LAB MANUAL
of
Compiler Design Laboratory (CS605)

Bachelor of Technology (CSE)

By
Sahil Agrawal (22000927)
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
(2025-2026)

TABLE OF CONTENT

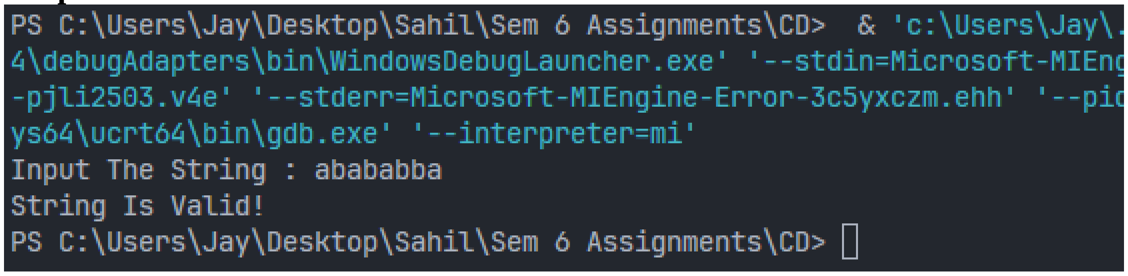
1) A) Write a program to recognize strings starts with ‘a’ over {a, b}.

Source Code:

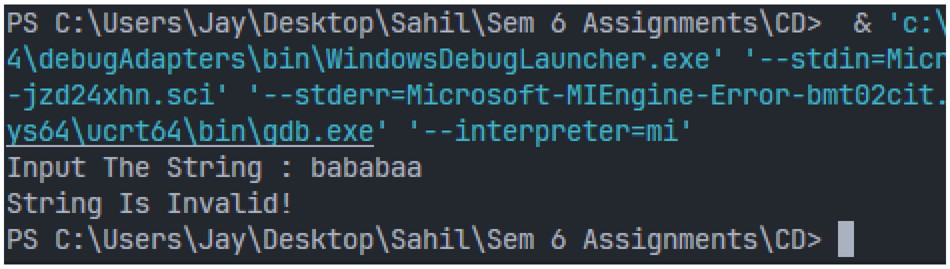
```
#include <stdio.h>

int main(){
   char input[10];
   int i=0, state=0;
   printf("Input 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 = 2;
               else state = 3;
               break;

           case 1:
               if (input[i] == 'a' || input[i] == 'b') state = 1;
               else state = 3;
               break;

           case 2:
               if (input[i] == 'a' || input[i] == 'b') state = 2;
               else state = 3;
               break;

           case 3:
               state = 3;
               break;
       }

       i++;
   }
```

```c
   if (state==1) printf("String Is Valid!");
   else if (state==2 || state==0) {printf("String Is Invalid!");}
   else if (state==3) printf("String Is Not Recognised!");
   return 0;
}
```

Output:



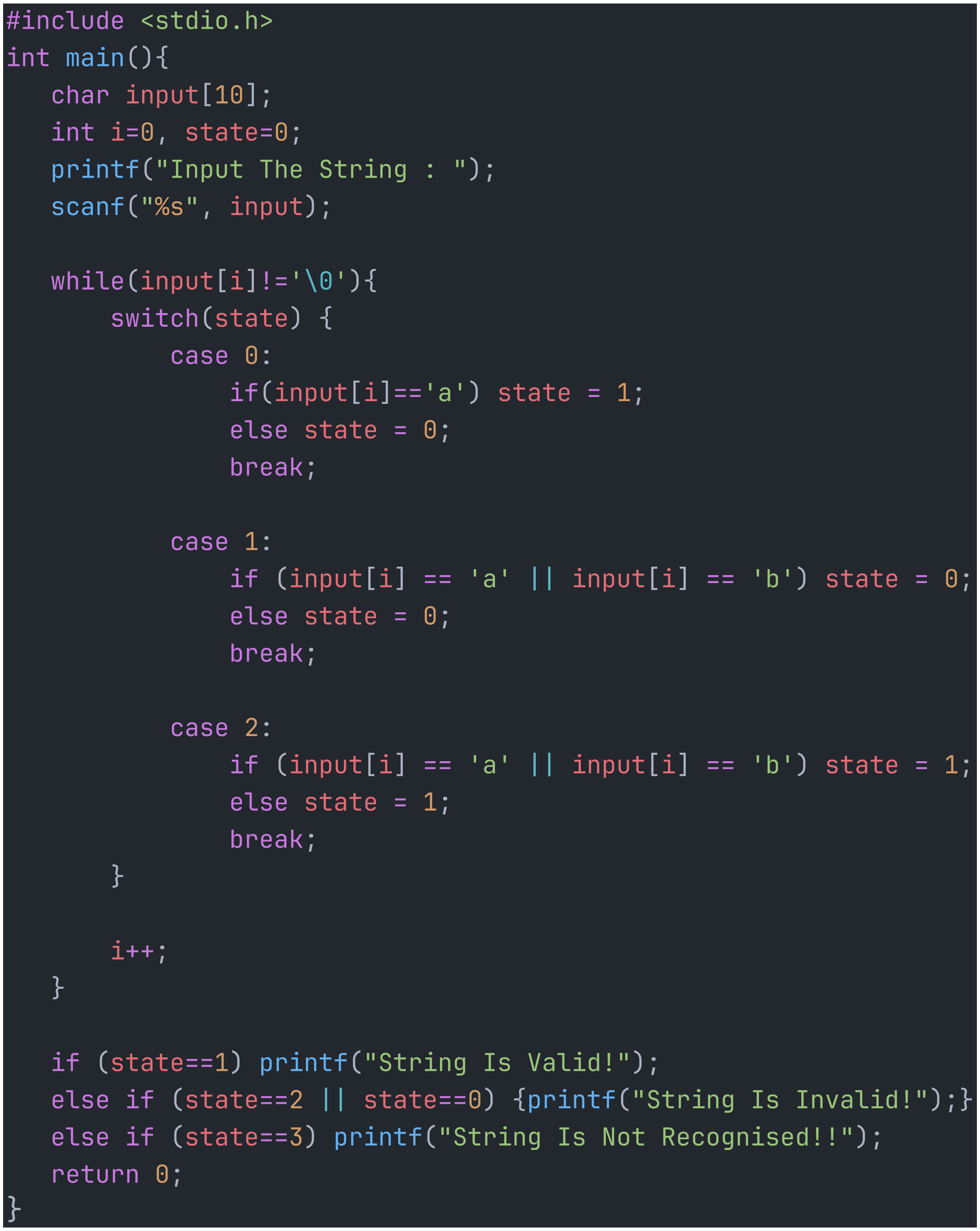
```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & 'c:\Users\Jay\.
4\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEng
-pjli2503.v4e' '--stderr=Microsoft-MIEngine-Error-3c5yxczm.ehh' '--pic
ys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Input The String : abababba
String Is Valid!
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD> 
```



```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & 'c:\
4\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Micr
-jzd24xhn.sci' '--stderr=Microsoft-MIEngine-Error-bmt02cit.
ys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Input The String : babababaa
String Is Invalid!
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD> 
```

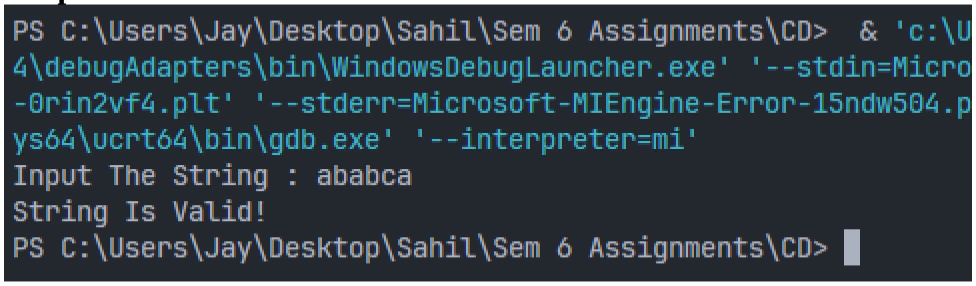
1) B) Write a program to recognize strings end with ‘a’.

Source Code:

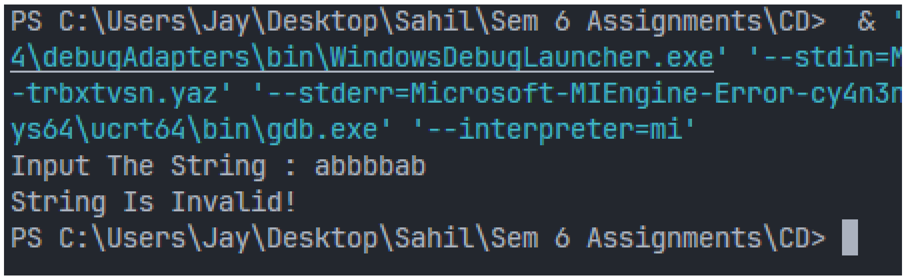


```c
#include <stdio.h>
int main(){
   char input[10];
   int i=0, state=0;
   printf("Input The String : ");
   scanf("%s", input);

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

           case 1:
               if (input[i] == 'a' || input[i] == 'b') state = 0;
               else state = 0;
               break;

           case 2:
               if (input[i] == 'a' || input[i] == 'b') state = 1;
               else state = 1;
               break;
       }

       i++;
   }

   if (state==1) printf("String Is Valid!");
   else if (state==2 || state==0) {printf("String Is Invalid!");}
   else if (state==3) printf("String Is Not Recognised!!");
   return 0;
}
```

Output:



```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & 'c:\U
4\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Micro
-0rin2vf4.plt' '--stderr=Microsoft-MIEngine-Error-15ndw504.p
ys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Input The String : ababca
String Is Valid!
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```



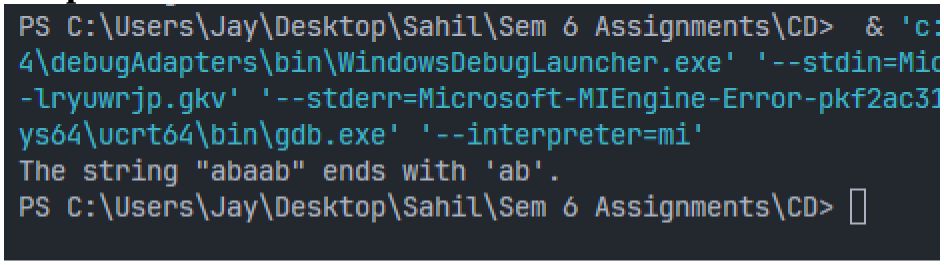
```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & '
4\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=M
-trbxtvsn.yaz' '--stderr=Microsoft-MIEngine-Error-cy4n3n
ys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Input The String : abbbbab
String Is Invalid!
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

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

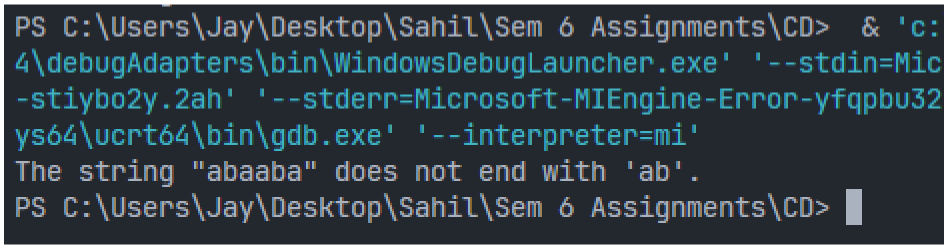
Source Code:

```c
#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: Could not open file.\n");
        return 1;
    }

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

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

                case 1:
                    if (input[i] == 'b') state = 2;
                    else state = 0;
                    break;

                case 2:
                    state = 0;
                    break;
            }
            i++;
        }

        if (state == 2) 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:



```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & 'c:
4\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Mic
-lryuwrjp.gkv' '--stderr=Microsoft-MIEngine-Error-pkf2ac31
ys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
The string "abaab" ends with 'ab'.
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD> 
```



```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & 'c:
4\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Mic
-stiybo2y.2ah' '--stderr=Microsoft-MIEngine-Error-yfqpbu32
ys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
The string "abaaba" does not end with 'ab'.
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD> 
```

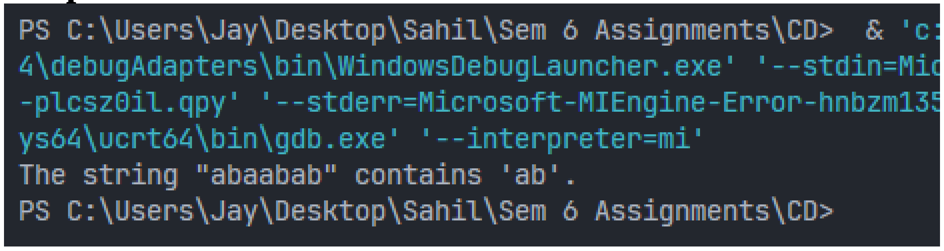
1) D) Write a program to recognize strings contains ‘ab’. Take the input from text file.

Source 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: Could not open file.\n");
        return 1;
    }

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

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

                case 1:
                    if (input[i] == 'b') state = 2;
                    else state = 0;
                    break;

                case 2:
                    state = 2;
```

```c
                    break;
            }
            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:

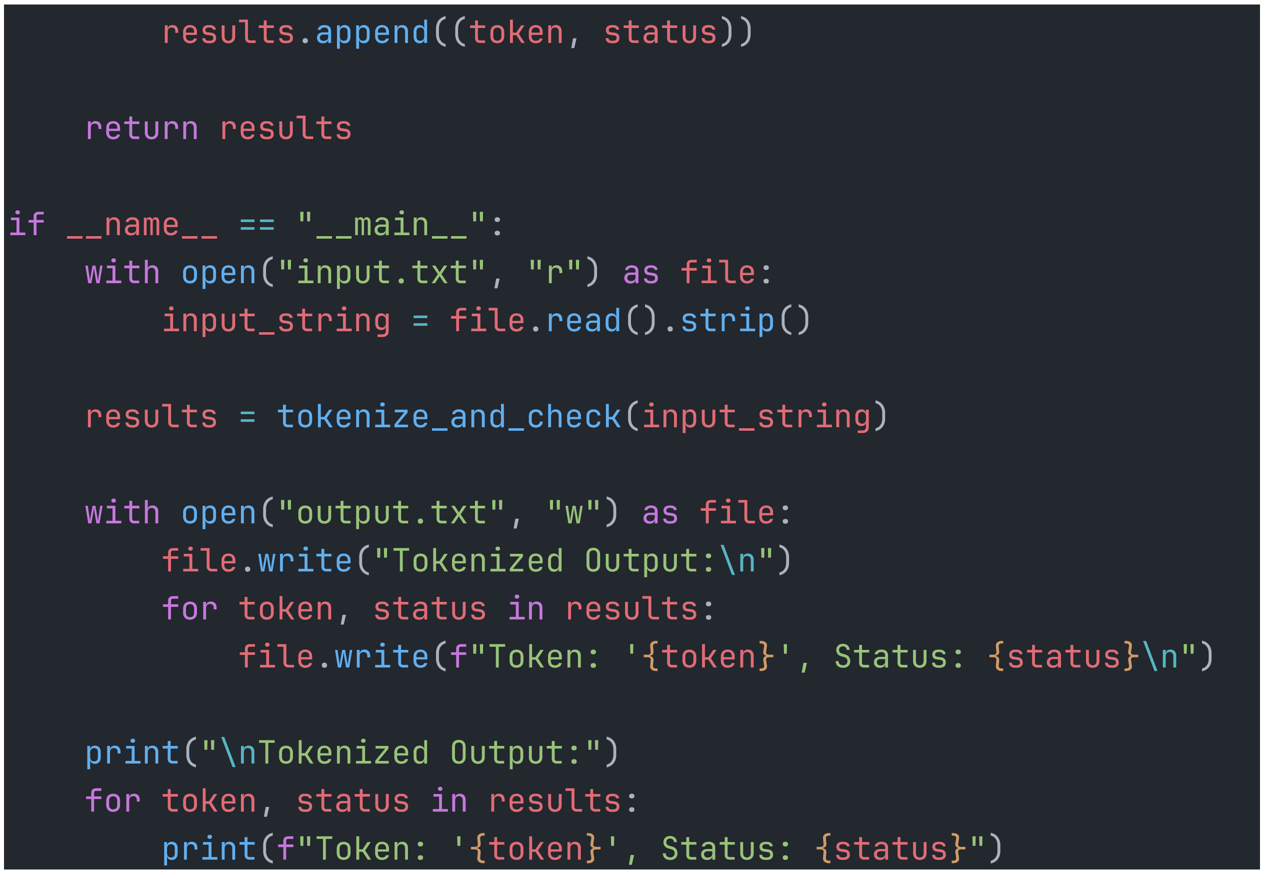


```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & 'c:
4\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Mic
-plcsz0il.qpy' '--stderr=Microsoft-MIEngine-Error-hnbzm135
ys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
The string "abaabab" contains 'ab'.
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

2) A) Write a program to recognize the valid identifiers and keywords.

Source Code:

```
def is_valid_identifier(token):
    state = 0
    for char in token:
        if state == 0:
            if char.isalpha() or char == '_':
                state = 1
            else:
                return False
        elif state == 1:
            if char.isalnum() or char == '_':
                state = 1
            else:
                return False
    return state == 1

def is_keyword(token):
    keywords = {"if", "else", "while", "return", "for", "def",
"class", "import", "from", "as", "with", "try", "except",
"finally", "raise", "lambda", "pass", "break", "continue", "in",
"not", "or", "and", "is", "None", "True", "False", "global",
"nonlocal", "assert", "yield"}
    return token in keywords

def tokenize_and_check(input_string):
    tokens = input_string.split()
    results = []
    for token in tokens:
        identifier = is_valid_identifier(token)
        keyword_check = is_keyword(token)
        status = "Both Identifier and Keyword" if identifier and
keyword_check else \
                 "Valid Identifier" if identifier else \
                 "Keyword" if keyword_check else "Invalid"
```

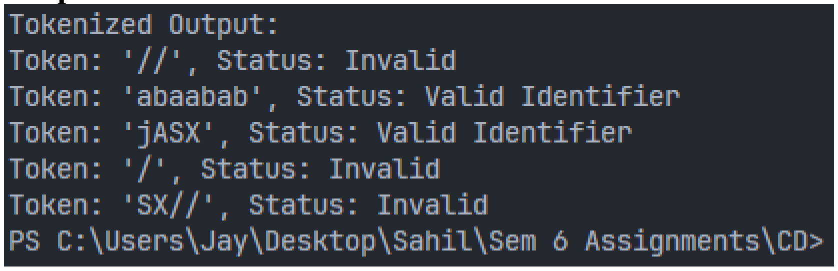


```
        results.append((token, status))

    return results

if __name__ == "__main__":
    with open("input.txt", "r") as file:
        input_string = file.read().strip()

    results = tokenize_and_check(input_string)

    with open("output.txt", "w") as file:
        file.write("Tokenized Output:\n")
        for token, status in results:
            file.write(f"Token: '{token}', Status: {status}\n")

    print("\nTokenized Output:")
    for token, status in results:
        print(f"Token: '{token}', Status: {status}")
```

input.txt file

// abaabab jASX / SX//

Output:



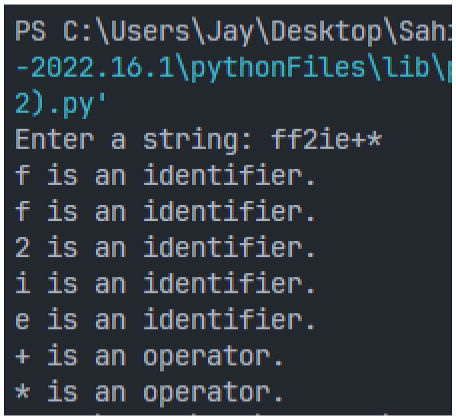
```
Tokenized Output:
Token: '//', Status: Invalid
Token: 'abaabab', Status: Valid Identifier
Token: 'jASX', Status: Valid Identifier
Token: '/', Status: Invalid
Token: 'SX//', Status: Invalid
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

2) B) Write a program to recognize the valid operators.

Source Code:

```python
operators = {'+', '-', '*', '/', '%', '=', '==', '!=', '>', '<',
'>=', '<='}

def is_identifier(token):
    return token.isalnum() and not token.isdigit()

expression = input("Enter a string: ")

for char in expression:
    if char in operators:
        print(f"{char} is an operator.")
    elif char.isalnum():
        print(f"{char} is an identifier.")
```

Output:



```
PS C:\Users\Jay\Desktop\Sah
-2022.16.1\pythonFiles\lib\
2).py'
Enter a string: ff2ie+*
f is an identifier.
f is an identifier.
2 is an identifier.
i is an identifier.
e is an identifier.
+ is an operator.
* is an operator.
```

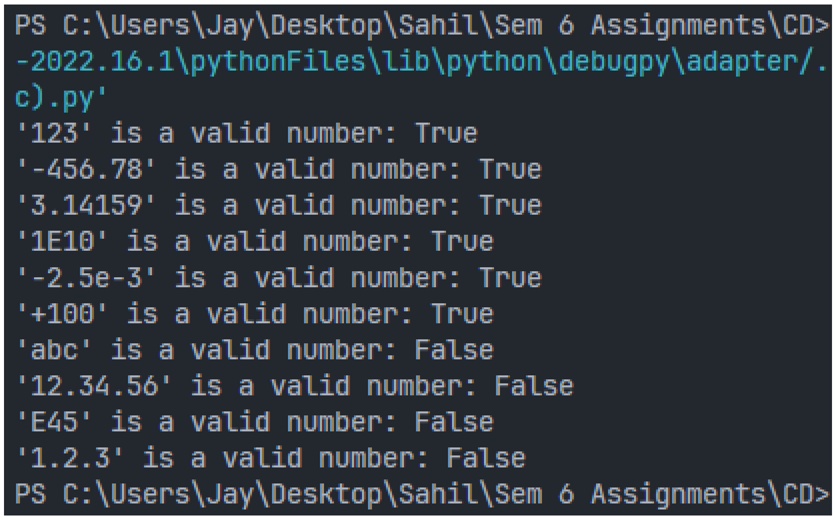
2) C) Write a program to recognize a valid number.

Numbers. Txt
123
-456.78
3.14159
1E10
-2.5e-3
+100
abc
12.34.56
E45
1.2.3

Source Code:

```python
def is_valid_number(s):
    state = 'a'
    for char in s:
        if state == 'a':
            if char in '+-':
                state = 'h'
            elif char.isdigit():
                state = 'b'
            else:
                return False
        elif state == 'h':
            if char.isdigit():
                state = 'b'
            else:
                return False
        elif state == 'b':
            if char.isdigit():
                state = 'b'
            elif char == '.':
                state = 'c'
            elif char in 'Ee':
                state = 'e'
            else:
                return False
        elif state == 'c':
            if char.isdigit():
                state = 'd'
            else:
                return False
        elif state == 'd':
            if char.isdigit():
                state = 'd'
            elif char in 'Ee':
                state = 'e'
            else:
                return False
        elif state == 'e':
            if char in '+-':
                state = 'f'
            elif char.isdigit():
                state = 'g'
            else:
                return False
        elif state == 'f':
            if char.isdigit():
                state = 'g'
            else:
                return False
        elif state == 'g':
            if char.isdigit():
                state = 'g'
            else:
                return False
    return state in {'b', 'd', 'g'}

try:
    with open("numbers.txt", "r") as file:
        for line in file:
            number = line.strip()
            print(f"'{number}' is a valid number: {is_valid_number(number)}")
except FileNotFoundError:
    print("Error: 'numbers.txt' file not found.")
```

Output:



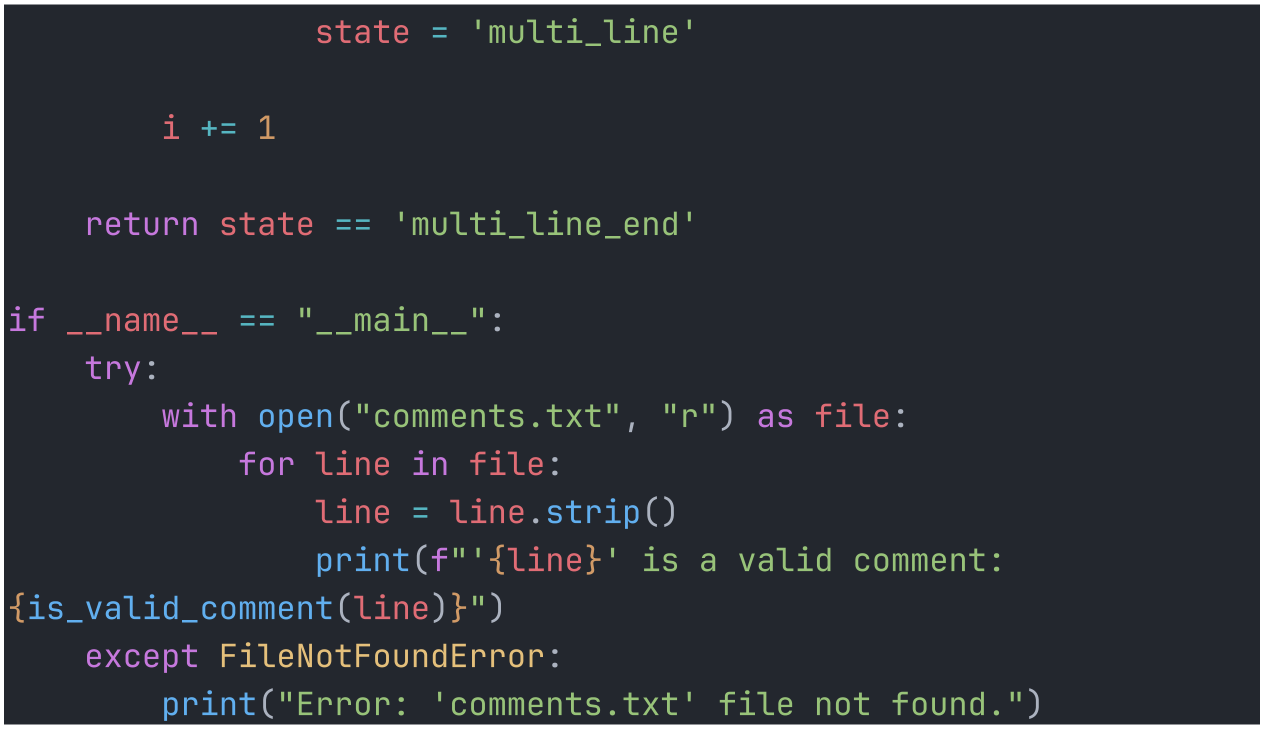
```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
-2022.16.1\pythonFiles\lib\python\debugpy\adapter/.
c).py'
'123' is a valid number: True
'-456.78' is a valid number: True
'3.14159' is a valid number: True
'1E10' is a valid number: True
'-2.5e-3' is a valid number: True
'+100' is a valid number: True
'abc' is a valid number: False
'12.34.56' is a valid number: False
'E45' is a valid number: False
'1.2.3' is a valid number: False
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

2) D) Write a program to recognize valid comments.

comments.txt
Hello World
// This is a single-line comment
/* This is a multi-line comment */
Not a comment
/* Unclosed comment

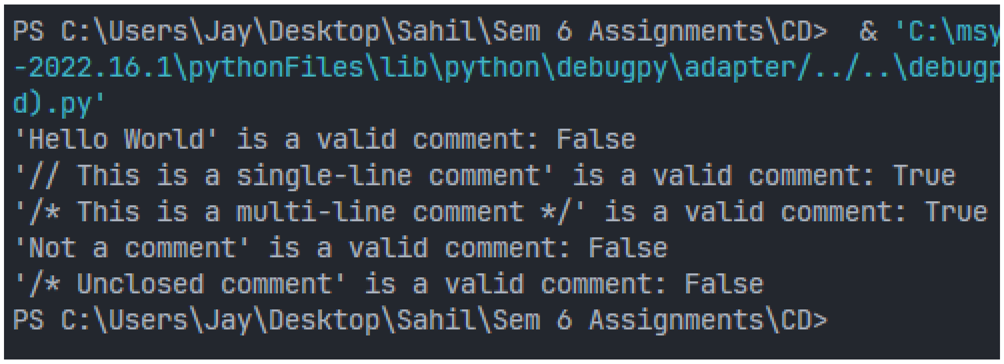
Source Code:

```
def is_valid_comment(line: str) -> bool:
    state = 'start'
    i = 0
    while i < len(line):
        char = line[i]

        if state == 'start':
            if char == '/':
                state = 'slash'
            else:
                return False

        elif state == 'slash':
            if char == '/':
                return True
            elif char == '*':
                state = 'multi_line'
            else:
                return False

        elif state == 'multi_line':
            if char == '*':
                state = 'multi_line_end'

        elif state == 'multi_line_end':
            if char == '/':
                return True
            elif char != '*':
```



```
                state = 'multi_line'

        i += 1

    return state == 'multi_line_end'

if __name__ == "__main__":
    try:
        with open("comments.txt", "r") as file:
            for line in file:
                line = line.strip()
                print(f"'{line}' is a valid comment:
{is_valid_comment(line)}")
    except FileNotFoundError:
        print("Error: 'comments.txt' file not found.")
```

Output:



```
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>  & 'C:\msy
-2022.16.1\pythonFiles\lib\python\debugpy\adapter/../..\debugp
d).py'
'Hello World' is a valid comment: False
'// This is a single-line comment' is a valid comment: True
'/* This is a multi-line comment */' is a valid comment: True
'Not a comment' is a valid comment: False
'/* Unclosed comment' is a valid comment: False
PS C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

2) E) Write a program to implement Lexical Analyzer.

Input2.txt

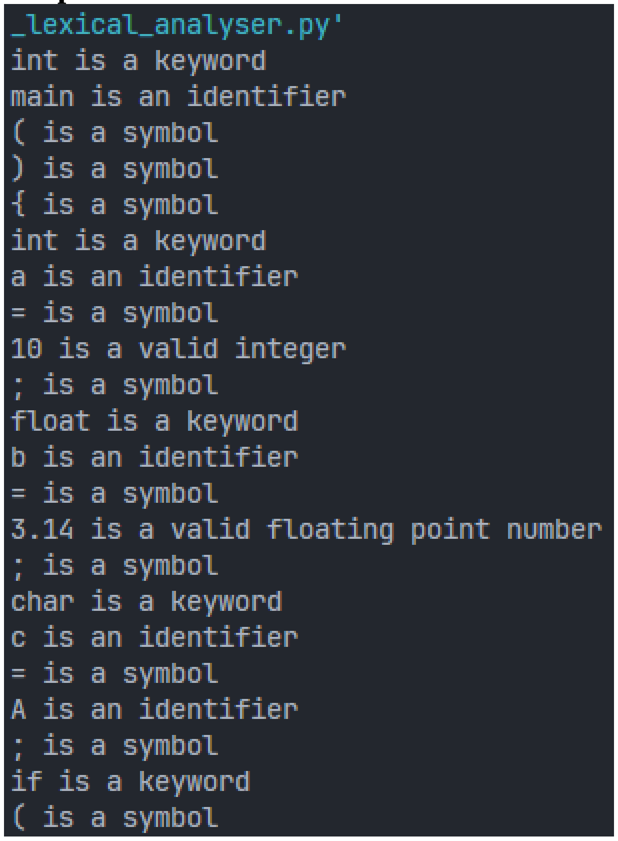
```
// This is a single-line comment
/* This is
  a multi-line comment */
int main() {
   int a = 10;
   float b = 3.14;
   char c = 'A';
   if (a < b) {
      a = a + 1;
   }
   return 0;
}
```

Source 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 FileNotFoundError:
        print("Error opening file")
        return

    state = 0
    lexeme = ""
    f = 0
    while f < len(buffer):
        c = buffer[f]
        if state == 0:
            if c.isalpha() or c == '_':
                state = 1
                lexeme += c
            elif c.isdigit():
                state = 13
                lexeme += c
            elif c == '/':
                state = 11
            elif c in " \t\n":
                state = 0
            elif c in ";,+-*/%=<>(){}[]":
                print(f"{c} is a symbol")
                state = 0
            else:
                state = 0
        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 "+-":
                state = 17
                lexeme += c
            elif c.isdigit():
                state = 18
                lexeme += c
            else:
                print("Error: Invalid scientific notation")
                lexeme = ""
                state = 0
        elif state == 17:
            if c.isdigit():
                state = 18
                lexeme += c
            else:
                print("Error: Invalid exponent format")
                lexeme = ""
                state = 0
        elif state == 18:
            if c.isdigit():
                lexeme += c
            else:
                print(f"{lexeme} is a valid scientific notation
number")
                lexeme = ""
                state = 0
                f -= 1
        f += 1
```

```
lexer("input2.txt")
```

Output:



```
_lexical_analyser.py'
int is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
int is a keyword
a is an identifier
= is a symbol
10 is a valid integer
; is a symbol
float is a keyword
b is an identifier
= is a symbol
3.14 is a valid floating point number
; is a symbol
char is a keyword
c is an identifier
= is a symbol
A is an identifier
; is a symbol
if is a keyword
( is a symbol
```

3) 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 tokenizing 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:

1. **Specification:** The user writes a .l file containing regular expressions and C actions.
2. **Processing:** The lex command processes the .l file and generates lex.yy.c.
3. **Compilation:** The lex.yy.c is compiled with gcc to create an executable scanner.
4. **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:

1. **Write a `` file** with pattern definitions and C-based actions.
2. **Use the `` command** to generate lex.yy.c.
3. **Compile the file** using gcc.
4. **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

Feature	**LEX**	**Flex**
Memory Usage	Higher	Optimized
Output File	lex.yy.c	lex.yy.c
Default Action	Returns first match	Returns longest match

Procedure

1. Create a .l file (e.g., lexer.l) containing regular expressions and C code.
2. Use the flex command to generate lex.yy.c.
3. Compile the generated C file using GCC.
4. 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 **tokenization** using **regular expressions** and **C functions**, making lexical analysis efficient.

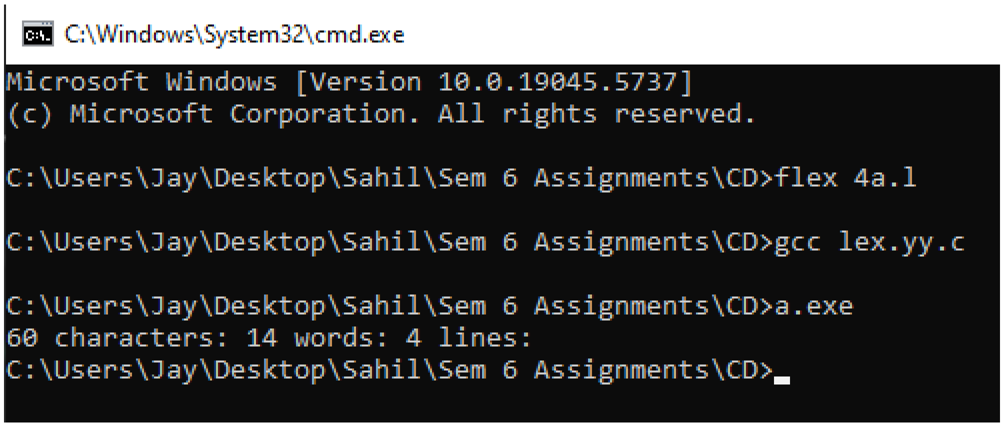
4) a) Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.

SOURCE CODE:
input.txt:

```
// abaabab jASX / SX//

Hello

This is my lex program

12345 677 34.676

56e56
```

4a.l

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

%%
\n { lines++; words++;}
[ \t]+ { words++; }
. { char_count++; }
%%

int main(){
yyin = fopen("input.txt", "r");
yylex();
printf("%d characters: ", char_count);
printf("%d words: ", words);
printf("%d lines: ", lines);
}
int yywrap(){
    return 1;
}
```



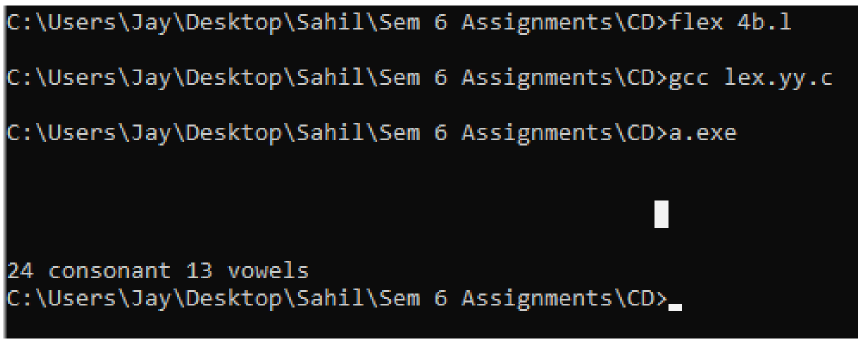
```
Microsoft Windows [Version 10.0.19045.5737]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>flex 4a.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>gcc lex.yy.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe
60 characters: 14 words: 4 lines:
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

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

SOURCE CODE:
input.txt:
// abaabab jASX / SX//
Hello
This is my lex program
12345 677 34.676

```
%{
#include<stdio.h>
int vowels=0, consonants=0;
%}
%%
[aeiouAEIOU] { vowels++; }
[a-zA-Z] { consonants++; }
. ;
%%
int main(){
yyin = fopen("input.txt","r");
yylex();
printf("%d consonant ", consonants);
printf("%d vowels ", vowels);
}
int yywrap(){ return 1; }
```

Output:



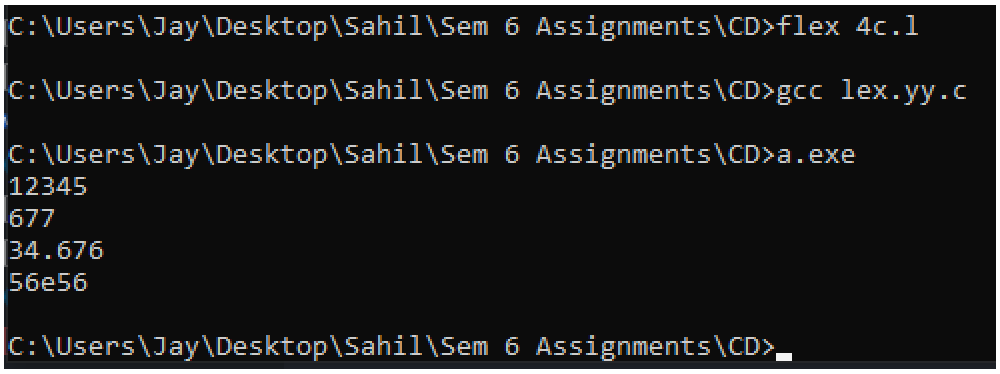
```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>flex 4b.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>gcc lex.yy.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe



24 consonant 13 vowels
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>_
```

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

SOURCE CODE:
input.txt:
// abaabab jASX / SX//
Hello
This is my lex program
12345 677 34.676
56e56

```
%{
#include<stdio.h>
%}
digits [0-9]+
%%
[+-]?{digits}(\.{digits})?([Ee][+-]?{digits})? { printf("%s\n", yytext); }
\n ;
. ;
%%
int main(){
yyin=fopen("file.txt","r");
yylex();
return 0;
}
int yywrap(){ return 1; }
```

Output:



```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>flex 4c.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>gcc lex.yy.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe
12345
677
34.676
56e56

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>_
```

4) d) Write a Lex program which adds line numbers to the given file and display the same into different file.
SOURCE CODE:

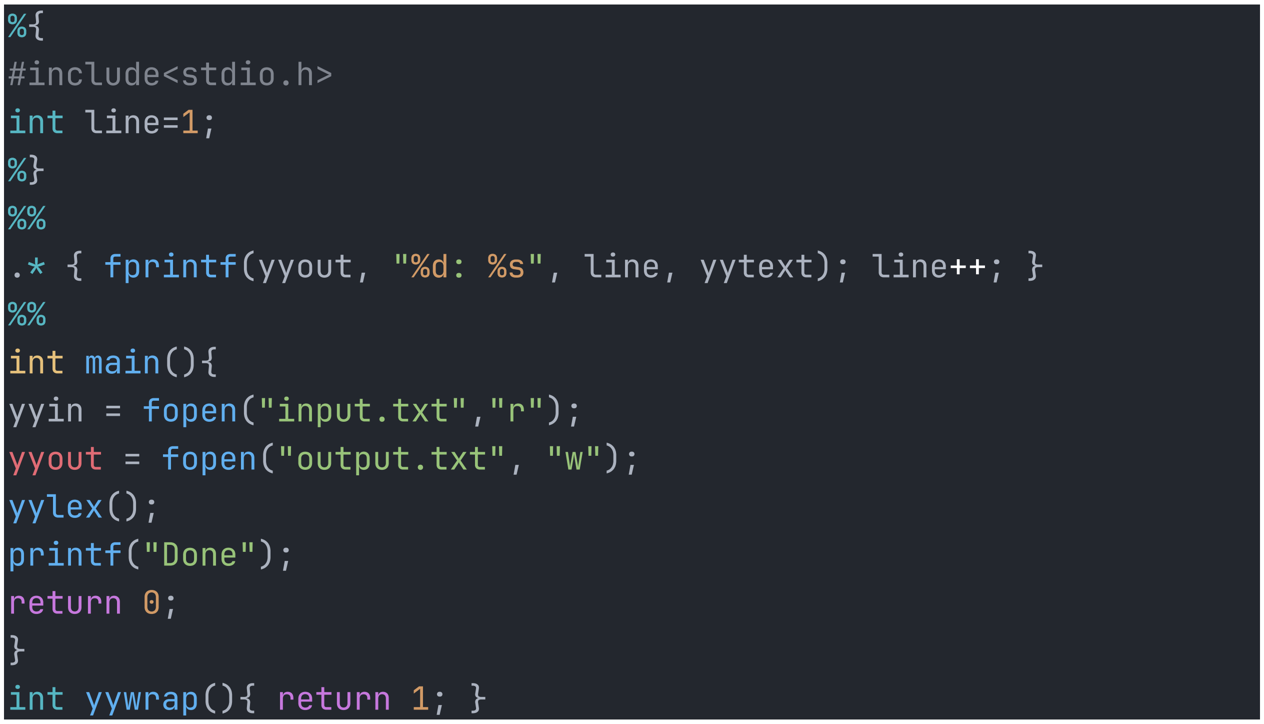
input.txt:

// abaabab jASX / SX//

Hello

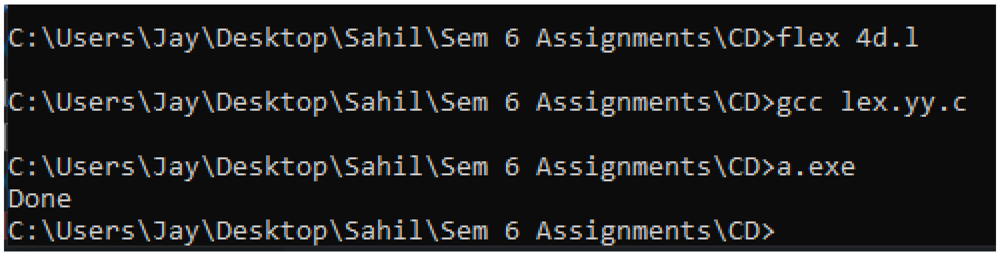
This is my lex program

12345 677 34.676

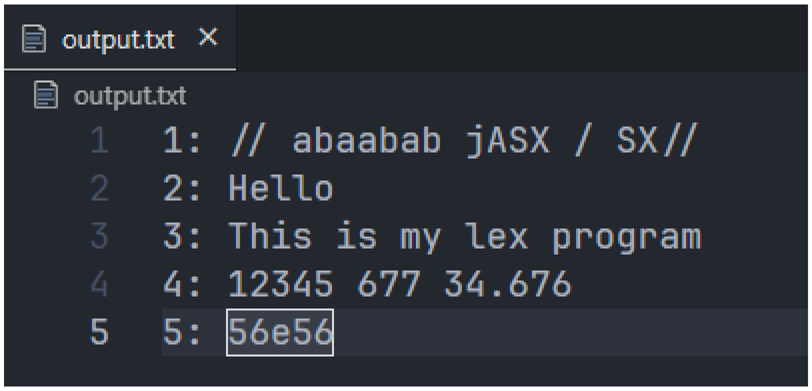


```
%{
#include<stdio.h>
int line=1;
%}
%%
.* { fprintf(yyout, "%d: %s", line, yytext); line++; }
%%
int main(){
yyin = fopen("input.txt","r");
yyout = fopen("output.txt", "w");
yylex();
printf("Done");
return 0;
}
int yywrap(){ return 1; }
```

Output:



```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>flex 4d.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>gcc lex.yy.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe
Done
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```


output.txt ×
output.txt

```
1: // abaabab jASX / SX//
2: Hello
3: This is my lex program
4: 12345 677 34.676
5: 56e56
```

4) e) Write a Lex program to printout all markup tags and HTML comments in file.
SOURCE CODE:

input.txt:

<html>

<head> Sahil Agrawal </head>

<body>

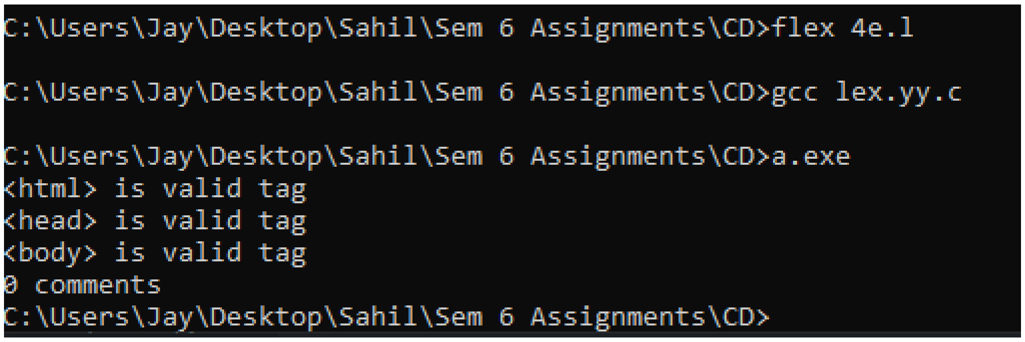
<!-- comment 122 -->

</body>

</html>

```
%{
#include<stdio.h>
int comment=0;
%}
%%
\<[a-zA-Z0-9]+">" { printf("%s is valid tag\n", yytext); }
"<!--"(.\n)*"-->" { comment++; }
. ;
\n ;
%%
int main(){
yyin = fopen("input.txt","r");
yylex();
printf("%d comments", comment);
return 0;
}
int yywrap(){
return 1;
}
```

Output:



```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>flex 4e.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>gcc lex.yy.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe
<html> is valid tag
<head> is valid tag
<body> is valid tag
0 comments
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

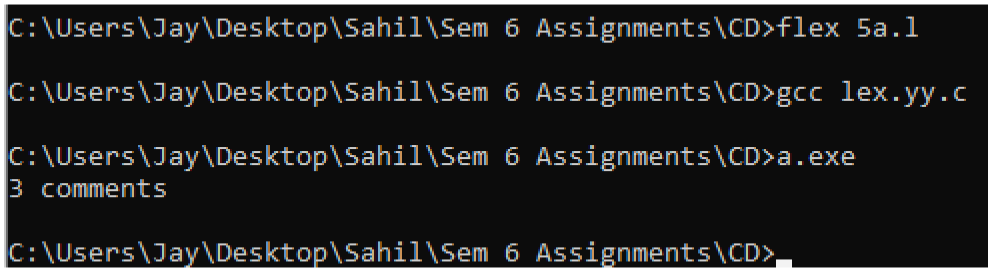
5) a) Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.
SOURCE CODE:
Comments.txt

```
Hello World /* This is an inline block comment */
// This is a single-line comment
/* This is a multi-line comment */
Not a comment
/* Unclosed comment
```

```
%{
#include <stdio.h>
int c = 0;
extern FILE *yyin, *yyout;
%}


%%
"//".*                    { c++; /* skip line comment */ }
"/*"([^*]|\*+[^*/])*"*"+"/"    { c++; /* skip block comment */ }
.|\n                      { fprintf(yyout, "%s", yytext); }
%%
```

```c
int main() {
    yyin = fopen("comments.txt", "r");
    yyout = fopen("output.txt", "w");
    yylex();
    printf("%d comments\n", c);
    return 0;
}

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

Output:



```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>flex 5a.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>gcc lex.yy.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe
3 comments

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

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

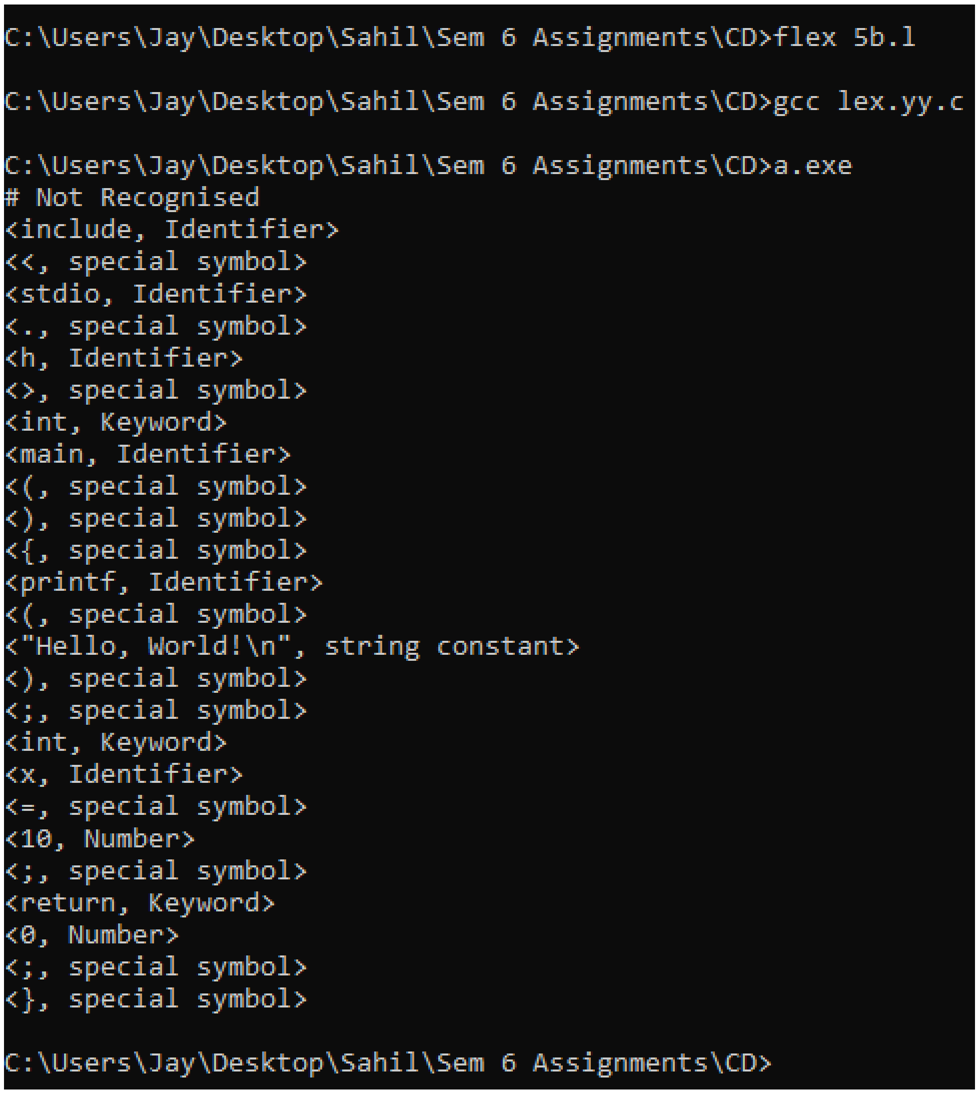
SOURCE CODE:

Code.txt

```
#include <stdio.h>
int main() {
   // This is a single-line comment
   printf("Hello, World!\n");
   /* This is
      a multi-line
      comment */
   int x = 10;  // Another comment
   return 0;
}
```

```
%{
#include<stdio.h>
%}
%%
if|else|while|do|switch|case|return|int {printf("<%s,
Keyword>\n", yytext);}
[a-zA-Z_][a-zA-Z0-9]* {printf("<%s, Identifier>\n", yytext);}
[0-9]+(\.[0-9]+)?([Ee][+-]?[0-9]+)? {printf("<%s, Number>\n",
yytext);}
"!"|"@"|"8"|"&"|"^"|"%"|")"|"("|"<"|">"|";"|","|"{"|"}"|"="|"."
{printf("<%s, special symbol>\n", yytext);}
[ \t\n]+ ;
"/*"[^*/]*"*/"  ;
"//"[^\n]+ ;
\"[^\"]+\" {printf("<%s, string constant>\n", yytext);}
. printf("%s Not Recognised\n", yytext);
%%
int main()
{
yyin = fopen("code.txt", "r");
yylex();
return 0;
}
int yywrap() {return(1);}
```

Output:

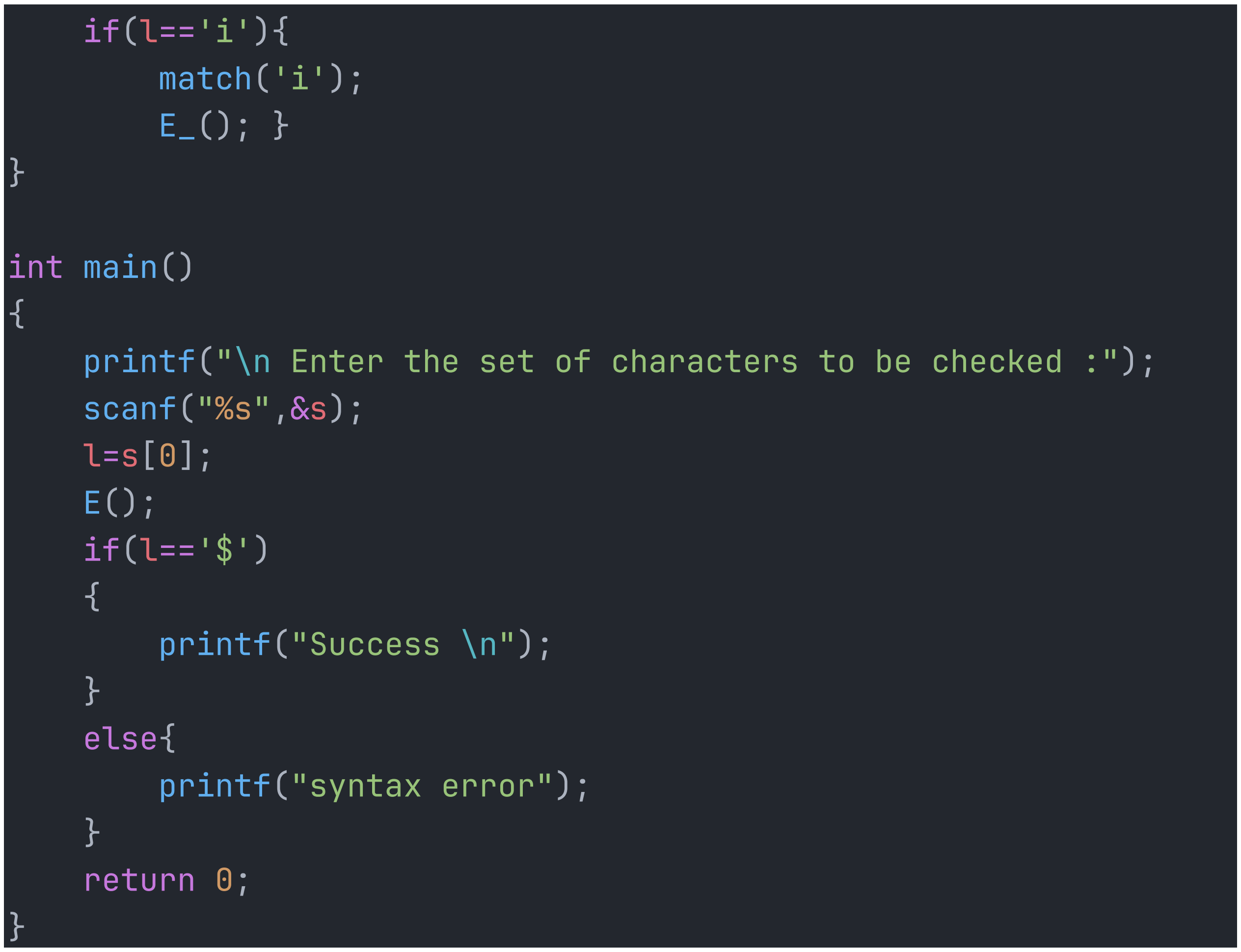


```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>flex 5b.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>gcc lex.yy.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe
# Not Recognised
<include, Identifier>
<<, special symbol>
<stdio, Identifier>
<., special symbol>
<h, Identifier>
<>, special symbol>
<int, Keyword>
<main, Identifier>
<(, special symbol>
<), special symbol>
<{, special symbol>
<printf, Identifier>
<(, special symbol>
<"Hello, World!\n", string constant>
<), special symbol>
<;, special symbol>
<int, Keyword>
<x, Identifier>
<=, special symbol>
<10, Number>
<;, special symbol>
<return, Keyword>
<0, Number>
<;, special symbol>
<}, special symbol>

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

6) Program to implement Recursive Descent Parsing in C.

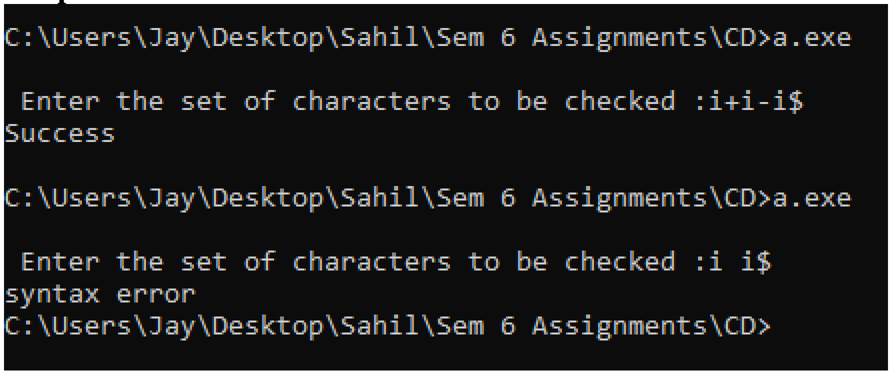
SOURCE CODE:

```
#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()
{
```



```c
    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\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe

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

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>a.exe

 Enter the set of characters to be checked :i i$
syntax error
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD>
```

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

YACC – Yet Another Compiler-Compiler

Introduction

- **YACC** is a **parser generator** developed by **Stephen C. Johnson** at AT&T Bell Labs.
- It automates the creation of a **syntax analyzer** (parser) based on a **context-free grammar (CFG)**.
- Typically used with **Lex**, which handles lexical analysis (tokenization).

Purpose of YACC

- Translates **grammar rules** into a **C program** that parses the input.
- Useful for building **compilers**, **interpreters**, and **language translators**.
- Converts a sequence of tokens (from Lex) into meaningful structures by checking the syntax.

Working of YACC

1. **Input**: Grammar written in YACC syntax in a .y file.
2. **YACC processes this file** and generates y.tab.c (a C source file for parsing).
3. The file is compiled along with Lex output (lex.yy.c) to create an executable.
4. When the program runs, it calls yyparse(), which processes input based on grammar rules.

Key Components

Component	**Description**
%token	Declares tokens (terminal symbols)
%%	Separates sections
Grammar rules	Rules in BNF form (e.g., expr : expr '+' term)
Semantic actions	C code within {} executed when a rule matches
yyparse()	Parser function generated by YACC
yyerror()	Error-handling function
yylval	Variable for passing values between Lex and YACC

How to write code in YACC:

A YACC program consists of three sections, just like LEX:

```
%{
// C declarations
%}
%%
// Grammar rules with actions
%%
// Supporting C code (like main)
```

Advantages of YACC

- Automatically handles syntax checking.
- Simplifies writing parsers for new languages.
- Flexible and extensible.
- Well-integrated with Lex.

Limitations

- Cannot parse all grammars (e.g., ambiguous or context-sensitive grammars).
- Requires understanding of grammar and parsing concepts.
- Original YACC only supports **LALR(1)** parsing.

Conclusion

YACC is a powerful tool that automates the generation of parsers. When used with Lex, it enables efficient and organized compiler development. Understanding YACC is essential for building any system that needs to interpret or validate complex language input.

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

SOURCE CODE:

sampleL.l

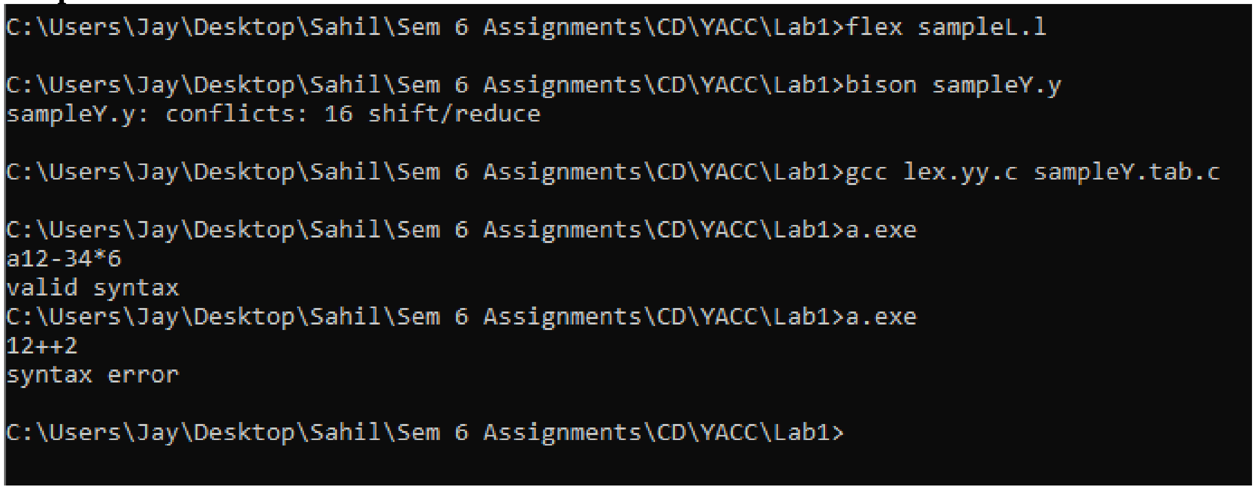
```
%{
#include <stdlib.h>
void yyerror(char *);
#include "sampleY.tab.h"
%}
%%
[0-9]+  return NUM;
[a-zA-Z_][a-zA-Z0-9_]* return id;
[-+*/\n] return *yytext;
[ \t] ;
. yyerror("invalid character");
%%
int yywrap() {
 return 1;
}
```

sampleY.y

```
%{
 #include<stdio.h>
 int yylex(void);
 void yyerror(char *);
%}
%token NUM
%token id
%%
S : E '\n' { printf("valid syntax");return 0; }
E : E '+' E { }
  | E '-' E { }
  | E '*' E { }
  | E '/' E { }
  | NUM { }
  | id { }
%%
```

```
void yyerror(char *s) {
 fprintf(stderr, "%s\n", s);
}
int main() {
 yyparse();
  return 0;
}
```

Output:



```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab1>flex sampleL.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab1>bison sampleY.y
sampleY.y: conflicts: 16 shift/reduce

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab1>gcc lex.yy.c sampleY.tab.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab1>a.exe
a12-34*6
valid syntax
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab1>a.exe
12++2
syntax error

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab1>
```

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

SOURCE CODE:
sampleL.l

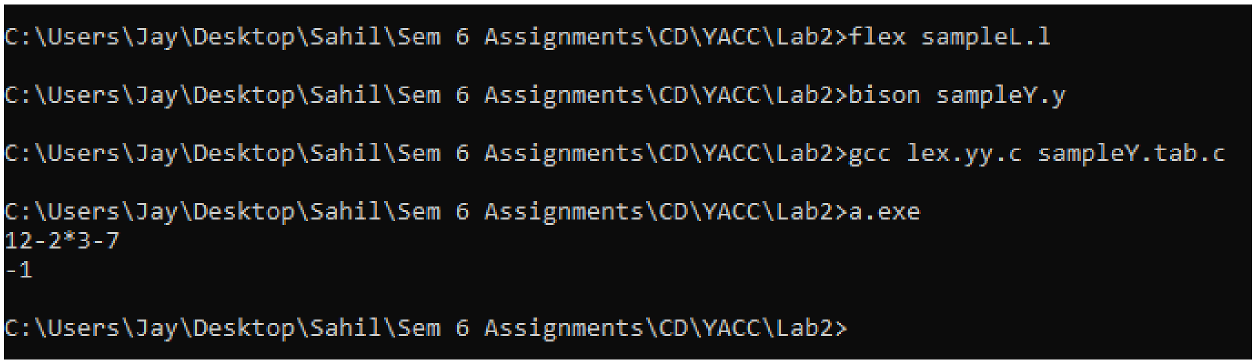
```
%{
#include <stdlib.h>
void yyerror(char *);
#include "sampleY.tab.h"
%}
%%
[0-9]+  {yylval = atoi(yytext); return NUM;}
[-+*\n] {return *yytext;}
[ \t] ; { }
. yyerror("invalid character");
%%
int yywrap() {
 return 0;
}
```

sampleY.y

```
%{
 #include<stdio.h>
 int yylex(void);
 void yyerror(char *);
%}
%token NUM
%token id
%%
S : E '\n' { printf("valid syntax");return 0; }
E : E '+' E { }
  | E '-' E { }
  | E '*' E { }
  | E '/' E { }
  | NUM { }
  | id { }
```

```
%%
void yyerror(char *s) {
 fprintf(stderr, "%s\n", s);
}
int main() {
 yyparse();
  return 0;
}
```

Output:



```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab2>flex sampleL.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab2>bison sampleY.y

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab2>gcc lex.yy.c sampleY.tab.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab2>a.exe
12-2*3-7
-1

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\Lab2>
```

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

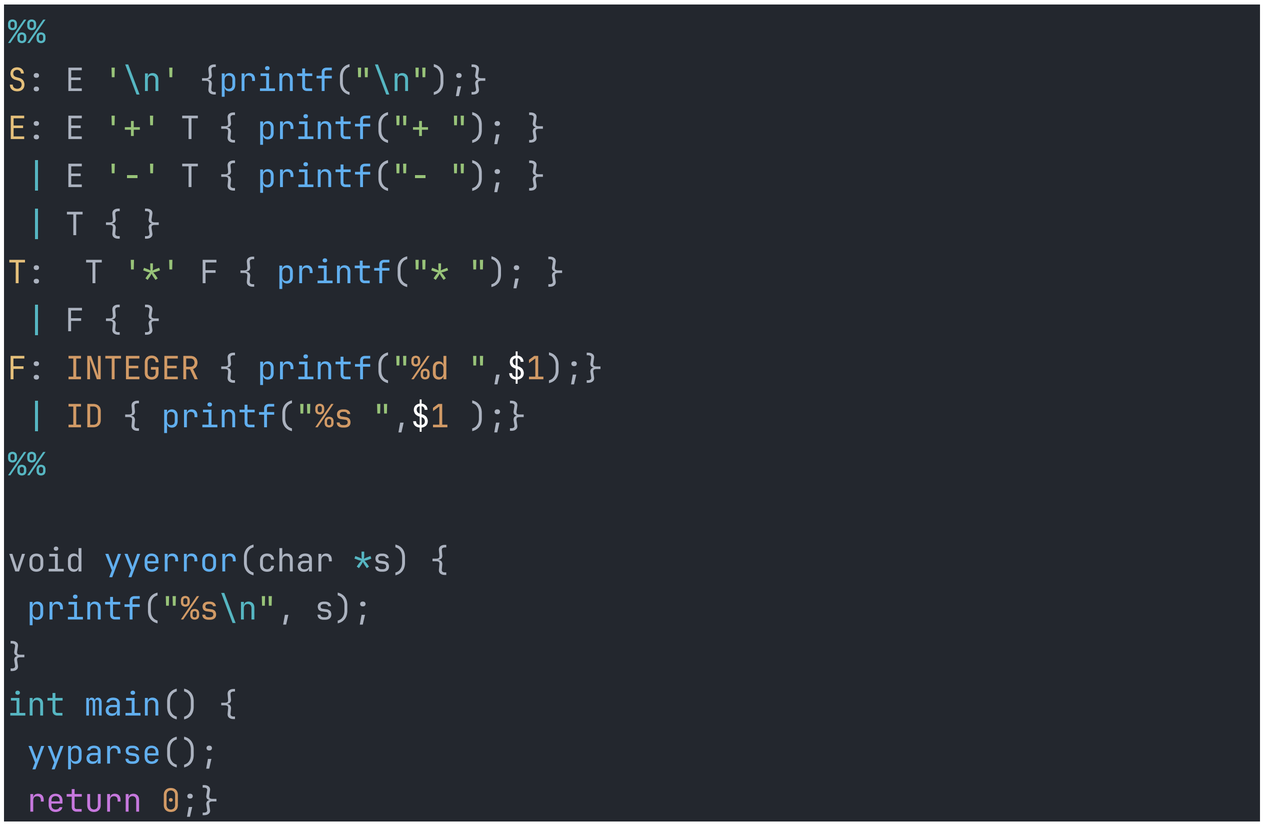
SOURCE CODE:

b.l

```
%{
#include <stdlib.h>
#include "b.tab.h"
void yyerror(char *);
%}

%%
[0-9]+ { yylval.num = atoi(yytext); return INTEGER; }
[A-Za-z_][A-Za-z0-9_]* { yylval.str = yytext; return ID; }
[-+;\n*] { return *yytext; }
[ \t] ;
. yyerror("invalid character");
%%

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

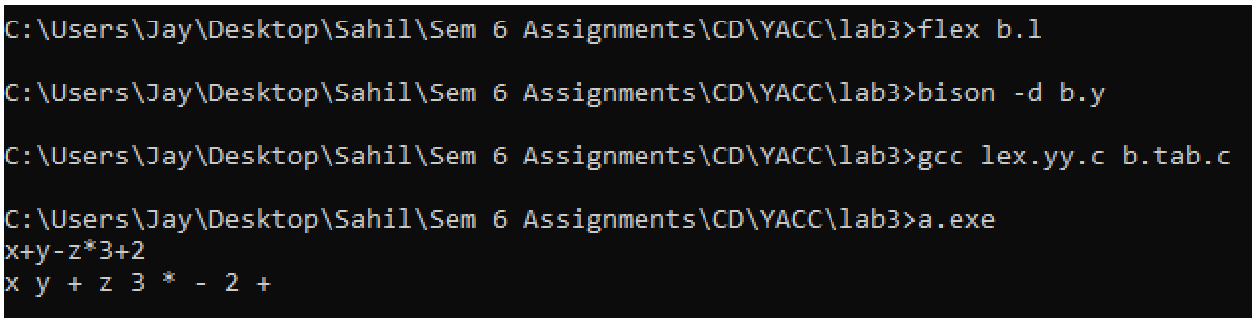
b.y

```
%{
 #include <stdio.h>
 int yylex(void);
 void yyerror(char *);
%}
%union {
    char *str;
    int num;
}
%token <num> INTEGER
%token <str> ID
```



```
%%
S: E '\n' {printf("\n");}
E: E '+' T { printf("+ "); }
 | E '-' T { printf("- "); }
 | T { }
T:  T '*' F { printf("* "); }
 | F { }
F: INTEGER { printf("%d ",$1);}
 | ID { printf("%s ",$1 );}
%%

void yyerror(char *s) {
 printf("%s\n", s);
}
int main() {
 yyparse();
 return 0;}
```

Output:



```
C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\lab3>flex b.l

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\lab3>bison -d b.y

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\lab3>gcc lex.yy.c b.tab.c

C:\Users\Jay\Desktop\Sahil\Sem 6 Assignments\CD\YACC\lab3>a.exe
x+y-z*3+2
x y + z 3 * - 2 +
```