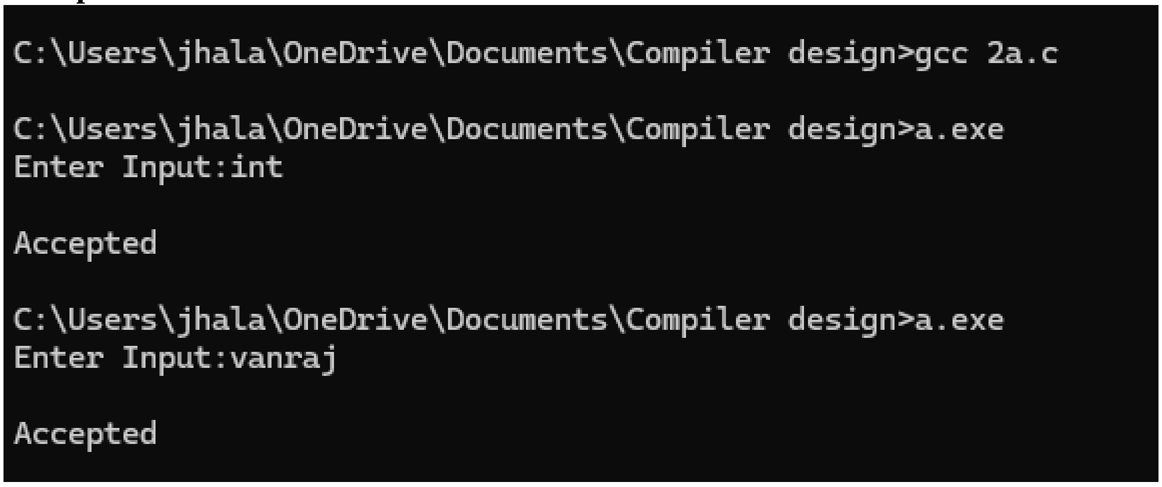
Aim : Write a Python program for the following :

A. Write a program to recognise the valid identifiers.

Code :

```
#include<stdio.h>
#include<conio.h>
int main()
{
char input[10]; int state = 0 , i=0;
printf("Enter Input:"); scanf("%s",input); while (i<=10)
{
switch (state)
{
case 0:
if(input[i]=='i'){ state = 1;
}
else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z')
|| (input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
{
state = 5;
}
break; case 1:
if(input[i]=='n'){ state = 2;
}
else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z')
|| (input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
{
state = 5;
}
break; case 2:
if(input[i]=='t'){ state = 3;
}

else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z')
|| (input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
{
state = 5;
}
else{
state = 0;
}
break; case 3:
if(input[i]=='\0'){ state = 0;
}
else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z')
|| (input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
{
state = 5;
}
break; case 5:
if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z') ||
(input[i]
>= 'a' && input[i] <= 'z') ||( input[i] == '_'))
{
state = 5;
}
else if(input[i]=='\0'){ state=0;
}
break;
}
i++;
}
printf("\n"); if(state == 0){
printf("Accepted\n");
}
else {
printf("The input is not recognized.(%d) \n",state);} return 0;
}
```

Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>gcc 2a.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter Input:int

Accepted

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Enter Input:vanraj

Accepted
```

B. Write a program to recognise the valid operators.

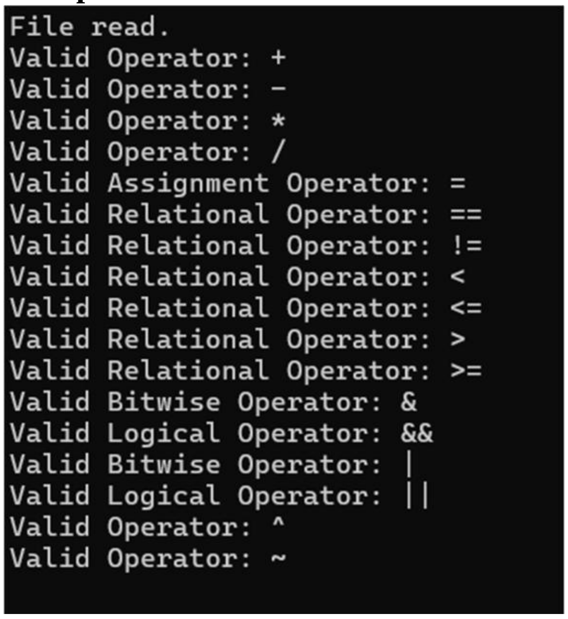
Code :

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

void main() {
char c, buffer[1000], lexeme[10]; int i = 0, j = 0, f = 0, state = 0;
FILE *fp = fopen("input_operators.txt", "r");

if (!fp) {
printf("Error opening file.\n"); return;
}

// Read entire file into buffer
while ((c = fgetc(fp)) != EOF && j < 1000) { buffer[j++] = c;
}
buffer[j] = '\0'; fclose(fp);
printf("File read.\n"); i = 0;
while (buffer[i] != '\0') {
c = buffer[i]; switch (state) {
case 0:
if (isspace(c)) {
// Skip spaces
} else if (c == '=') { lexeme[f++] = c; state = 1;
} else if (c == '!') { lexeme[f++] = c; state = 2;
} else if (c == '<') { lexeme[f++] = c; state = 3;

} else if (c == '>') { lexeme[f++] = c; state = 4;
} else if (c == '&') { lexeme[f++] = c; state = 5;
} else if (c == '|') { lexeme[f++] = c; state = 6;
} else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '^' || c
== '~') { lexeme[0] = c;
lexeme[1] = '\0';
printf("Valid Operator: %s\n", lexeme); f = 0;
} else {
printf("Invalid character encountered: %c\n", c);
}
break;

case 1: // after '=' if (c == '=') {
lexeme[f++] = c; lexeme[f] = '\0';
printf("Valid Relational Operator: %s\n", lexeme); // ==
} else {
lexeme[f] = '\0';
printf("Valid Assignment Operator: %s\n", lexeme); // = i--; // reprocess current
char
}
f = 0;
state = 0; break;

case 2: // after '!' if (c == '=') {
```

```
lexeme[f++] = c; lexeme[f] = '\0';
printf("Valid Relational Operator: %s\n", lexeme); // !=
} else {
lexeme[f] = '\0';
printf("Invalid Operator: %s\n", lexeme); // only ! is invalid here i--;

}
f = 0;
state = 0; break;

case 3: // after '<' if (c == '=') {
lexeme[f++] = c; lexeme[f] = '\0';
printf("Valid Relational Operator: %s\n", lexeme); // <=
} else {
lexeme[f] = '\0';
printf("Valid Relational Operator: %s\n", lexeme); // < i--;
}
f = 0;
state = 0; break;

case 4: // after '>' if (c == '=') {
lexeme[f++] = c; lexeme[f] = '\0';
printf("Valid Relational Operator: %s\n", lexeme); // >=
} else {
lexeme[f] = '\0';
printf("Valid Relational Operator: %s\n", lexeme); // > i--;
}
f = 0;
state = 0; break;

case 5: // after '&' if (c == '&') {
lexeme[f++] = c; lexeme[f] = '\0';
printf("Valid Logical Operator: %s\n", lexeme); // &&
} else {
lexeme[f] = '\0';
printf("Valid Bitwise Operator: %s\n", lexeme); // & i--;

}
f = 0;
state = 0; break;

case 6: // after '|' if (c == '|') {
lexeme[f++] = c; lexeme[f] = '\0';
printf("Valid Logical Operator: %s\n", lexeme); // ||
} else {
lexeme[f] = '\0';
printf("Valid Bitwise Operator: %s\n", lexeme); // | i--;
}
f = 0;
state = 0; break;
```

```
default:
printf("Unknown error.\n"); f = 0;
state = 0; break;
}
i++;
}
}
```

Output :



```
File read.
Valid Operator: +
Valid Operator: -
Valid Operator: *
Valid Operator: /
Valid Assignment Operator: =
Valid Relational Operator: ==
Valid Relational Operator: !=
Valid Relational Operator: <
Valid Relational Operator: <=
Valid Relational Operator: >
Valid Relational Operator: >=
Valid Bitwise Operator: &
Valid Logical Operator: &&
Valid Bitwise Operator: |
Valid Logical Operator: ||
Valid Operator: ^
Valid Operator: ~
```

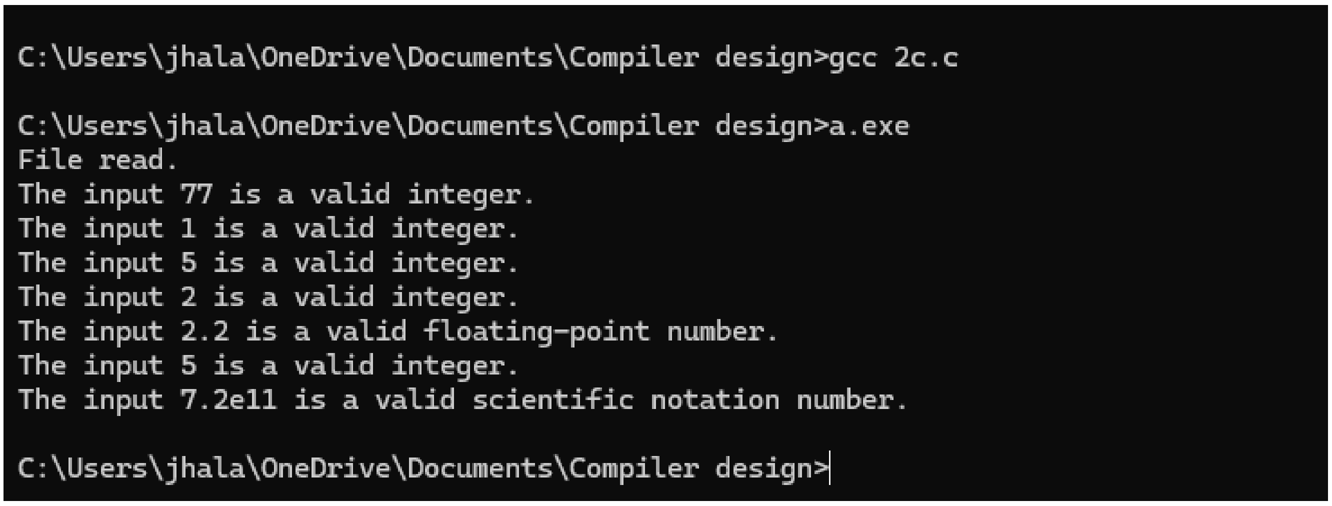
C. Write a program to recognise the valid number.

Code :

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

int main() {
    char c, buffer[1000], lexeme[1000];
    int i = 0, state = 0, f = 0, j = 0;

    FILE *fp = fopen("input_Number.txt", "r");
    if (!fp) {
        printf("Could not open file.\n");
        return 1;
    }

    while ((c = fgetc(fp)) != EOF && j < 1000) {
        buffer[j++] = c;
    }
    buffer[j] = '\0';
    printf("File read.\n");
    fclose(fp);

    while (buffer[i] != '\0') {
        c = buffer[i];
        switch (state) {
            case 0:
                if (isdigit(c)) {
                    state = 1;
                    lexeme[f++] = c;
                } else if (c == '+' || c == '-') {
                    state = 0;
                    lexeme[f++] = c;
                } else if (isspace(c)) {
                    // skip whitespace
                } else {
                    state = 99;
                }
                break;

            case 1:
                if (isdigit(c)) {
                    state = 1;
                    lexeme[f++] = c;
                } else if (c == '.') {
                    state = 2;
                    lexeme[f++] = c;
                } else if (c == 'e' || c == 'E') {
                    state = 4;
                    lexeme[f++] = c;
                } else {
                    lexeme[f] = '\0';
                    printf("The input %s is a valid integer.\n", lexeme);
                    f = 0;
                    state = 0;
                    i--;
                }
                break;

            case 2:
                if (isdigit(c)) {
                    state = 3;
                    lexeme[f++] = c;
                } else {
                    lexeme[f] = '\0';
                    printf("%s is an invalid floating point input.\n", lexeme);
                    f = 0;
                    state = 0;
                    i--;
                }
                break;

            case 3:
                if (isdigit(c)) {
                    state = 3;
                    lexeme[f++] = c;
                } else if (c == 'e' || c == 'E') {
                    state = 4;
                    lexeme[f++] = c;
                } else {
                    lexeme[f] = '\0';
                    printf("The input %s is a valid floating-point number.\n",
lexeme);

                    f = 0;
                    state = 0;
                    i--;
                }
                break;

            case 4:
                if (isdigit(c)) {
                    state = 6;
                    lexeme[f++] = c;
                } else if (c == '+' || c == '-') {
                    state = 5;
                    lexeme[f++] = c;
                } else {
                    lexeme[f] = '\0';
                    printf("%s is an invalid scientific notation.\n", lexeme);
                    f = 0;
```

```
                    state = 0;
                    i--;
                }
                break;

            case 5:
                if (isdigit(c)) {
                    state = 6;
                    lexeme[f++] = c;
                } else {
                    lexeme[f] = '\0';
                    printf("%s is an invalid scientific notation.\n", lexeme);
                    f = 0;
                    state = 0;
                    i--;
                }
                break;

            case 6:
                if (isdigit(c)) {
                    state = 6;
                    lexeme[f++] = c;
                } else {
                    lexeme[f] = '\0';
                    printf("The input %s is a valid scientific notation number.\n",
lexeme);
                    f = 0;
                    state = 0;
                    i--;
                }
                break;

            default:
                printf("Invalid character encountered: %c\n", c);
                f = 0;
                state = 0;
                break;
        }
        i++;
    }

    // Final token check
    if (f != 0) {
        lexeme[f] = '\0';
        if (state == 1)
            printf("The input %s is a valid integer.\n", lexeme);
        else if (state == 3)
            printf("The input %s is a valid floating-point number.\n", lexeme);
        else if (state == 6)
            printf("The input %s is a valid scientific notation number.\n", lexeme);
    }
    return 0;
}
```

Output :



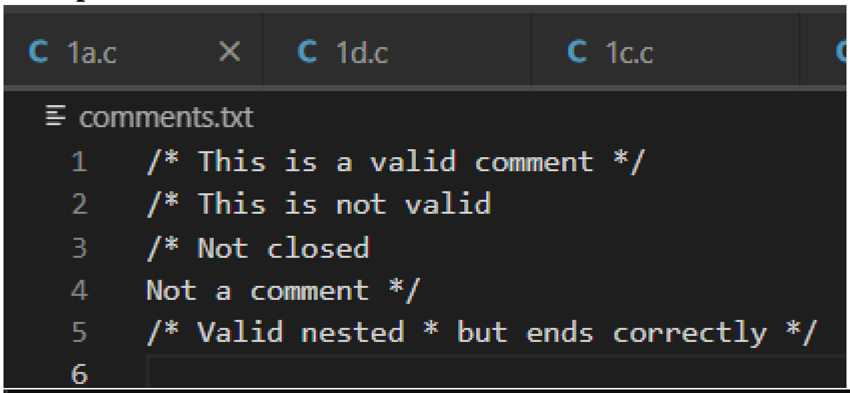
```
C:\Users\jhala\OneDrive\Documents\Compiler design>gcc 2c.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
File read.
The input 77 is a valid integer.
The input 1 is a valid integer.
The input 5 is a valid integer.
The input 2 is a valid integer.
The input 2.2 is a valid floating-point number.
The input 5 is a valid integer.
The input 7.2e11 is a valid scientific notation number.

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

D. Write a program to recognise the valid comments.

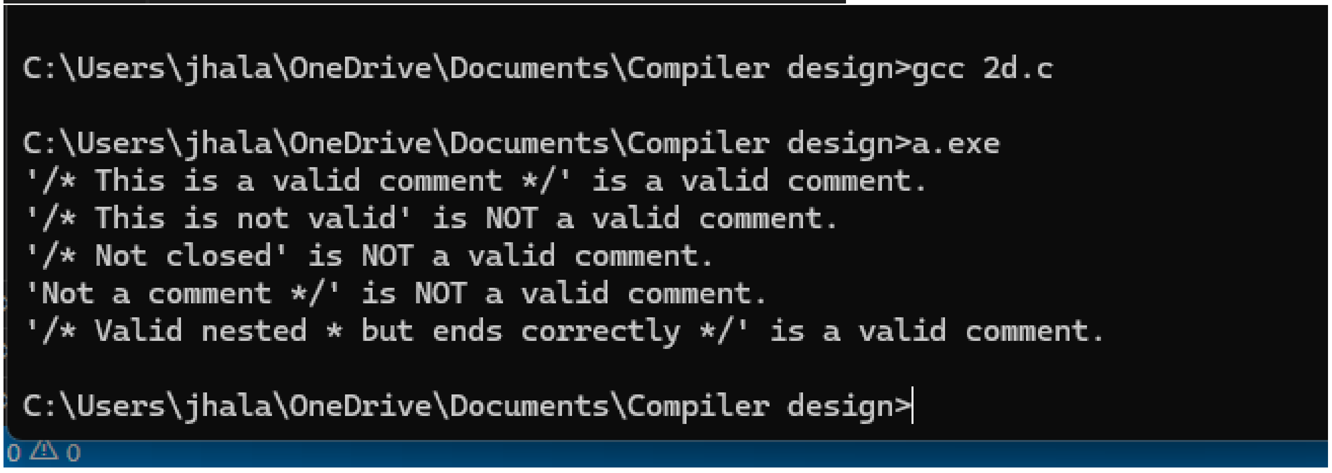
Code :

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

// Function to check if a line is a valid single-line or multi-line comment
bool is_valid_comment(const char *line) {
    // Trim leading and trailing spaces (you can add more handling here)
    char trimmed_line[256];
    int len = strlen(line);
    int i = 0, j = 0;

    // Remove leading spaces
    while (i < len && (line[i] == ' ' || line[i] == '\t')) {
        i++;
    }

    // Copy the rest of the string
    while (i < len) {
        trimmed_line[j++] = line[i++];
    }
    trimmed_line[j] = '\0';

    // Check for single-line comments "//"
    if (strncmp(trimmed_line, "//", 2) == 0) {
        return true;
    }
    // Check for multi-line comments "/* */"
    if (strncmp(trimmed_line, "/*", 2) == 0 && trimmed_line[len - 2] == '*' &&
trimmed_line[len - 1] == '/') {
        return true;
    }

    return false;
}

int main() {
    FILE *file;
    char line[256];

    file = fopen("comments.txt", "r");
    if (file == NULL) {
        printf("ERROR: 'comments.txt' file not found!\n");
        return 1;
    }

    while (fgets(line, sizeof(line), file)) {
        // Remove newline character from line
        line[strcspn(line, "\n")] = 0;
        if (is_valid_comment(line)) {
            printf("'%s' is a valid comment.\n", line);
        } else {
            printf("'%s' is NOT a valid comment.\n", line);
        }
    }

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

Output :



```
comments.txt
/* This is a valid comment */
/* This is not valid
/* Not closed
Not a comment */
/* Valid nested * but ends correctly */
```

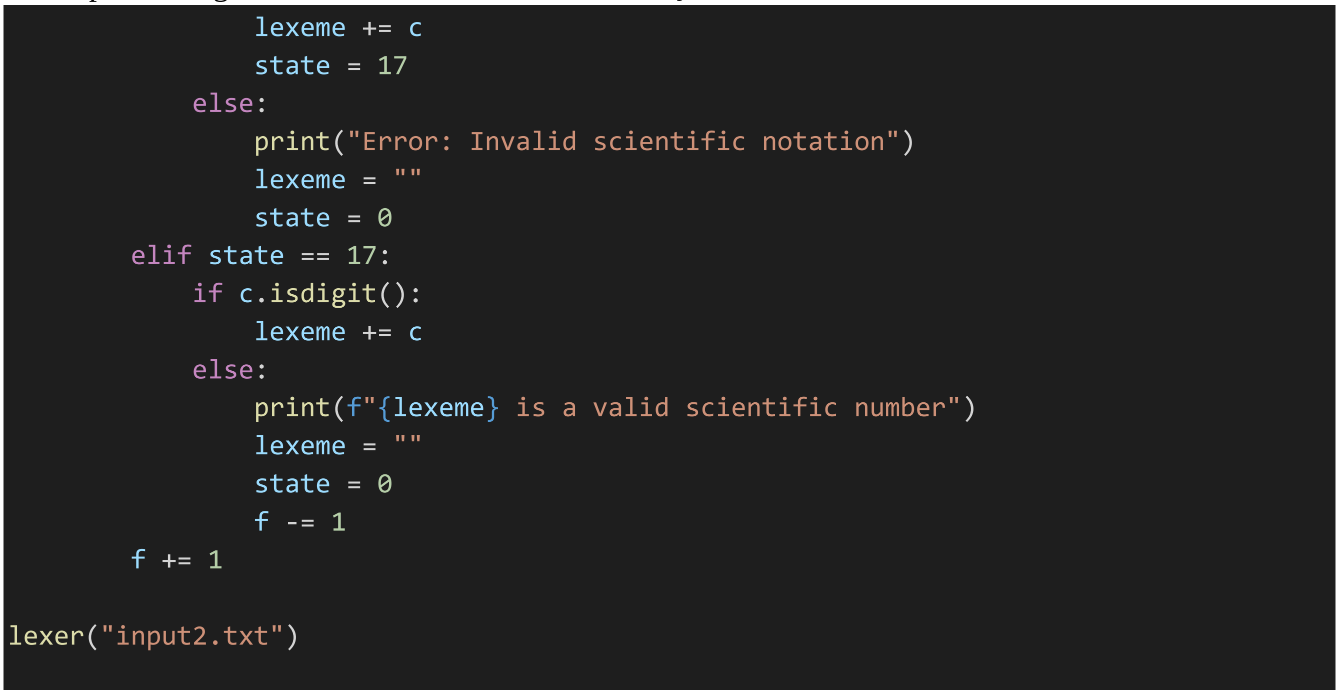


```
C:\Users\jhala\OneDrive\Documents\Compiler design>gcc 2d.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
'/* This is a valid comment */' is a valid comment.
'/* This is not valid' is NOT a valid comment.
'/* Not closed' is NOT a valid comment.
'Not a comment */' is NOT a valid comment.
'/* Valid nested * but ends correctly */' is a valid comment.

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

E. Program to implement Lexical Analyser.

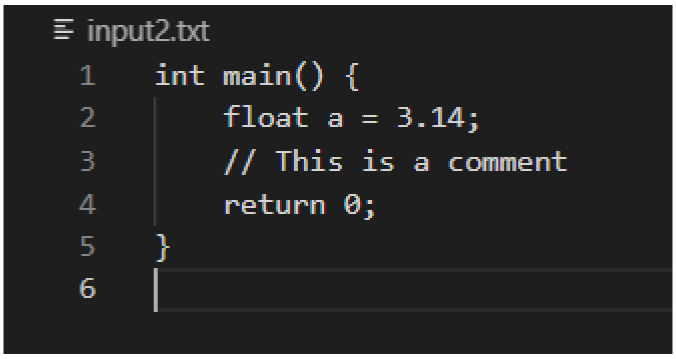
Code :

```
def check(lexeme):
    keywords = {"auto", "break", "case", "char", "const", "continue", "default",
"do",
                "double", "else", "enum", "extern", "float", "for", "goto", "if",
                "inline", "int", "long", "register", "restrict", "return", "short",
"signed",
                "sizeof", "static", "struct", "switch", "typedef", "union",
"unsigned",
                "void", "volatile", "while"}
    if lexeme in keywords:
        print(f"{lexeme} is a keyword")
    else:
        print(f"{lexeme} is an identifier")

def lexer(filename):
    try:
        with open(filename, "r") as f:
            buffer = f.read()
    except:
        print("Error opening file")
        return

    lexeme = ""
    state = 0
    f = 0
    while f < len(buffer):
        c = buffer[f]
        if state == 0:
            if c.isspace():
                pass
            elif c.isalpha() or c == '_':
                lexeme = c
                state = 1
            elif c.isdigit():
                lexeme = c
                state = 13
            elif c == '/':
                state = 11
            else:
                print(f"{c} is an operator")
        elif state == 1:
            if c.isalnum() or c == '_':
                lexeme += c
            else:
                check(lexeme)
                lexeme = ""
                state = 0
                f -= 1
        elif state == 11:
            if c == '/':
                while f < len(buffer) and buffer[f] != '\n':
                    f += 1
                state = 0
            elif c == '*':
                f += 1
                while f < len(buffer) - 1 and not (buffer[f] == '*' and buffer[f + 1]
== '/'):
                    f += 1
                f += 2
                state = 0
            else:
                print("/ is an operator")
                state = 0
                f -= 1
        elif state == 13:
            if c.isdigit():
                lexeme += c
            elif c == '.':
                state = 14
                lexeme += c
            elif c in "Ee":
                state = 16
                lexeme += c
            else:
                print(f"{lexeme} is a valid integer")
                lexeme = ""
                state = 0
                f -= 1
        elif state == 14:
            if c.isdigit():
                lexeme += c
                state = 15
            else:
                print("Error: Invalid floating point format")
                lexeme = ""
                state = 0
        elif state == 15:
            if c.isdigit():
                lexeme += c
            elif c in "Ee":
                state = 16
                lexeme += c
            else:
                print(f"{lexeme} is a valid floating point number")
                lexeme = ""
                state = 0
                f -= 1
        elif state == 16:
            if c in "+-" or c.isdigit():
```



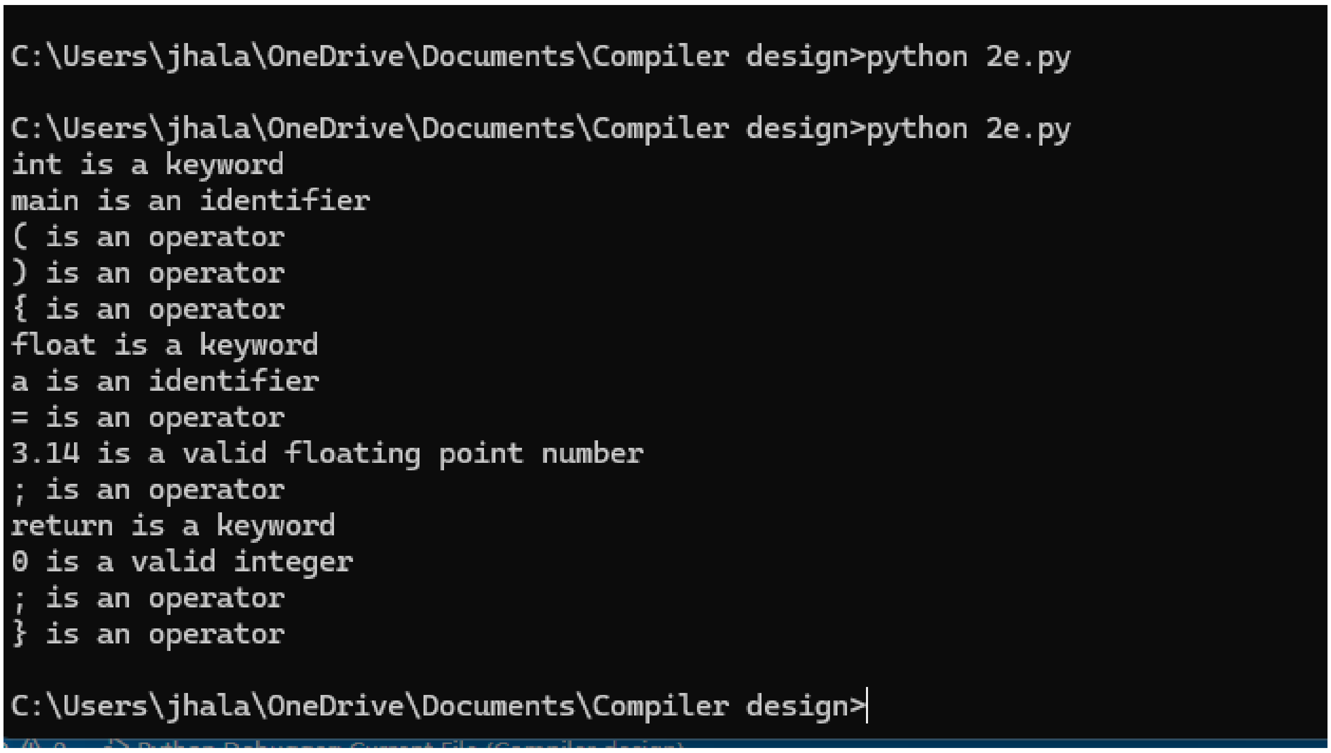
```
                lexeme += c
                state = 17
            else:
                print("Error: Invalid scientific notation")
                lexeme = ""
                state = 0
        elif state == 17:
            if c.isdigit():
                lexeme += c
            else:
                print(f"{lexeme} is a valid scientific number")
                lexeme = ""
                state = 0
                f -= 1
        f += 1

lexer("input2.txt")
```

Output :



input2.txt

```
int main() {
    float a = 3.14;
    // This is a comment
    return 0;
}

```



```
C:\Users\jhala\OneDrive\Documents\Compiler design>python 2e.py

C:\Users\jhala\OneDrive\Documents\Compiler design>python 2e.py
int is a keyword
main is an identifier
( is an operator
) is an operator
{ is an operator
float is a keyword
a is an identifier
= is an operator
3.14 is a valid floating point number
; is an operator
return is a keyword
0 is a valid integer
; is an operator
} is an operator

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

Experiment - 3

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

DESCRIPTION :

Lexical analysis is the first phase of a compiler, responsible for converting source code into tokens. This phase is automated using Lexical Analyzer Generators like LEX and Flex.

LEX (Lexical Analyzer Generator)

LEX is a tool used for generating lexical analyzers in compiler design. It helps in pattern recognition and tokenising input text using regular expressions. LEX works by defining patterns and corresponding actions in a .l file, which is then processed to generate a C-based scanner.

Key Features of LEX :

- Uses regular expressions to match patterns in input text.
- Generates lex.yy.c, a C program implementing the scanner.
- Can be compiled using a C compiler to produce an executable lexer.
- Works with YACC (Yet Another Compiler Compiler) to build full-fledged compilers.

Working of LEX :

- Specification: The user writes a .l file containing regular expressions and C actions.
- Processing: The lex command processes the .l file and generates lex.yy.c.
- Compilation: The lex.yy.c is compiled with gcc to create an executable scanner.
- Execution: The scanner reads input, matches patterns, and executes the corresponding actions.

Flex (Fast Lexical Analyzer)

Flex is an enhanced and faster version of LEX, designed for improved performance and portability. It follows the same working mechanism as LEX but generates more efficient and optimized C code.

Key Features of Flex :

- Faster and more efficient than LEX.
- Uses longest match rule over first match rule.
- Generates lex.yy.c, similar to LEX but optimized for better performance.
- Works seamlessly on Linux, Unix, and Windows with the required dependencies.

Working of Flex :

- Write a `` file with pattern definitions and C-based actions.
- Use the `` command to generate lex.yy.c.
- Compile the file using gcc.
- Run the executable, which scans the input and processes tokens.

Differences Between LEX and Flex

Feature	***LEX***	***Flex***
Speed	Slower	Faster
Portability	Limited	Widely used in Linux & Unix
Memory Usage	Higher	Optimized
Output File	lex.yy.c	lex.yy.c
Default Action	Returns first match	Returns longest match

Procedure

- Create a .l file (e.g., lexer.l) containing regular expressions and C code.
- Use the flex command to generate lex.yy.c.
- Compile the generated C file using GCC.
- Run the executable and provide input for analysis.

Example Code (LEX/Flex Program)

```
%{
#include <stdio.h>
%}
%%
[0-9]+       { printf("NUMBER\n"); }
[a-zA-Z]+    { printf("IDENTIFIER\n"); }
.            { printf("SPECIAL CHARACTER\n"); }
%%
int main() {
    yylex();
    return 0;
}
```

Conclusion

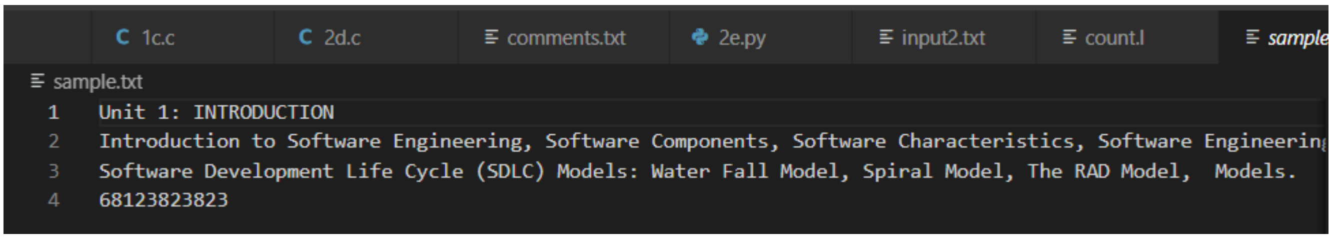
LEX and Flex are powerful tools for lexical analysis in compilers. They help automate tokenisation using regular expressions and C functions, making lexical analysis efficient.

Experiment - 4

Aim : Implement the following using LEX :

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

Code :



1c.c 2d.c comments.txt 2e.py input2.txt count.l sample

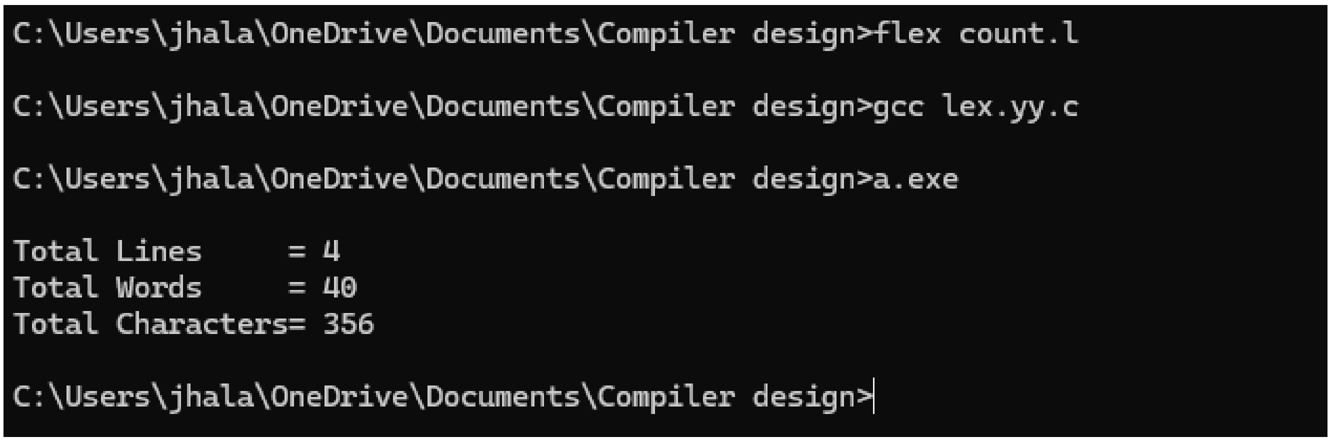
sample.txt

```
Unit 1: INTRODUCTION
Introduction to Software Engineering, Software Components, Software Characteristics, Software Engineerin
Software Development Life Cycle (SDLC) Models: Water Fall Model, Spiral Model, The RAD Model,  Models.
68123823823
```

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

int char_count = 0;
int word_count = 0;
int line_count = 0;
int last_char_was_newline = 0;
%}

%%
\n           { line_count++; char_count++; last_char_was_newline = 1; }
[ \t]+        { char_count += yyleng; last_char_was_newline = 0; }
[a-zA-Z0-9]+    { word_count++; char_count += yyleng; last_char_was_newline = 0; }
.            { char_count++; last_char_was_newline = 0; }
%%

int main() {
    yyin = fopen("sample.txt", "r");
    if (!yyin) {
        perror("Error opening file sample.txt");
        return 1;
    }

    yylex();

    // If the last line doesn't end with \n, count it manually
    if (!last_char_was_newline && char_count > 0)
        line_count++;

    printf("\nTotal Lines     = %d\n", line_count);
    printf("Total Words     = %d\n", word_count);
    printf("Total Characters= %d\n", char_count);

    fclose(yyin);
    return 0;
}

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

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

int char_count = 0;
int word_count = 0;
int line_count = 0;
int last_char_was_newline = 0;
%}

%%
\n            { line_count++; char_count++; last_char_was_newline = 1; }
[ \t]+         { char_count += yyleng; last_char_was_newline = 0; }
[a-zA-Z0-9]+   { word_count++; char_count += yyleng; last_char_was_newline = 0; }
.             { char_count++; last_char_was_newline = 0; }
%%

int main() {
    yyin = fopen("sample.txt", "r");
    if (!yyin) {
        perror("Error opening file sample.txt");
        return 1;
    }

    yylex();

    // If the last line doesn't end with \n, count it manually
    if (!last_char_was_newline && char_count > 0)
        line_count++;

    printf("\nTotal Lines     = %d\n", line_count);
    printf("Total Words     = %d\n", word_count);
    printf("Total Characters= %d\n", char_count);

    fclose(yyin);
    return 0;
}

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

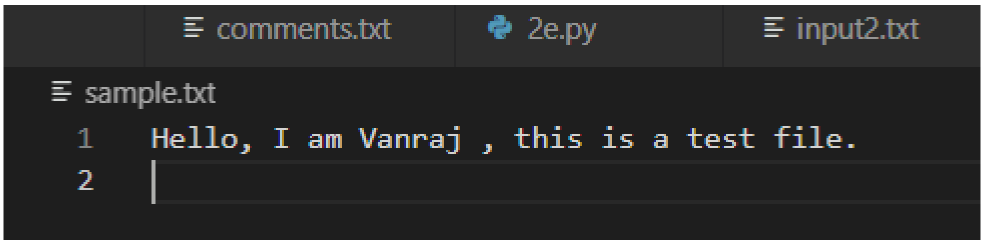
Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>flex count.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe

Total Lines     = 4
Total Words     = 40
Total Characters= 356

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

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

Code :



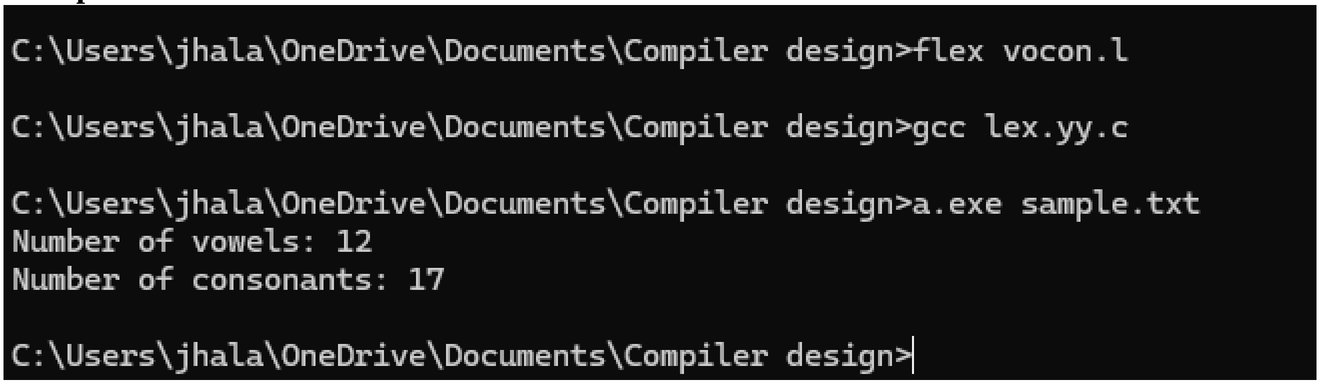
comments.txt | 2e.py | input2.txt

sample.txt

```
Hello, I am Vanraj , this is a test file.

```

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

  int vowels=0;
  int consonants=0;

  FILE *yyin;
  %}

  %%

  [aeiuoAEIOU]  {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 sample.txt \n" , argv[0]);
     return 1;
     }

     FILE*file = fopen(argv[1] , "r");
     if(!file){
        printf("Cant 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;

}
```

Output :



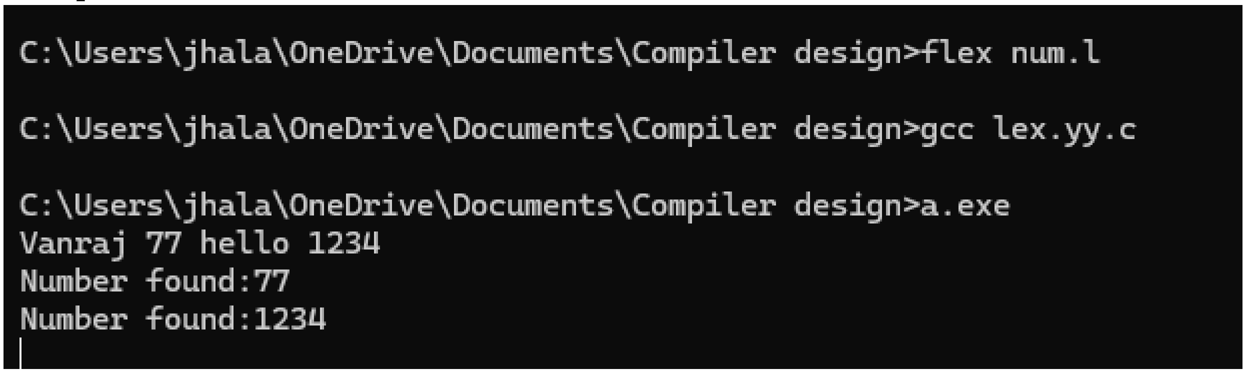
```
C:\Users\jhala\OneDrive\Documents\Compiler design>flex vocon.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe sample.txt
Number of vowels: 12
Number of consonants: 17

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

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

Code :

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

%}

%%

[0-9]+([\.0-9]+)?([Ee][+-]?[0-9]+)?  {printf("Number found:%s \n" , yytext);}

.|\n  { /* Ignore all other characters*/}

%%

int yywrap() {
   return 1;
}

int main(){
   yylex();

   return 0;
}
```

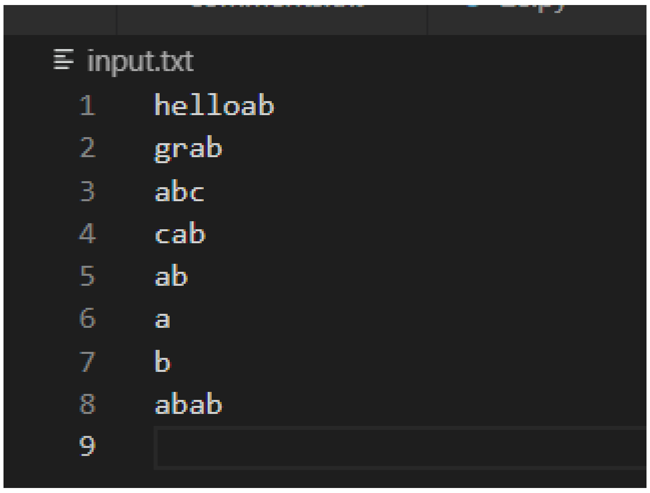
Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>flex num.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
Vanraj 77 hello 1234
Number found:77
Number found:1234
```

D. Write a Lex program which adds line numbers to the given file and display the same into different file.

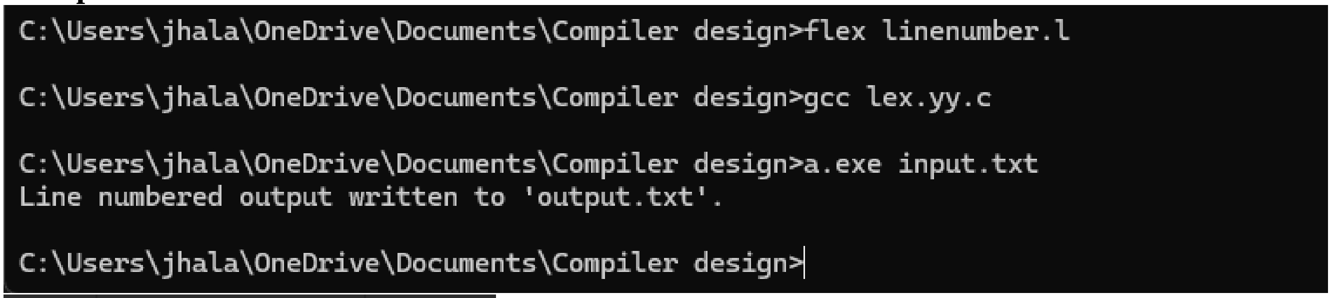
Code :



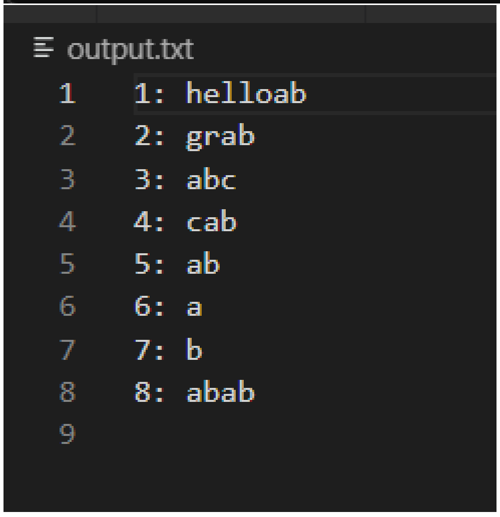
```
input.txt
helloab
grab
abc
cab
ab
a
b
abab
```

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

int line_number = 1;
FILE *yyin, *output;
%}

%%

^.*\n   {
            fprintf(output, "%d: %s", line_number++, yytext);
        }

.       {
            // For lines that don't end in \n (e.g., last line)
            fprintf(output, "%d: %s\n", line_number++, yytext);
        }

%%

int yywrap() {
    return 1;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    yyin = fopen(argv[1], "r");
    if (!yyin) {
        printf("Error: Cannot open input file %s\n", argv[1]);
        return 1;
    }

    output = fopen("output.txt", "w");
    if (!output) {
        printf("Error: Cannot create output file.\n");
        return 1;
    }

    yylex();

    fclose(yyin);
    fclose(output);

    printf("Line numbered output written to 'output.txt'.\n");
    return 0;
}
```

Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>flex linenumber.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe input.txt
Line numbered output written to 'output.txt'.

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

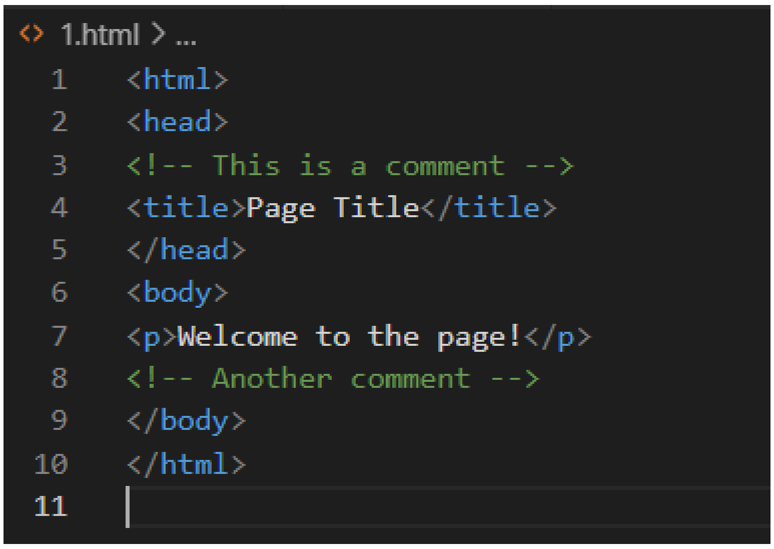


output.txt

```
1: helloab
2: grab
3: abc
4: cab
5: ab
6: a
7: b
8: abab
```

E. Write a Lex program to printout all markup tags and HTML comments in file.

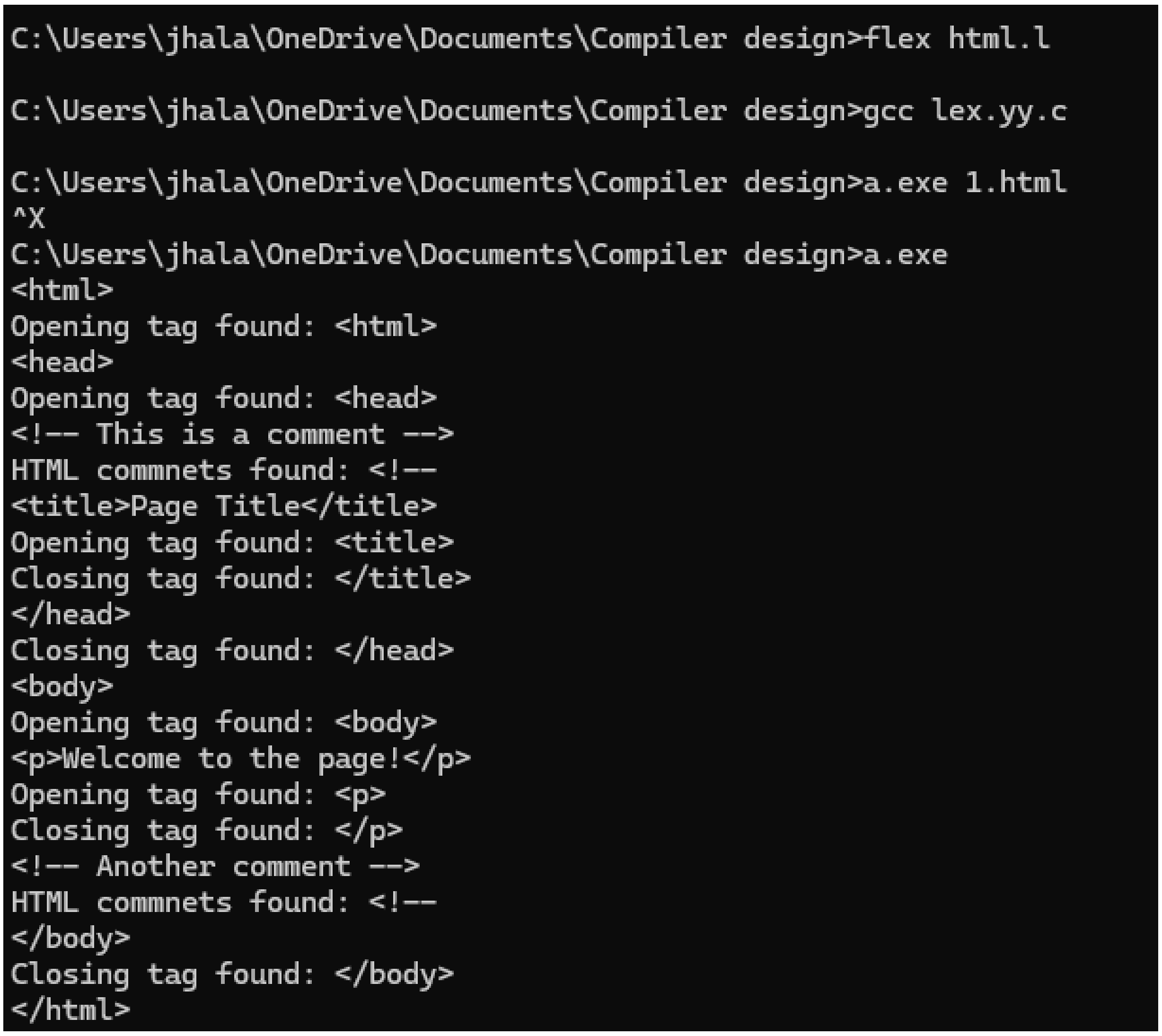
Code :



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

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

%}

%%

"<!--"([^>]|[\n]*"-->")  {printf("HTML commnets 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;
}
```

Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>flex html.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe 1.html
^X
C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
<html>
Opening tag found: <html>
<head>
Opening tag found: <head>
<!-- This is a comment -->
HTML commnets found: <!--
<title>Page Title</title>
Opening tag found: <title>
Closing tag found: </title>
</head>
Closing tag found: </head>
<body>
Opening tag found: <body>
<p>Welcome to the page!</p>
Opening tag found: <p>
Closing tag found: </p>
<!-- Another comment -->
HTML commnets found: <!--
</body>
Closing tag found: </body>
</html>
```

Experiment - 5

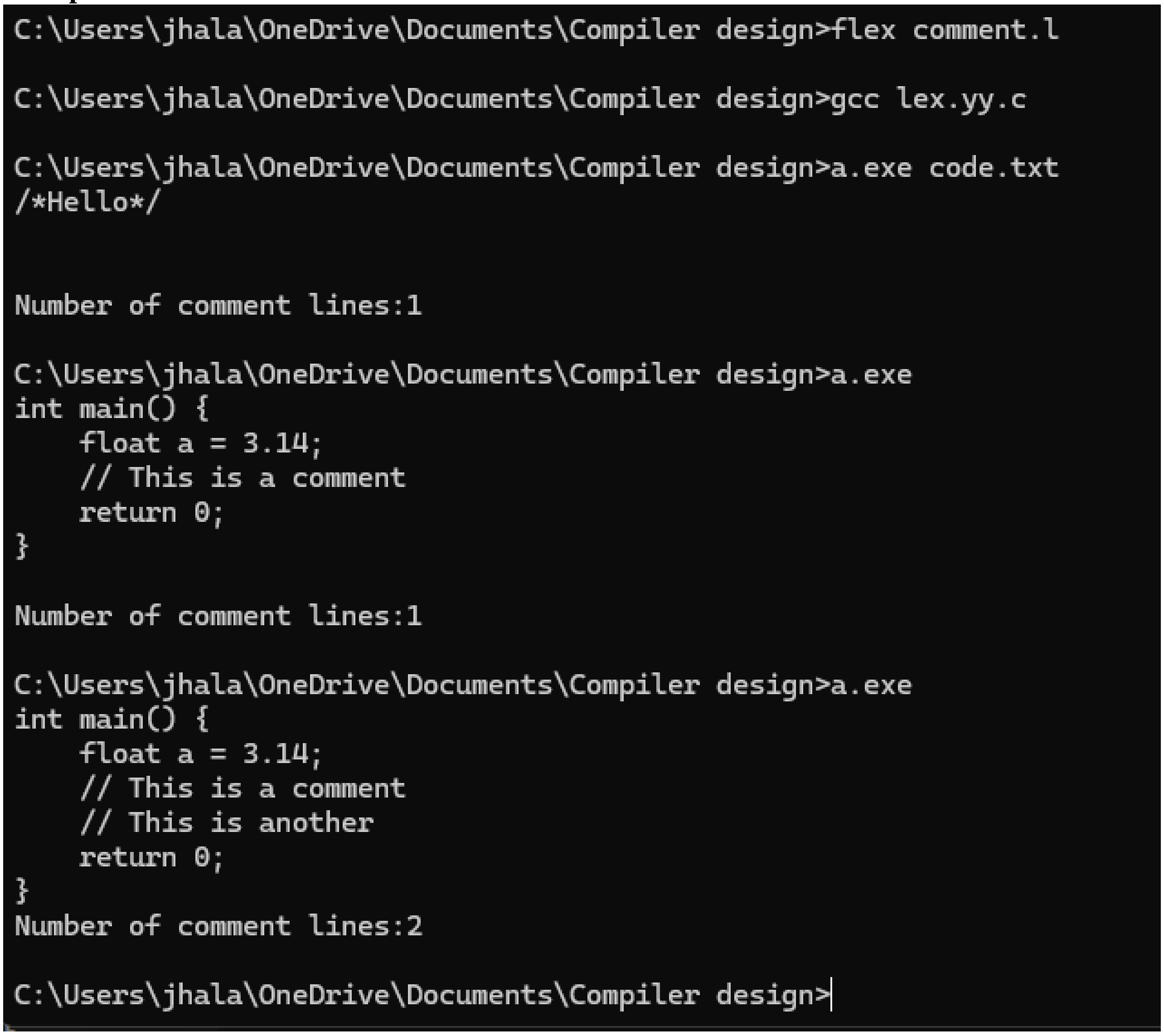
Aim : Perform the following :

A. Write a Lex program to count the number of C comment lines from a given C program.

Code :

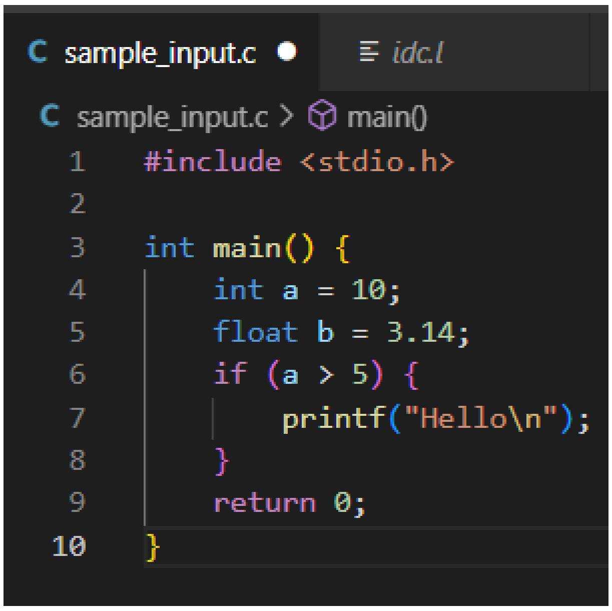
```
%{

#include<stdio.h>
#include<stdlib.h>
int comment_count=0;

%}

%%

\/\/.*  {comment_count++;}

\/\*[^*]*\*+([^/*][^*]*\*+)*\/  {comment_count++;}

.|\n  { /* Ignore all other characters*/}

%%

int yywrap(){
   return 1;
}

int main() {
   yyin = stdin;
   yylex();

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

Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>flex comment.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe code.txt
/*Hello*/


Number of comment lines:1

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
int main() {
    float a = 3.14;
    // This is a comment
    return 0;
}

Number of comment lines:1

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe
int main() {
    float a = 3.14;
    // This is a comment
    // This is another
    return 0;
}
Number of comment lines:2

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

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



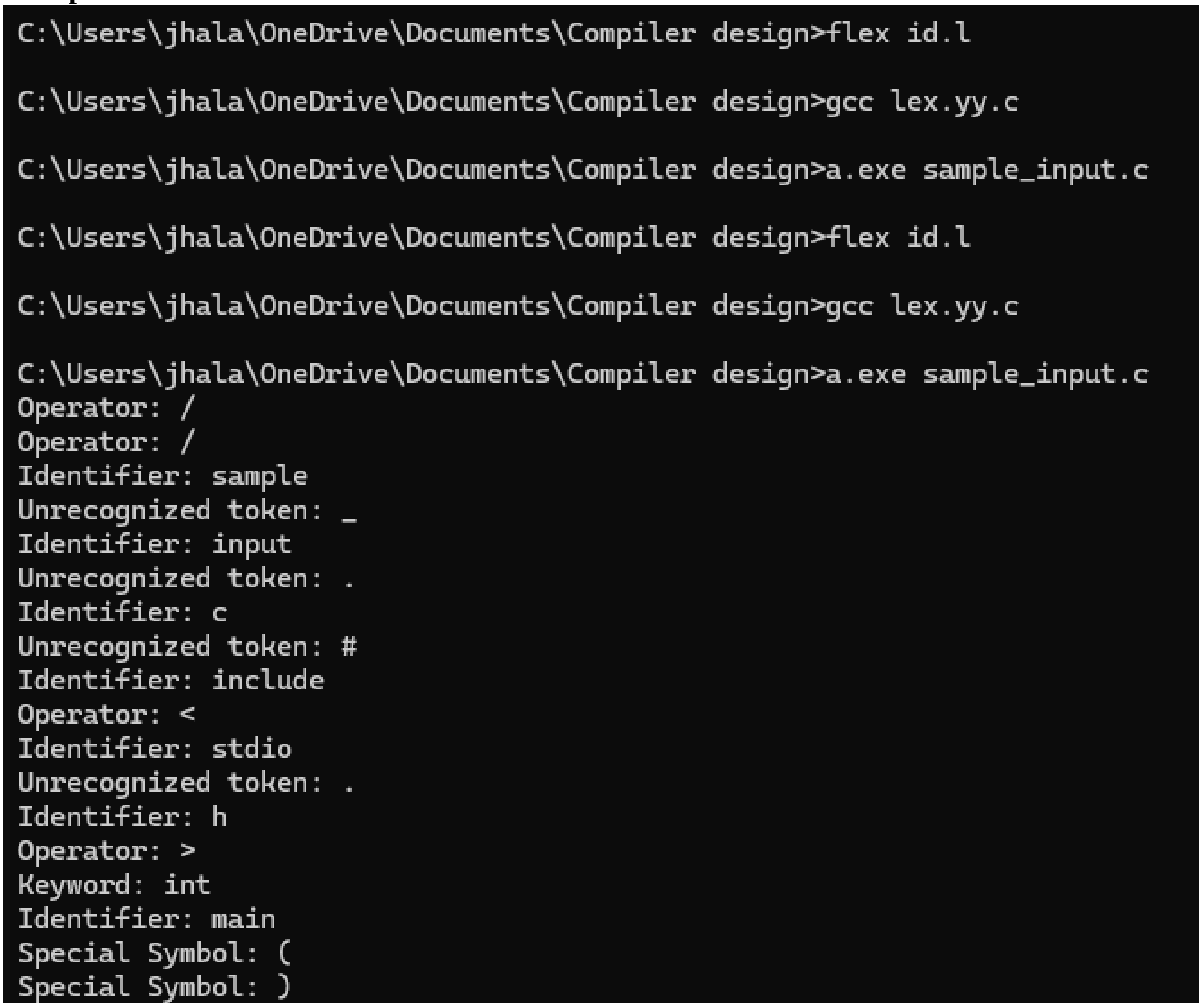
```c
#include <stdio.h>

int main() {
    int a = 10;
    float b = 3.14;
    if (a > 5) {
        printf("Hello\n");
    }
    return 0;
}
```

Code :

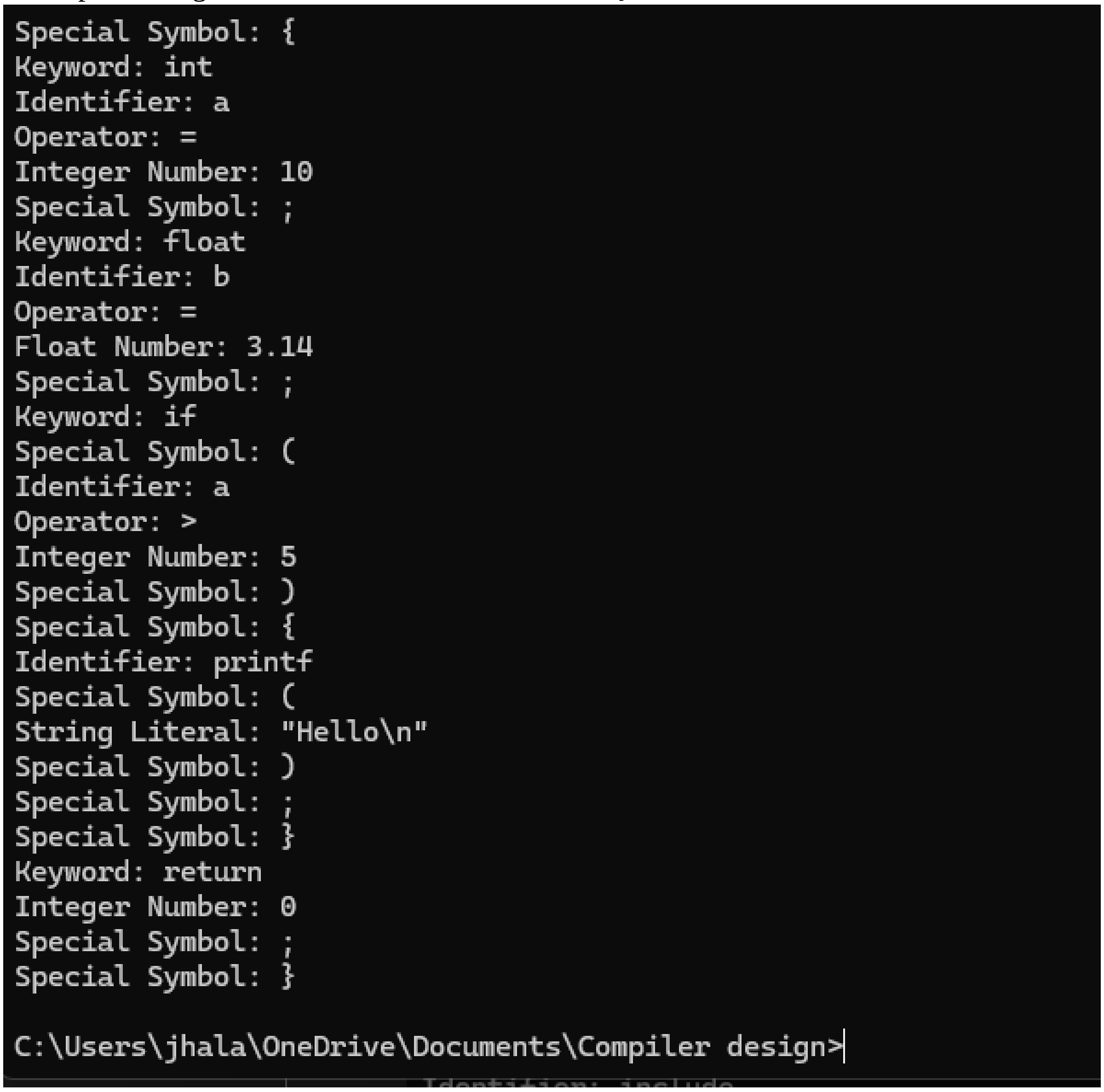
```
%{
#include <stdio.h>
%}

digit        [0-9]
alpha        [a-zA-Z]
id           {alpha}({alpha}|{digit})*
int_const    {digit}+
float_const  {digit}+\.{digit}*([eE][+-]?{digit}+)?
string_lit   \"([^\"\\]|\\.)*\"
char_lit     \'([^\'\\]|\\.)*\'

%%

"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern
"|"float"|"for"|"goto"|"if"|"inline"|"int"|"long"|"register"|"restrict"|"return"|"short"|"signed"|"si
zeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"|"void"|"volatile"|"while" {
  printf("Keyword: %s\n", yytext);
}

{id}           { printf("Identifier: %s\n", yytext); }
"=="|"!="|"<="|">="|"&&"|"||"|"++"|"--"|"+"|"-"|"*"|"/"|"="|"<"|">" {
  printf("Operator: %s\n", yytext);
}
{float_const}     { printf("Float Number: %s\n", yytext); }
{int_const}       { printf("Integer Number: %s\n", yytext); }
{string_lit}      { printf("String Literal: %s\n", yytext); }
{char_lit}        { printf("Character Literal: %s\n", yytext); }
[{}();,]          { printf("Special Symbol: %s\n", yytext); }
[ \t\n]+          ; /* skip whitespace */
.                 { printf("Unrecognized token: %s\n", yytext); }
```

```
%%

int main() {
    yyin = fopen("sample_input.c", "r");
    if (!yyin) {
        perror("Could not open file");
        return 1;
    }
    yylex();
    fclose(yyin);
    return 0;
}

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

Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design>flex id.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe sample_input.c

C:\Users\jhala\OneDrive\Documents\Compiler design>flex id.l

C:\Users\jhala\OneDrive\Documents\Compiler design>gcc lex.yy.c

C:\Users\jhala\OneDrive\Documents\Compiler design>a.exe sample_input.c
Operator: /
Operator: /
Identifier: sample
Unrecognized token: _
Identifier: input
Unrecognized token: .
Identifier: c
Unrecognized token: #
Identifier: include
Operator: <
Identifier: stdio
Unrecognized token: .
Identifier: h
Operator: >
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
```



```
Special Symbol: {
Keyword: int
Identifier: a
Operator: =
Integer Number: 10
Special Symbol: ;
Keyword: float
Identifier: b
Operator: =
Float Number: 3.14
Special Symbol: ;
Keyword: if
Special Symbol: (
Identifier: a
Operator: >
Integer Number: 5
Special Symbol: )
Special Symbol: {
Identifier: printf
Special Symbol: (
String Literal: "Hello\n"
Special Symbol: )
Special Symbol: ;
Special Symbol: }
Keyword: return
Integer Number: 0
Special Symbol: ;
Special Symbol: }

C:\Users\jhala\OneDrive\Documents\Compiler design>
```

Experiment - 6

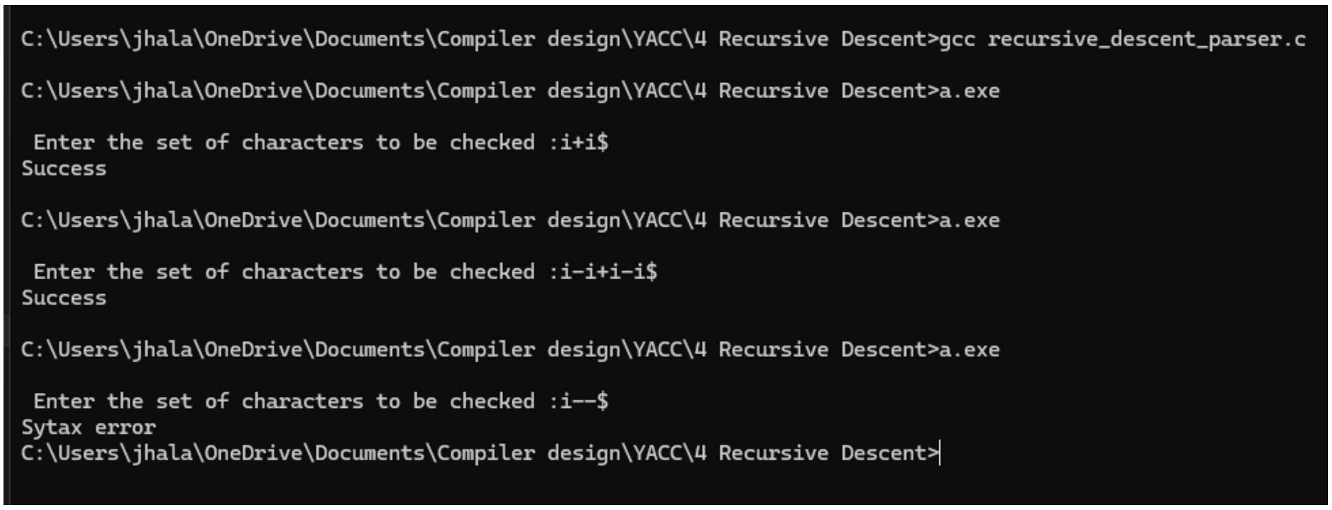
Aim : Program to implement Recursive Descent Parsing in C.

Code :

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

```
#include<stdio.h>
#include<stdlib.h>
/*
E-> iE_
E_-> +iE_ / -iE_ / epsilon
*/
char s[20];
int i=1;
char l;
int match(char t)
{
    if(l==t){
        l=s[i];
        i++; }
    else{
       printf("Sytax error");
       exit(1);}
}
int E_()
{
    if(l=='+'){
        match('+');
        match('i');
        E_(); }
    else  if(l=='-'){
        match('-');
        match('i');
        E_(); }
    else
        return(1);
}
int E()
{
    if(l=='i'){
        match('i');
        E_(); }
}

int main()
{
    printf("\n Enter the set of characters to be checked :");
    scanf("%s",&s);
    l=s[0];
    E();
    if(l=='$')
    {
        printf("Success \n");
    }
    else{
        printf("syntax error");
    }
    return 0;
}
```

Output :



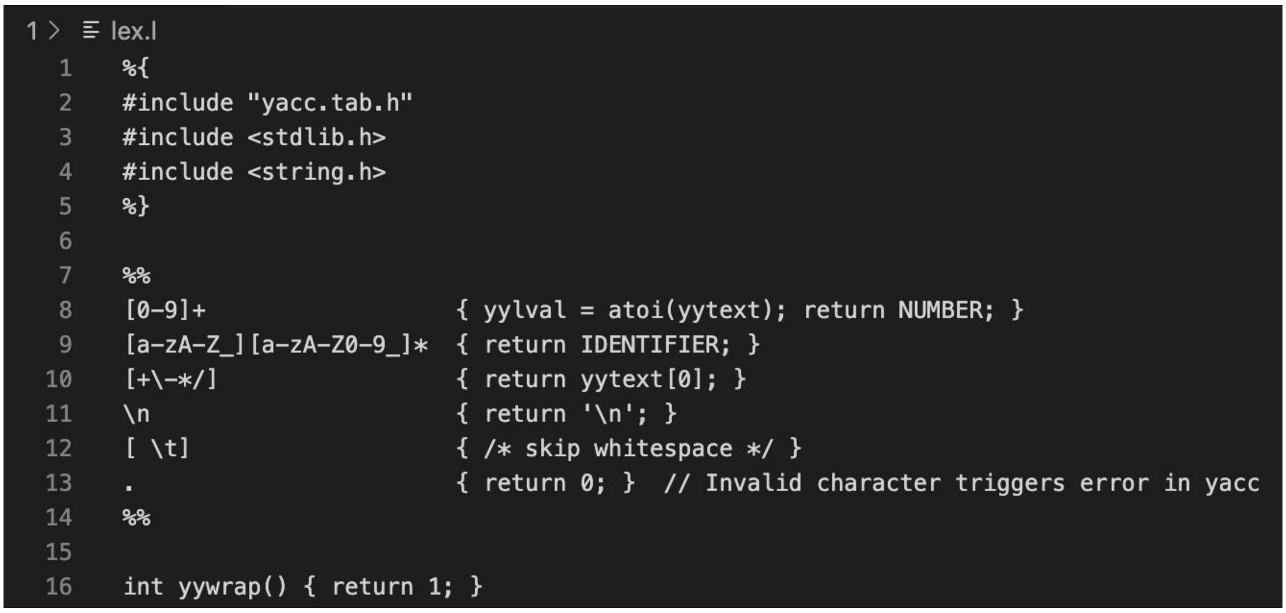
```
C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\4 Recursive Descent>gcc recursive_descent_parser.c

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\4 Recursive Descent>a.exe

 Enter the set of characters to be checked :i+i$
Success

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\4 Recursive Descent>a.exe

 Enter the set of characters to be checked :i-i+i-i$
Success

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\4 Recursive Descent>a.exe

 Enter the set of characters to be checked :i--$
Sytax error
C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\4 Recursive Descent>
```

Experiment - 7

Aim : Perform the following :

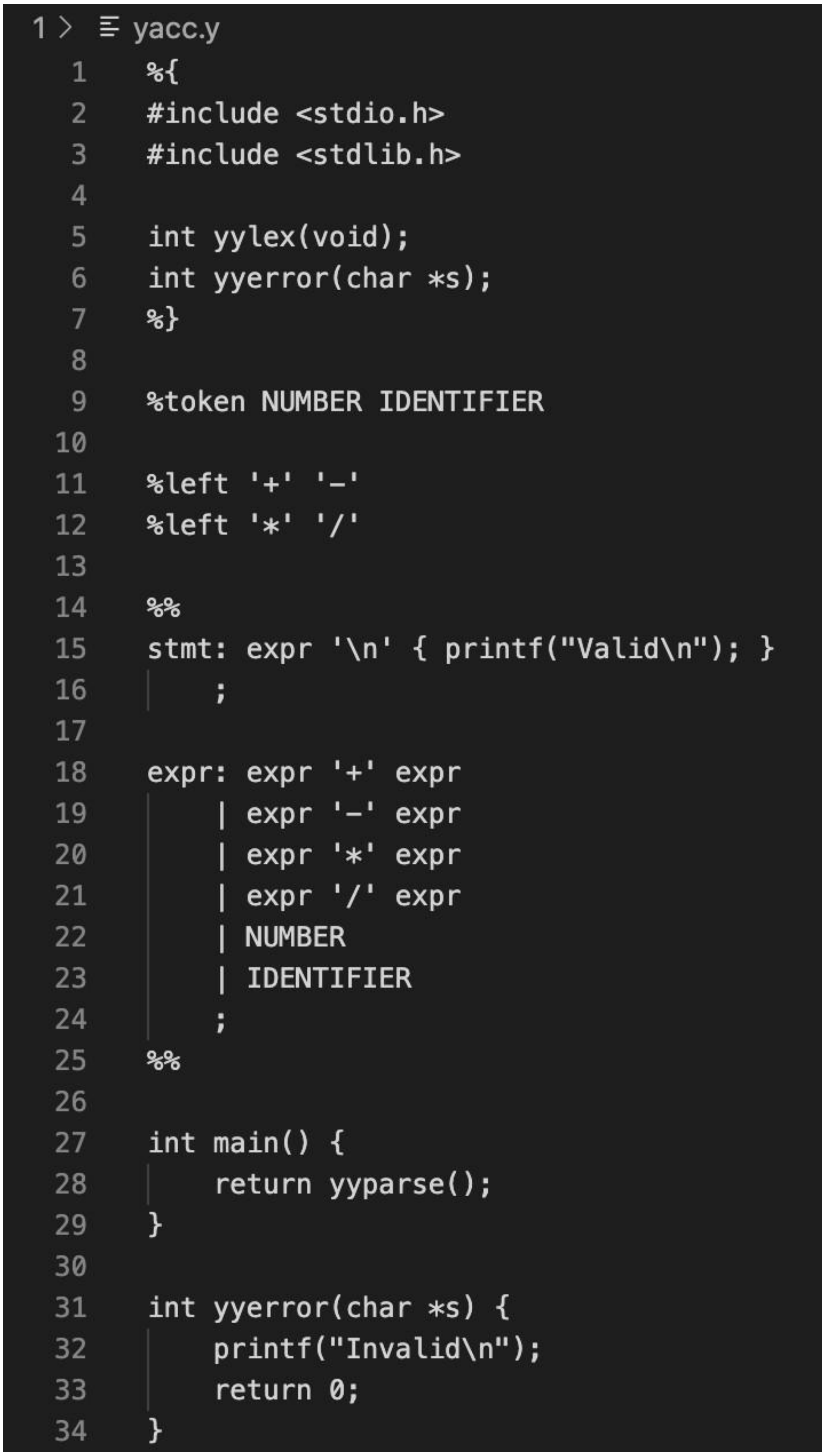
A. Create Yacc and Lex specification files to recognises arithmetic expressions involving +, -, * and / .

Code :



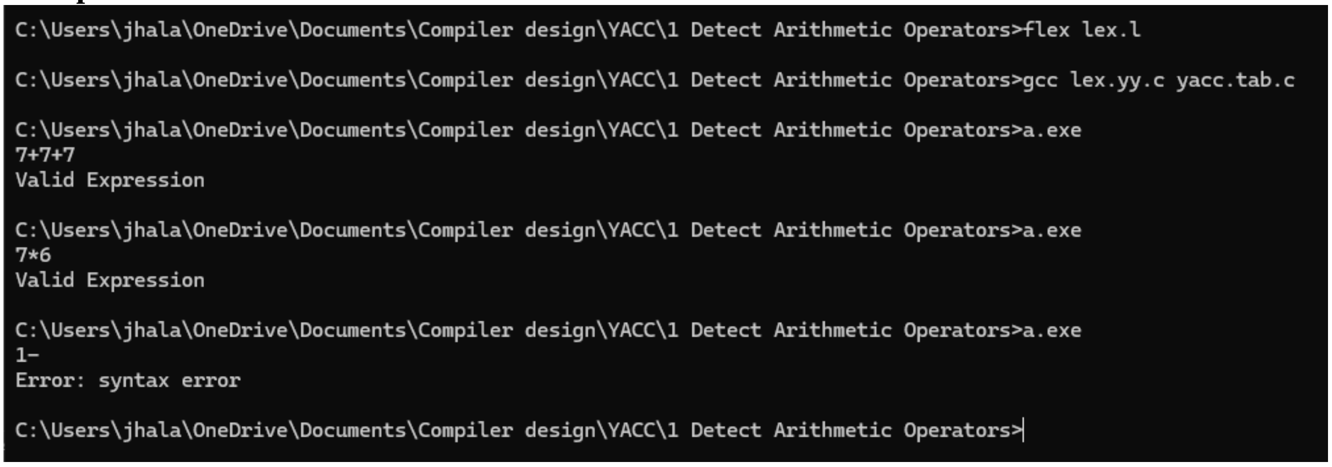
1 > lex.l

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

%%
[0-9]+                  { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { return IDENTIFIER; }
[+\-*/]                 { return yytext[0]; }
\n                      { return '\n'; }
[ \t]                   { /* skip whitespace */ }
.                       { return 0; }  // Invalid character triggers error in yacc
%%

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


1 > yacc.y

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

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

%token NUMBER IDENTIFIER

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

%%
stmt: expr '\n' { printf("Valid\n"); }
    ;

expr: expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NUMBER
    | IDENTIFIER
    ;
%%

int main() {
    return yyparse();
}

int yyerror(char *s) {
    printf("Invalid\n");
    return 0;
}
```

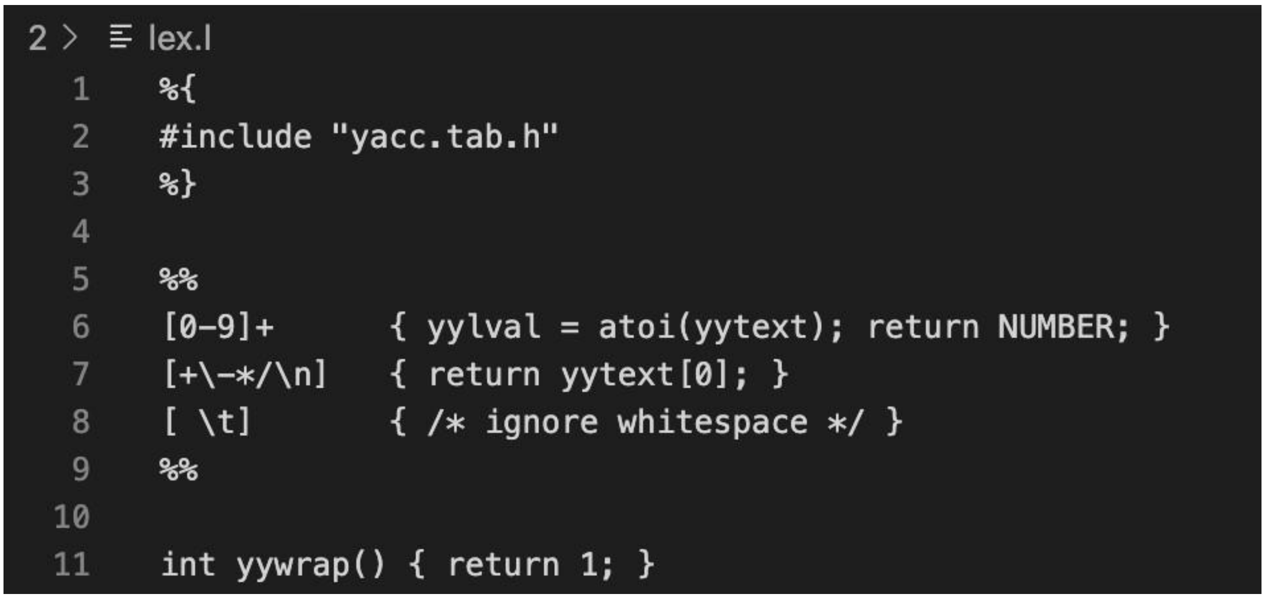
Output :



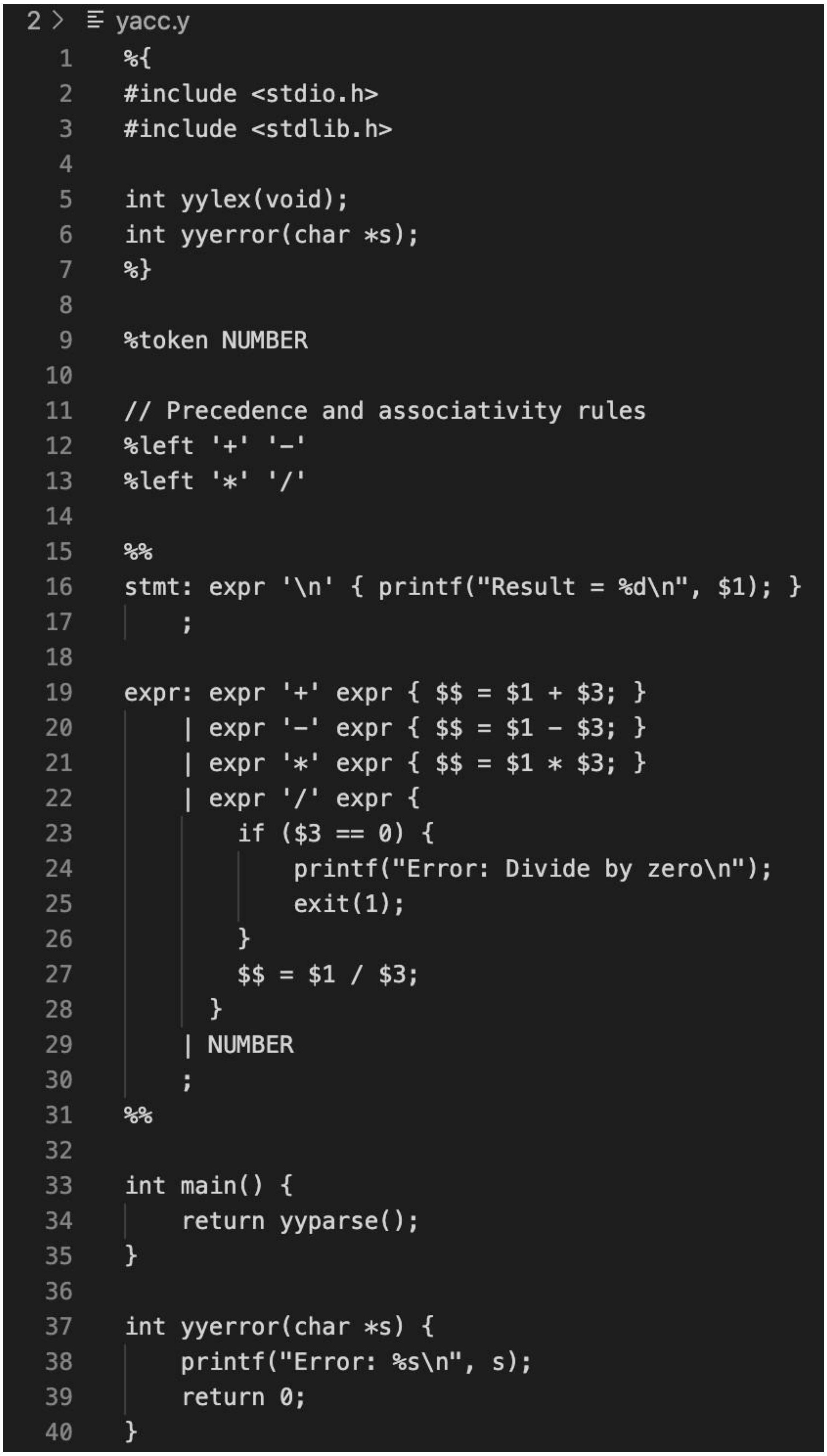
```
C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\1 Detect Arithmetic Operators>flex lex.l

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\1 Detect Arithmetic Operators>gcc lex.yy.c yacc.tab.c

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\1 Detect Arithmetic Operators>a.exe
7+7+7
Valid Expression

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\1 Detect Arithmetic Operators>a.exe
7*6
Valid Expression

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\1 Detect Arithmetic Operators>a.exe
1-
Error: syntax error

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\1 Detect Arithmetic Operators>
```

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

Code :

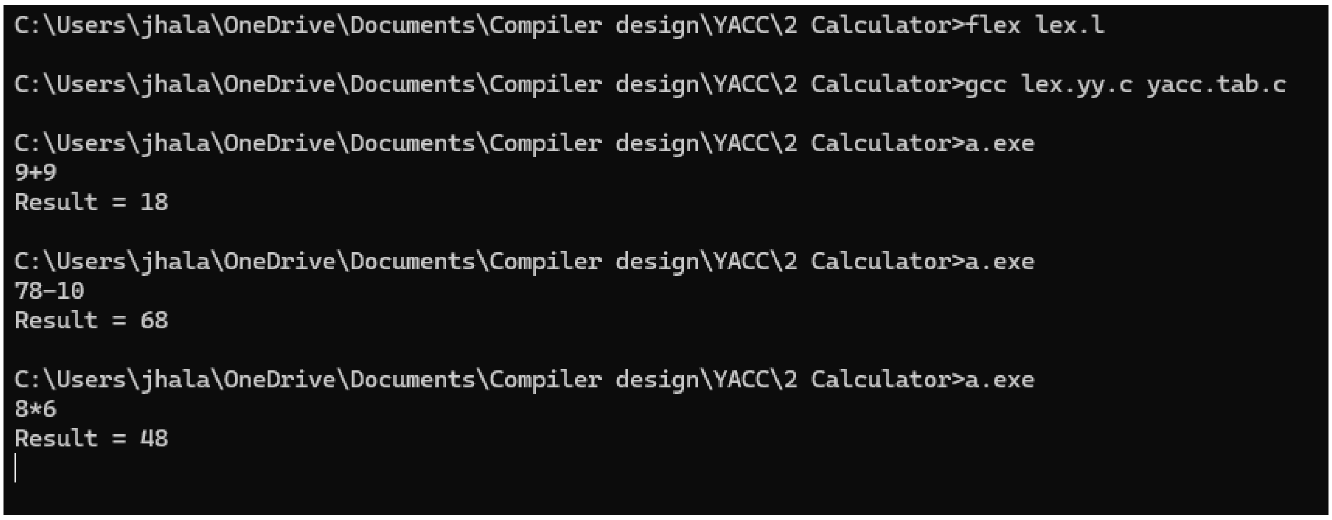

2 > lex.l

```
%{
#include "yacc.tab.h"
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[+\-*/\n]   { return yytext[0]; }
[ \t]       { /* ignore whitespace */ }
%%

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



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

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

%token NUMBER

// Precedence and associativity rules
%left '+' '-'
%left '*' '/'

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

expr: expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr {
        if ($3 == 0) {
            printf("Error: Divide by zero\n");
            exit(1);
        }
        $$ = $1 / $3;
      }
    | NUMBER
    ;
%%

int main() {
    return yyparse();
}

int yyerror(char *s) {
    printf("Error: %s\n", s);
    return 0;
}
```

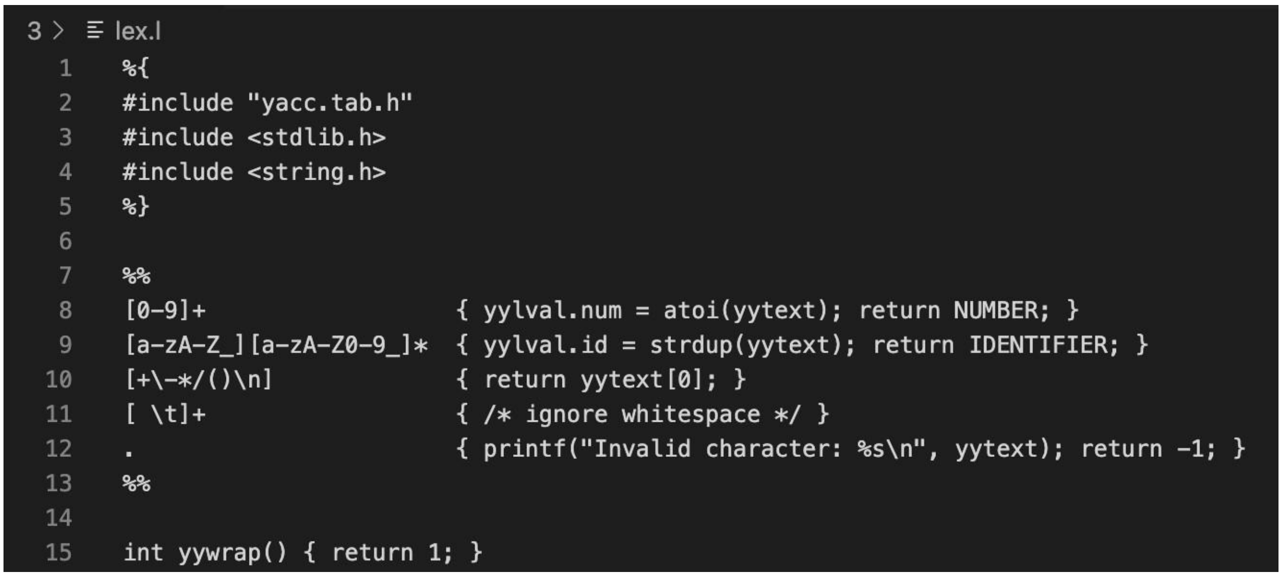
Output :



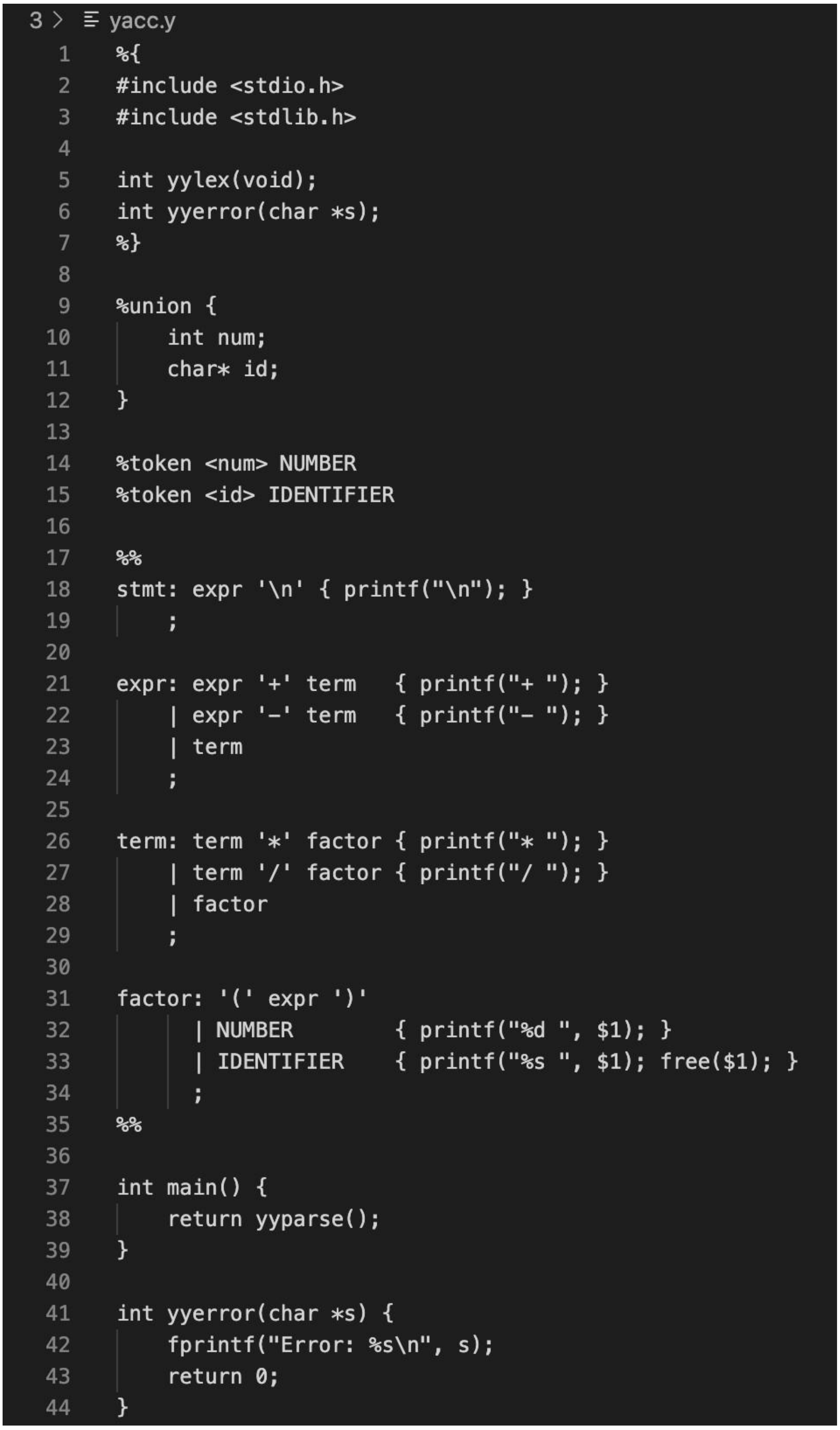
```
C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\2 Calculator>flex lex.l

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\2 Calculator>gcc lex.yy.c yacc.tab.c

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\2 Calculator>a.exe
9+9
Result = 18

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\2 Calculator>a.exe
78-10
Result = 68

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\2 Calculator>a.exe
8*6
Result = 48
```

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

Code :



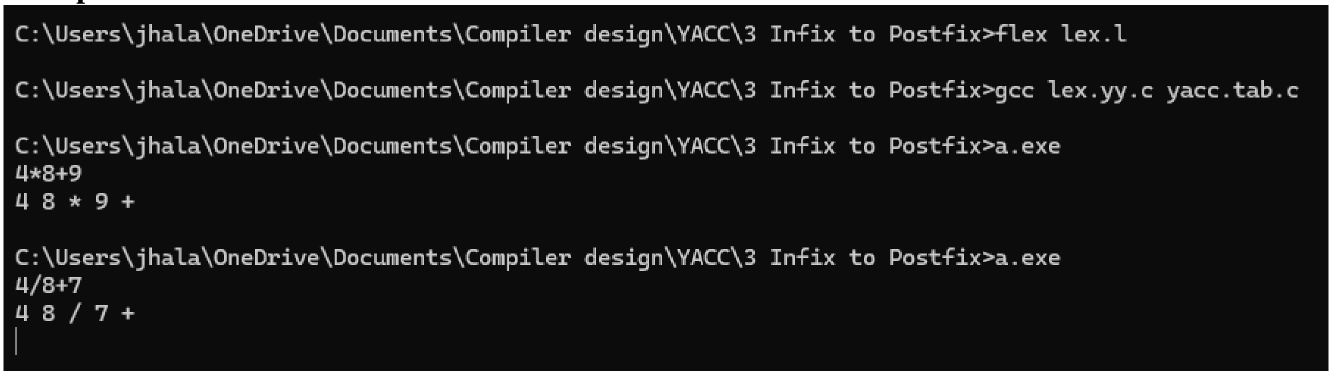
```
3 > lex.l
%{
#include "yacc.tab.h"
#include <stdlib.h>
#include <string.h>
%}

%%
[0-9]+                  { yylval.num = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { yylval.id = strdup(yytext); return IDENTIFIER; }
[+\-*/()\n]             { return yytext[0]; }
[ \t]+                  { /* ignore whitespace */ }
.                       { printf("Invalid character: %s\n", yytext); return -1; }
%%

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



3 > yacc.y

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

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

%union {
    int num;
    char* id;
}

%token <num> NUMBER
%token <id> IDENTIFIER

%%
stmt: expr '\n' { printf("\n"); }
    ;

expr: expr '+' term   { printf("+ "); }
    | expr '-' term   { printf("- "); }
    | term
    ;

term: term '*' factor { printf("* "); }
    | term '/' factor { printf("/ "); }
    | factor
    ;

factor: '(' expr ')'
      | NUMBER        { printf("%d ", $1); }
      | IDENTIFIER    { printf("%s ", $1); free($1); }
      ;
%%

int main() {
    return yyparse();
}

int yyerror(char *s) {
    fprintf("Error: %s\n", s);
    return 0;
}
```

Output :



```
C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\3 Infix to Postfix>flex lex.l

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\3 Infix to Postfix>gcc lex.yy.c yacc.tab.c

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\3 Infix to Postfix>a.exe
4*8+9
4 8 * 9 +

C:\Users\jhala\OneDrive\Documents\Compiler design\YACC\3 Infix to Postfix>a.exe
4/8+7
4 8 / 7 +
```