SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Laboratory Manual

TABLE OF CONTENT

Sr. No	Experiment Title	
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.
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) Program to implement Lexical Analyzer.
3		To Study about Lexical Analyzer Generator (LEX) and Flex (Fast Lexical Analyzer)
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.
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.
6		Program to implement Recursive Descent Parsing in C.
7		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.

1.

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

CODE:

```
#include<stdio.h>

int main() {
                        char input[10];
                        int state=0, i=0;

                        printf("Enter the input 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;
                }
                i++;
            }

            if(state==0 || state==2) printf("String is invalid!");
            else if(state==1) printf("String is valid!");
            else printf("String not recognised.");

return 0;
}
```

OUTPUT:

```
Enter the input string: abba
String is valid!
--------------------------------
Process exited after 4.163 seconds with return value 0
Press any key to continue . . .
```

```
Enter the input string: baab
String is invalid!
--------------------------------
Process exited after 7.438 seconds with return value 0
Press any key to continue . . .
```

```
Enter the input string: abcde123
String not recognised.
--------------------------------
Process exited after 8.853 seconds with return value 0
Press any key to continue . . .
```

b) Write a program to accept string that ends with 'a'.

CODE:

```
#include<stdio.h>

int main() {
                        char input[10];
                        int state=0, i=0;

                        printf("Enter the input 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("String is invalid!");
                        else printf("String is valid!");

return 0;
```

}

OUTPUT:

```
Enter the input string: bcaa
String is valid!
--------------------------------
Process exited after 10.56 seconds with return value 0
Press any key to continue . . .
```

```
Enter the input string: abb
String is invalid!
--------------------------------
Process exited after 2.72 seconds with return value 0
Press any key to continue . . .
```

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

CODE:

```
#include<stdio.h>

int main() {
   char input[10];
   int state = 0, i = 0;

   printf("Enter the input 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 if (input[i] == 'b') state = 2;
            else state = 0;
            break;
         case 2:
            if (input[i] == 'a') state = 1;
            else state = 0;
            break;
      }
      i++;
   }
```

```
    if (state == 2) printf("String is valid!");
    else printf("String is invalid!");

    return 0;
}
```

OUTPUT:

```
Enter the input string: babab
String is valid!
--------------------------------
Process exited after 8.487 seconds with return value 0
Press any key to continue . . .
```

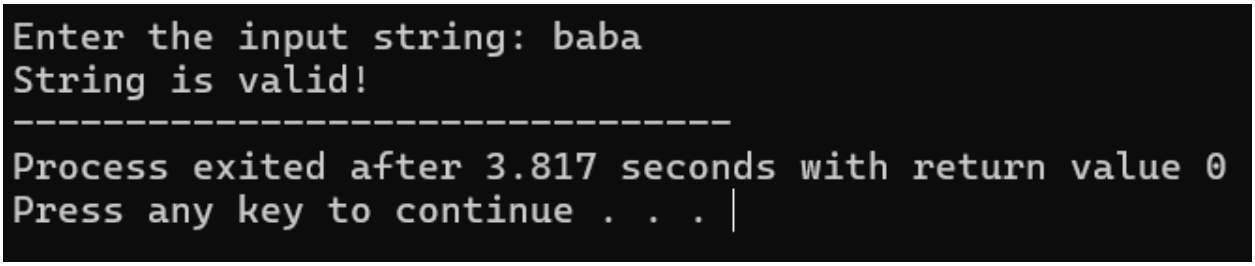
d) Write a program to recognize strings which contains ‘ab’. Take the input from text file.

CODE:

```
#include<stdio.h>


int main() {
   char input[100];
   int state = 0, i = 0;


   printf("Enter the input 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 if (input[i] == 'b') {
               state = 2;
            }
            else state = 0;
            break;
         case 2:
            state = 2;
            break;
      }
      i++;
```

```
    }

    if (state == 2) printf("String is valid!");
    else printf("String is invalid!");

    return 0;
}
```

OUTPUT:



2.

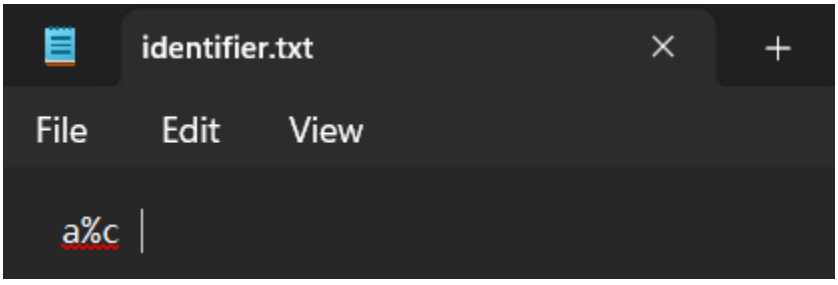
a) Write a program to recognize the valid identifiers.

CODE:

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

int main() {
   FILE *file;
   char filename[] = "identifier.txt";
   char ch;
   int state = 0, i = 0;



   file = fopen(filename, "r");



   if (file == NULL) {
      perror("Error opening file");
      return 1;
   }



   while ((ch = fgetc(file)) != EOF) {
      switch (state) {
         case 0:
            if (isalpha(ch) || ch=='_' || ch==' ') state = 1;
            else state = 2;
            break;

         case 1:
```

```
            if (isalpha(ch) || ch=='_' || isdigit(ch)) state = 1;
            else state = 2;
            break;

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


  if (state == 1)
    printf("String is a valid identifier\n");
  else
    printf("Invalid identifier\n");


  fclose(file);

return 0;
}
```

OUTPUT:



```
Invalid identifier

--------------------------------
Process exited after 5.194 seconds with return value 0
Press any key to continue . . .
```

b) Write a program to recognize the valid operators.

CODE:

```
#include <stdio.h>

int main() {
    char input[10];
    int i = 0;

    printf("Enter the input string: ");
    scanf("%s", input);

    int state = 50;

    while (input[i] != '\0') {
        switch (state) {
            case 50:
                if (input[i] == '+') {
                    if (input[i + 1] == '+') state = 100;
                    else if (input[i + 1] == '=') state = 101;
                    else if (input[i + 1] == '\0' || input[i + 1] == ' ') state = 102;
                    else state = -1;
                }
                else if (input[i] == '-') {
                    if (input[i + 1] == '-') state = 103;
                    else if (input[i + 1] == '=') state = 104;
                    else if (input[i + 1] == '\0' || input[i + 1] == ' ') state = 105;
                    else state = -1;
                }
                else if (input[i] == '*') {
                    if (input[i + 1] == '=') state = 107;
```

```
            else if (input[i + 1] == '\0' || input[i + 1] == ' ') state = 108;
            else state = -1;
        }
        else if (input[i] == '/') {
            if (input[i + 1] == '=') state = 109;
            else if (input[i + 1] == '\0' || input[i + 1] == ' ') state = 110;
            else state = -1;
        }
        else if (input[i] == '%') {
            if (input[i + 1] == '=') state = 111;
            else if (input[i + 1] == '\0' || input[i + 1] == ' ') state = 112;
            else state = -1;
        }
        else if (input[i] == '=') {
            if (input[i + 1] == '=') state = 119;
            else if (input[i + 1] == '\0' || input[i + 1] == ' ') state = 120;
            else state = -1;
        }
        else {
            state = -1;
        }
        break;
    }
    i++;
}

if (state == 100 || state == 103)
    printf("It is a valid unary operator\n");
else if (state == 102 || state == 105 || state == 108 || state == 110 || state == 112)
    printf("It is a valid arithmetic operator\n");
else if (state == 119)
```

```
        printf("It is a valid relational operator\n");
    else if (state == 101 || state == 104 || state == 107 || state == 109 || state == 111 || state == 120)
        printf("It is a valid assignment operator\n");
    else
        printf("Invalid Operator!\n");

    return 0;
}
```

OUTPUT:

```
Enter the input string: +
It is a valid arithmetic operator

--------------------------------
Process exited after 4.258 seconds with return value 0
Press any key to continue . . .
```

c) Write a program to recognize the valid numbers.

CODE:

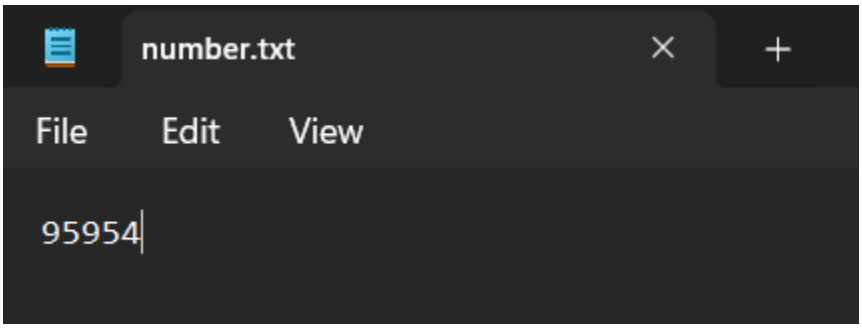
```
#include <stdio.h>
#include <ctype.h>

int main() {
    FILE *file;
    char filename[] = "number.txt";
    char ch;
    int state = 0;

    file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    while ((ch = fgetc(file)) != EOF) {
        switch (state) {
            case 0:
                if (isdigit(ch)) {
                    state = 1;  // Move to integer part detected
                } else {
                    state = 0;  // Ignore invalid start characters
                }
                break;

            case 1:
                if (isdigit(ch)) {
                    state = 1;  // Stay in integer part
```

```
    } else if (ch == '.') {
        state = 2;  // Decimal point detected
    } else if (ch == 'E' || ch == 'e') {
        state = 4;  // Exponent detected
    } else {
        state = 0;  // Invalid character, reset
    }
    break;

case 2:
    if (isdigit(ch)) {
        state = 3;  // Fractional part detected
    } else {
        state = 0;  // Invalid, reset
    }
    break;

case 3:
    if (isdigit(ch)) {
        state = 3;  // Stay in fractional part
    } else if (ch == 'E' || ch == 'e') {
        state = 4;  // Exponent detected
    } else {
        state = 0;  // Invalid, reset
    }
    break;

case 4:
    if (ch == '+' || ch == '-') {
        state = 5;  // Sign detected after E
    } else if (isdigit(ch)) {
```

```
                state = 6;  // Digit directly after E
            } else {
                state = 0;  // Invalid, reset
            }
            break;

        case 5:
            if (isdigit(ch)) {
                state = 6;  // Digit after exponent sign
            } else {
                state = 0;  // Invalid, reset
            }
            break;

        case 6:
            if (isdigit(ch)) {
                state = 6;  // Stay in exponent digits
            } else {
                state = 0;  // Invalid, reset
            }
            break;
    }
}

fclose(file);
if (state == 1 || state == 3 || state == 6) {
    printf("Valid number found in the file.\n");
} else {
    printf("No valid number found.\n");
}
return 0;
```

```
}
```

OUTPUT:




```
Valid number found in the file.

--------------------------------
Process exited after 1.004 seconds with return value 0
Press any key to continue . . .
```

d) Write a program to recognize the valid comments.

CODE:

```
#include<stdio.h>

int main() {
                    FILE *file;
                    char input[256];
                    int state=0, ch;

                    file = fopen("P4.txt", "r");

                    while ((ch = fgetc(file)) != EOF) {
                    switch(state){
                     case 0:
                            if(ch == '/') state=1;
                            else state=2;
                            break;
                     case 1:
                            if(ch == '/') state=3;
                            else if(ch == '*') state=4;
                            else state=2;
                            break;
                     case 2:
                            state=2;
                            break;
                     case 3:
                            state=3;
                            break;
                     case 4:
                            if(ch == '*') state=5;
```

```
            else state=4;
            break;
        case 5:
            if(ch == '/') state=6;
            else state=4;
            break;
        case 6:
            state=2;
            break;
        }
    }

    if(state==3 || state==6) printf("Valid Comment!");
    else printf("Invalid Comment!");

    fclose(file);

return 0;
}
```

OUTPUT:

e) Program to implement Lexical Analyzer.

CODE:

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

#define BUFFER_SIZE 1000

void check(char *lexeme);

void main() {
  FILE *f1;
  char buffer[BUFFER_SIZE], lexeme[50];
  char c;
  int f = 0, state = 0, i = 0;

  f1 = fopen("input.txt", "r");
  if (!f1) {
    printf("Error opening file.\n");
    return;
  }

  int bytesRead = fread(buffer, sizeof(char), BUFFER_SIZE - 1, f1);
  buffer[bytesRead] = '\0'; // Null terminate
  fclose(f1);

  while (buffer[f] != '\0') {
    switch (state) {
```

```
case 0:
    c = buffer[f];

    if (isalpha(c) || c == '_') {
        state = 1;  // Identifier or Keyword
        lexeme[i++] = c;
    }
    else if (isdigit(c)) {
        state = 13;  // Number
        lexeme[i++] = c;
    }
    else if (c == '/') {
        state = 11;  // Potential comment
    }
    else if (c == ';' || c == ',' || c == '{' || c == '}' || c == '(' || c == ')') {
        printf(" %c is a symbol\n", c);
    }
    else if (strchr("+-*/=<>!&|", c)) {
        state = 20;  // Operator
        lexeme[i++] = c;
    }
    break;

case 1:
    c = buffer[f];
    if (isalnum(c) || c == '_') {
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        check(lexeme);
```

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

    case 11:
        c = buffer[f];
        if (c == '/') {
            while (buffer[f] != '\n' && buffer[f] != '\0') f++;
            printf("Single-line comment detected\n");
        }
        else if (c == '*') {
            f++;
            while (buffer[f] != '\0' && !(buffer[f] == '*' && buffer[f + 1] == '/')) f++;
            f += 2;
            printf("Multi-line comment detected\n");
        }
        else {
            printf("/ is an operator\n");
            f--;
        }
        state = 0;
        break;

    case 13:
        c = buffer[f];
        if (isdigit(c)) {
            lexeme[i++] = c;
        } else if (c == '.') {
```

```
            state = 14;
            lexeme[i++] = c;
        } else if (c == 'E' || c == 'e') {
            state = 16;
            lexeme[i++] = c;
        } else {
            lexeme[i] = '\0';
            printf("%s is a valid number\n", lexeme);
            i = 0;
            state = 0;
            f--;
        }
        break;

    case 14:
        c = buffer[f];
        if (isdigit(c)) {
            lexeme[i++] = c;
        } else if (c == 'E' || c == 'e') {
            state = 16;
            lexeme[i++] = c;
        } else {
            lexeme[i] = '\0';
            printf("%s is a valid floating point number\n", lexeme);
            i = 0;
            state = 0;
            f--;
        }
        break;
```

```
case 16:
   c = buffer[f];
   if (isdigit(c) || c == '+' || c == '-') {
      state = 17;
      lexeme[i++] = c;
   } else {
      lexeme[i] = '\0';
      printf("%s is a valid number\n", lexeme);
      i = 0;
      state = 0;
      f--;
   }
   break;

case 17:
   c = buffer[f];
   if (isdigit(c)) {
      lexeme[i++] = c;
   } else {
      lexeme[i] = '\0';
      printf("%s is a valid scientific notation number\n", lexeme);
      i = 0;
      state = 0;
      f--;
   }
   break;

case 20:
   c = buffer[f];
   if ((lexeme[0] == '=' && c == '=') ||
```

```
            (lexeme[0] == '!' && c == '=') ||
            (lexeme[0] == '>' && c == '=') ||
            (lexeme[0] == '<' && c == '=') ||
            (lexeme[0] == '&' && c == '&') ||
            (lexeme[0] == '|' && c == '|')) {
            lexeme[i++] = c;
            lexeme[i] = '\0';
            printf("%s is an operator\n", lexeme);
            i = 0;
            state = 0;
          } else {
            lexeme[i] = '\0';
            printf("%s is an operator\n", lexeme);
            i = 0;
            state = 0;
            f--;
          }
          break;
      }
      f++;
    }
}

void check(char *lexeme) {
    char *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"
    };
```

```
    for (int i = 0; i < 32; i++) {
        if (strcmp(lexeme, keywords[i]) == 0) {
            printf("%s is a keyword\n", lexeme);
            return;
        }
    }
    printf("%s is an identifier\n", lexeme);
}
```

OUTPUT:


```
int is a keyword
a is an identifier
= is an operator
10 is a valid number
 ; is a symbol
float is a keyword
b is an identifier
= is an operator
3.14 is a valid floating point number
 ; is a symbol
if is a keyword
 ( is a symbol
a is an identifier
> is an operator
5 is a valid number
 ) is a symbol
 { is a symbol
return is a keyword
a is an identifier
 ; is a symbol
 } is a symbol
Single-line comment detected

--------------------------------
Process exited after 0.8809 seconds with return value 0
Press any key to continue . . .
```

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

✅ What is Lexical Analysis?

Lexical Analysis is the first phase of a compiler, responsible for scanning the source code and breaking it into tokens (such as keywords, identifiers, numbers, operators, etc.). This phase removes whitespaces and comments and prepares a stream of tokens for the parser.

🔧 Lexical Analyzer Generator (LEX)

What is LEX?

- LEX is a tool for automatically generating a Lexical Analyzer (Scanner).
- It takes a set of rules (patterns) written in regular expressions, and generates C code to recognize these patterns.
- Output: A C program (lex.yy.c) that reads an input and identifies tokens.

⚡ Flex (Fast Lexical Analyzer)

What is Flex?

- Flex is an enhanced and faster version of LEX, widely used today.
- It is POSIX-compliant, more flexible, and generates more efficient C code.
- Syntax is largely compatible with LEX, but Flex offers additional options, speed improvements, and better debugging.

Advantages of Flex over Lex:

Lex	**Flex (Fast Lex)**
Older, limited features	Modern, optimized
Slower execution	Faster execution
Less portable	Highly portable
Limited debugging	Built-in verbose & debug options
Static tables	Dynamic memory handling

🌐 Use of LEX/Flex in Compiler Design:

- Creating custom lexical analyzers for new languages (like Devta Lang).
- Tokenizing code for interpreters, compilers, and preprocessors.
- Syntax highlighting tools, code analyzers, and formatters.

4. Implement following programs using Lex.

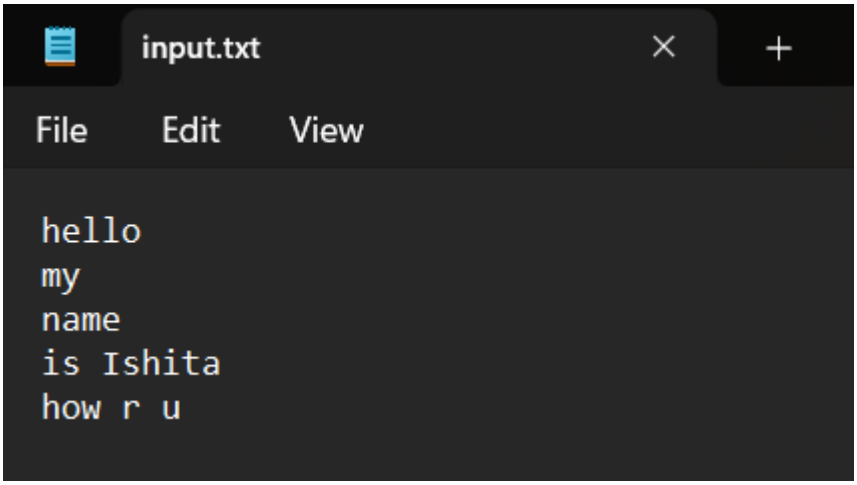
a) Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.

CODE:

```
%{
   #include<stdio.h>
   int letters=0, words=0, lines=1;
%}
%%
[a-zA-Z] letters++;
[\t ] words++;
\n {lines++; words++;}
%%
void main(){
   yyin=fopen("input.txt", "r");
   yylex();
   fclose(yyin);
   printf("The file contains %d letters", letters);
   printf("\nThe file contains %d words", words);
   printf("\nThe file contains %d lines", lines);

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

OUTPUT:

```
D:\22000810\Compiler Design\Lab Programs\Flex1>flex sample.l

D:\22000810\Compiler Design\Lab Programs\Flex1>gcc lex.yy.c

D:\22000810\Compiler Design\Lab Programs\Flex1>a.exe
The file contains 24 letters
The file contains 8 words
The file contains 5 lines
D:\22000810\Compiler Design\Lab Programs\Flex1>
```



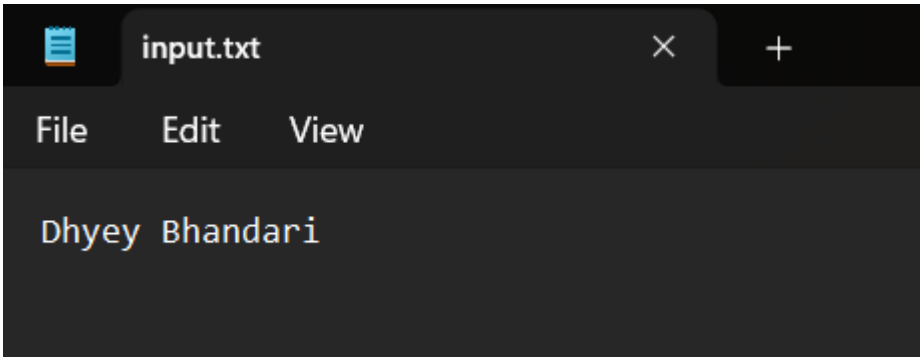
b) Write a Lex program to take input from text file and count number of vowels and consonants.

CODE:

```
%{
   #include<stdio.h>
   int vowels=0, consonants=0;
%}
%%
[aeiouAEIOU] vowels++;
[a-zA-Z] { consonants++; }
. ;
%%
void main(){
   yyin=fopen("input.txt", "r");
   yylex();
   fclose(yyin);
   printf("The file contains %d vowels & %d consonants.", vowels, consonants);

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

OUTPUT:

```
D:\22000810\Compiler Design\Lab Programs\Flex2>flex sample.l

D:\22000810\Compiler Design\Lab Programs\Flex2>gcc lex.yy.c

D:\22000810\Compiler Design\Lab Programs\Flex2>a.exe
The file contains 4 vowels & 9 consonants.
D:\22000810\Compiler Design\Lab Programs\Flex2>
```

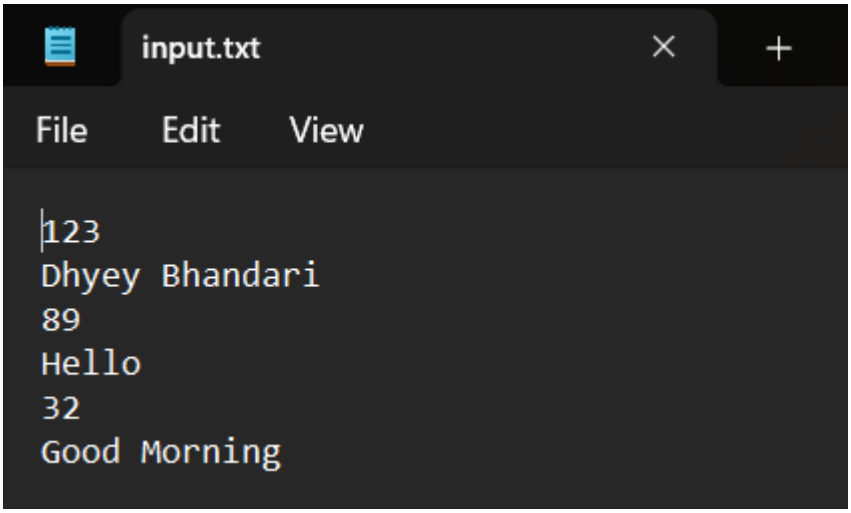

input.txt
File
Edit
View
Dhyey Bhandari

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

CODE:

```
%{
   #include<stdio.h>
   int numbers[100], size=0;
%}
%%
[0-9] numbers[size++] = atoi(yytext);
[\n] ;
. ;
%%
void main(){
   yyin=fopen("input.txt", "r");
   yylex();
   fclose(yyin);
   printf("Numbers in the file: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", numbers[i]);
   }
}
int yywrap() { return(1); }
```

OUTPUT:




```
D:\22000810\Compiler Design\Lab Programs\Flex3>flex sample.l

D:\22000810\Compiler Design\Lab Programs\Flex3>gcc lex.yy.c

D:\22000810\Compiler Design\Lab Programs\Flex3>a.exe
Numbers in the file: 1 2 3 8 9 3 2
D:\22000810\Compiler Design\Lab Programs\Flex3>
```

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

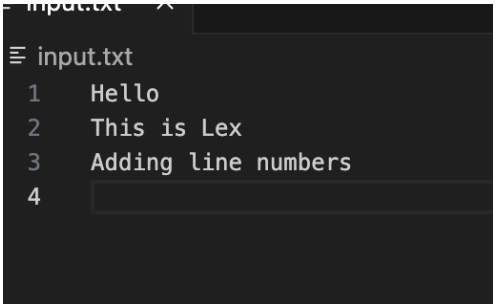
CODE:

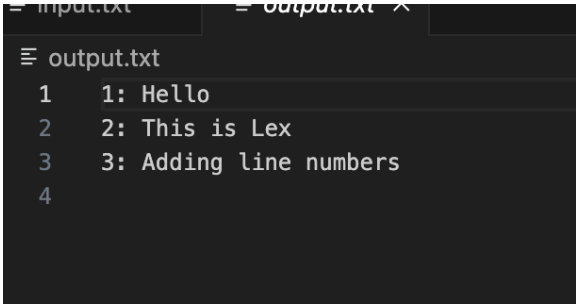
```
%{
#include <stdio.h>
int line_number = 1;
FILE *out;
%}

%%

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

.|\n      {
            // For any characters missed (like last line without newline)
            fprintf(out, "%d: %s\n", line_number++, yytext);
        }

%%

int yywrap() {
  return 1;
}

int main() {
  FILE *in = fopen("input.txt", "r");
  out = fopen("output.txt", "w");
```

```
    if (!in || !out) {
        printf("Error opening files!\n");
        return 1;
    }

    yyin = in;
    yylex();

    fclose(in);
    fclose(out);

    printf("Line numbers added successfully to output.txt\n");
    return 0;
}
```

OUTPUT:

```
Line numbers added successfully to output.txt
```





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

CODE:

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

%%
"<!--"([^<]|"<"[^!]|"<!"[^\-]|"<!-"[^-])*"-->"    { printf("HTML Comment: %s\n", yytext); }

\<[^>]*\>                          { printf("HTML Tag: %s\n", yytext); }

.|\n                          ; // Ignore everything else
%%

int yywrap() {
  return 1;
}

int main() {
  yyin = fopen("input.html", "r");
  if (!yyin) {
    printf("Error opening input.html\n");
    return 1;
  }

  yylex();  // start scanning
  fclose(yyin);
  return 0;
}
```

OUTPUT:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <b>hi how are u </b>
    <h1>23</h1>
    <!-- hi how r u  -->
</body>
</html>
```

```
HTML Tag: <!DOCTYPE html>
HTML Tag: <html lang="en">
HTML Tag: <head>
HTML Tag: <meta charset="UTF-8">
HTML Tag: <meta name="viewport" content="width=device-width, initial-scale=1.0">
HTML Tag: <title>
HTML Tag: </title>
HTML Tag: </head>
HTML Tag: <body>
HTML Tag: <b>
HTML Tag: </b>
HTML Tag: <h1>
HTML Tag: </h1>
HTML Comment: <!-- hi how r u  -->
HTML Tag: </body>
HTML Tag: </html>
```

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.

CODE:

```
%{
   #include<stdio.h>
   int single=0, multi=0;
%}
%%
"//".* single++;
"/*"([^*]|\*+[^*/])*"*/" multi++;
%%
void main() {
   yyin=fopen("input.txt", "r");
   yylex();
   fclose(yyin);
   printf("The number of single-line comments are: %d", single);
   printf("The number of multi-line comments are: %d", multi);
}
int yywrap() { return 1; }
```

OUTPUT:

```
D:\22000810\Compiler Design\Lab Programs\Flex5>flex sample.l

D:\22000810\Compiler Design\Lab Programs\Flex5>gcc lex.yy.c

D:\22000810\Compiler Design\Lab Programs\Flex5>a.exe


The number of single-line comments are: 1The number of multi-line comments are: 2
D:\22000810\Compiler Design\Lab Programs\Flex5>
```

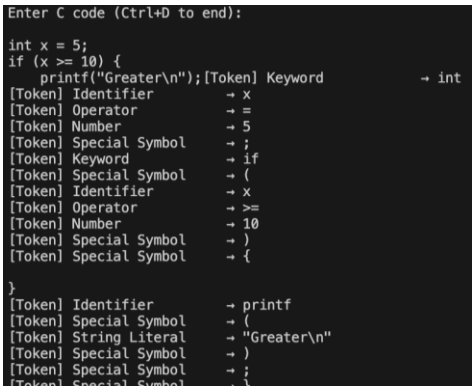
b) Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

CODE:

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

// List of C keywords
char *keywords[] = {
   "auto","break","case","char","const","continue","default","do","double","else",
   "enum","extern","float","for","goto","if","int","long","register","return",
   "short","signed","sizeof","static","struct","switch","typedef","union",
   "unsigned","void","volatile","while"
};

int isKeyword(char *str) {
   for (int i = 0; i < sizeof(keywords)/sizeof(char*); i++) {
      if (strcmp(str, keywords[i]) == 0)
         return 1;
   }
   return 0;
}
%}

%option noyywrap

%%
[0-9]+(\.[0-9]+)?     { printf("[Token] %-18s → %s\n", "Number", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* {
                  if (isKeyword(yytext))
```

```
                printf("[Token] %-18s → %s\n", "Keyword", yytext);
            else
                printf("[Token] %-18s → %s\n", "Identifier", yytext);
        }
"++"|"--"|"+"|"-"|"=="|"="|"<="|">="|"!="|"&&"|"||"|"*"|"/"|"%"   { printf("[Token] %-18s → %s\n", "Operator", yytext); }
[\[\]\{\}\(\),;\.:]   { printf("[Token] %-18s → %s\n", "Special Symbol", yytext); }
\'([^\\\n]|(\\.))\'   { printf("[Token] %-18s → %s\n", "Char Literal", yytext); }
\"([^\\\n]|(\\.))*\"   { printf("[Token] %-18s → %s\n", "String Literal", yytext); }
[ \t\n]+          ;  // Skip whitespace
.                 { printf("[Token] %-18s → %s\n", "Unknown", yytext); }
%%

int main() {
   printf("Enter C code (Ctrl+D to end):\n\n");
   yylex();
   return 0;
}
```

OUTPUT:



6. Program to implement Recursive Descent Parsing in C.

CODE:

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

const char *input;
int pos = 0;

int match(char expected) {
   if (input[pos] == expected) {
      pos++;
      return 1;
   } else {
      printf("Syntax Error: Expected '%c' at position %d\n", expected, pos);
      exit(1);
   }
}

int E();
int E_prime();

int E() {
   if (input[pos] == 'i') {
      match('i');
      return E_prime();
   } else {
      printf("Syntax Error: Expected 'i' at position %d\n", pos+1);
      exit(1);
```

```
    }
}

int E_prime() {
    if (input[pos] == '+') {
        pos++;
        if (input[pos] == 'i') {
            match('i');
            return E_prime();
        } else {
            printf("Syntax Error: Expected 'i' after '+' at position %d\n", pos+1);
            exit(1);
        }
    } else if (input[pos] == '-') {
        pos++;
        if (input[pos] == 'i') {
            match('i');
            return E_prime();
        } else {
            printf("Syntax Error: Expected 'i' after '-' at position %d\n", pos+1);
            exit(1);
        }
    }
    return 1;
}

int parse() {
    E();
    if (input[pos] == '\0') {
        printf("Parsing successful!\n");
```

```
        return 1;
    } else {
        printf("Syntax Error: Unexpected characters at position %d\n", pos+1);
        exit(1);
    }
}

int main() {
    input = "i+i-i";
    parse();
    return 0;
}
```

OUTPUT:

```
Parsing successful!

--------------------------------
Process exited after 0.6792 seconds with return value 0
Press any key to continue . . .
```

7.

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

✅ What is YACC?

- YACC stands for Yet Another Compiler-Compiler.
- It is a parser generator tool that produces a syntactic analyzer (parser) based on a given grammar.
- Developed by Stephen C. Johnson at Bell Labs.
- Works closely with LEX/FLEX, which supplies tokens to YACC.
- YACC is used to create LR(1) parsers, especially LALR(1) (Look-Ahead LR) parsers by default.

🔧 Purpose of YACC:

- To automate the construction of parsers that process the sequence of tokens provided by the lexical analyzer.
- To verify the syntactic structure of programs or languages (e.g., programming languages, domain-specific languages).
- To build abstract syntax trees (AST) or directly execute code via actions.

🔄 **Integration with LEX/FLEX:**

LEX/FLEX	YACC
Lexical Analyzer	Syntax Analyzer (Parser)
Produces tokens	Consumes tokens
Tokenizes input	Validates token sequence
Generates lex.yy.c	Generates y.tab.c

Communication:

- YACC expects a function yylex() from LEX/FLEX.
- Tokens defined in YACC (%token) are sent by LEX via return TOKEN;.

✅ **Features of YACC:**

- **LALR(1) Parsing**: Efficient, handles most programming language grammars.
- **Error Recovery**: Allows grammar rules for handling syntax errors using error.
- **Semantic Actions**: Embed C code inside grammar to perform actions.
- **Symbol Table Handling**, AST construction, and code generation support.

🚀 **Advantages:**

- Automates parser writing.
- Reduces human errors in writing parsers.
- Well-integrated with C.
- Handles complex grammars.

🛠 **Tools Similar to YACC:**

Tool	Description
Bison	GNU replacement for YACC (widely used)
ANTLR	Modern parser generator (supports multiple languages)
JavaCC	Parser generator for Java

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

CODE:

Flex:

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

void yyerror(char *);
%}

%%
[0-9]+       { yylval = atoi(yytext); return NUM; }
[\n]         { return '\n'; }
[ \t]        ;
.            { return yytext[0]; }
%%

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

Bison:

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

int yylex(void);
void yyerror(char *s);
```

```
%}

%token NUM

%%
S : E '\n'     { printf("Valid syntax\n"); return 0; }
  ;

E : E '+' T
  | E '-' T
  | T
  ;

T : T '*' F
  | T '/' F
  | F
  ;

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

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

int main() {
    printf("Enter expression: ");
    yyparse();
```

```
    return 0;
}
```

OUTPUT:

```
D:\22000810\Compiler Design\CD_Lab_EndSem\syntax>bison -d sample.y

D:\22000810\Compiler Design\CD_Lab_EndSem\syntax>flex sample.l

D:\22000810\Compiler Design\CD_Lab_EndSem\syntax>gcc lex.yy.c sample.tab.c

D:\22000810\Compiler Design\CD_Lab_EndSem\syntax>a.exe
Enter expression: 2*3+5
Valid syntax

D:\22000810\Compiler Design\CD_Lab_EndSem\syntax>a.exe
Enter expression: 2+
Error: syntax error
```

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

CODE:

Flex:

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

%%

[0-9]+     { yylval = atoi(yytext); return NUM; }
[ \t]      ; // Ignore spaces and tabs
[-+*/()\n]  return yytext[0];
.          { printf("Invalid character: %s\n", yytext); }

%%

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

Bison:

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

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

```
%token NUM

%%
S : E '\n'       { printf("Result = %d\n", $1); return 0; }
 ;

E : E '+' T      { $$ = $1 + $3; }
 | E '-' T       { $$ = $1 - $3; }
 | T             { $$ = $1; }
 ;

T : T '*' F      { $$ = $1 * $3; }
 | T '/' F       {
                  if ($3 == 0) {
                    yyerror("Division by zero");
                    exit(1);
                  }
                  $$ = $1 / $3;
                }
 | F             { $$ = $1; }
 ;

F : NUM          { $$ = $1; }
 | '(' E ')'     { $$ = $2; }
 ;
%%

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

```
}

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

OUTPUT:

```
D:\22000810\Compiler Design\CD_Lab_EndSem\calculator>bison -d calc.y

D:\22000810\Compiler Design\CD_Lab_EndSem\calculator>flex calc.l

D:\22000810\Compiler Design\CD_Lab_EndSem\calculator>gcc lex.yy.c calc.tab.c

D:\22000810\Compiler Design\CD_Lab_EndSem\calculator>a.exe
Enter an arithmetic expression:
2*4+3
Result = 11
```

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

CODE:

Flex:

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

digit   [0-9]
number  {digit}+

%%

"+"       { return PLUS; }
"-"       { return MINUS; }
"*"       { return MUL; }
"/"       { return DIV; }
"%"       { return MOD; }
"("       { return LPAREN; }
")"       { return RPAREN; }
{number}    { yylval.intval = atoi(yytext); return NUMBER; }
[ \t]      { /* skip whitespace */ }
\n        { return '\n'; }
.         { return yytext[0]; }

%%

int yywrap() {
   return 1;
```

```
}
```

Bison:

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

extern int yylex();
extern int yyparse();
extern FILE *yyin;

void yyerror(const char *s);
void infix_to_postfix(char *expr);

char expr_buffer[1024];
%}

%union {
   int intval;
}

%token <intval> NUMBER
%token PLUS MINUS MUL DIV MOD
%token LPAREN RPAREN

%type <intval> expr term factor

%%
```

```
program:
   | program expr '\n' {
       printf("Postfix Expression: ");
       infix_to_postfix(expr_buffer);
       printf("\n");
       expr_buffer[0] = '\0';
   }
   ;

expr:
     term              { $$ = $1; }
   | expr PLUS term       { strcat(expr_buffer, "+ "); }
   | expr MINUS term      { strcat(expr_buffer, "- "); }
   ;

term:
     factor            { $$ = $1; }
   | term MUL factor      { strcat(expr_buffer, "* "); }
   | term DIV factor      { strcat(expr_buffer, "/ "); }
   | term MOD factor      { strcat(expr_buffer, "% "); }
   ;

factor:
     NUMBER            {
                         char temp[20];
                         sprintf(temp, "%d ", $1);
                         strcat(expr_buffer, temp);
                         $$ = $1;
                       }
   | LPAREN expr RPAREN    { $$ = $2; }
```

```
    ;

%%

int main(void) {
    printf("Enter an arithmetic expression: ");
    expr_buffer[0] = '\0';  // Clear buffer before use
    yyparse();
    return 0;
}

void infix_to_postfix(char *expr) {
    printf("%s", expr);
}

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

OUTPUT:

```
D:\22000810\Compiler Design\CD_Lab_EndSem\infix>bison -d infix.y

D:\22000810\Compiler Design\CD_Lab_EndSem\infix>flex infix.l

D:\22000810\Compiler Design\CD_Lab_EndSem\infix>gcc lex.yy.c infix.tab.c

D:\22000810\Compiler Design\CD_Lab_EndSem\infix>a.exe
Enter an arithmetic expression: 2+3+5
Postfix Expression: 2 3 + 5 +
```