

SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6<sup>TH</sup> SEMESTER

DEPARTMENT OF COMPUTER SCIENCE &  
ENGINEERING

Name=Bhuvan Rathod

Enrollment number=22000994

Subject=Compiler design

## 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.
--	--	--

## Practical-1

**Aim1:** Write a program to recognize strings starts with 'a' over {a, b}.

Procedure: Source

Code:

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

void main()
{
    int n, i = 0, state = 0;
    printf("Enter the length of the string: ");
    scanf("%d", &n);
    char input[n];
    printf("Enter the string: ");
    scanf("%s", &input);
    while (input[i] != '\0')
    {
        switch (state)
        {
            case 0:
                if (input[i] == 'a')
                {
                    state = 1;
                }
                else if (input[i] == 'b')
                {

```

```
state = 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++;
}
if (state == 0 || state == 2)
{
    printf("String is invalid");
}
else if (state == 1)

```

```
{  
    printf("String is valid");  
}  
else  
    printf("String is not recognized");  
}
```

Output:

```
D:\6 sem\compiler design\lab>gcc q1.c -o q1  
  
D:\6 sem\compiler design\lab>q1  
Enter the length of the string: 3  
Enter the string: abb  
String is valid
```

**Aim2:** Write a program to recognize strings end with 'a'.

Procedure: Source

Code:

```
#include <stdio.h>  
#include <stdlib.h>  
void main()  
{  
    int n, i = 0, state = 0;  
    printf("Enter the length of the string: ");  
    scanf("%d", &n);
```

```
{  
char input[n];  
printf("Enter the string: ");  
scanf("%s", &input);  
while (input[i] != '\0')
```



```
{
    switch (state)
    {
    case 0:
        if (input[i] == 'a')
        {
            state = 1;
        }
        else
        {
            state = 0;
        }
        break;
    case 1:
        if (input[i] == 'a')
        {
            state = 1;
        }
        else
        {
            state = 0;
        }
        break;
    }
    i++;
}
if (state == 0)
```

```

    {
        printf("String is invalid");
    }
    else if (state == 1)
    {
        printf("String is valid");
    }
}

```

Output:

```

D:\6 sem\compiler design\lab>gcc q2.c -o q2

D:\6 sem\compiler design\lab>q2
Enter the length of the string: 3
Enter the string: aaa
String is valid

```

**Aim3:** Write a program to recognize strings end with 'ab'. Take the input from text file.

Procedure: Source

Code:

```

#include <stdio.h>

void main()
{
    int state = 0, i = 0;
    FILE *fptr;
    fptr = fopen("input.txt", "r");
    char input[100];
    fgets(input, 100, fptr);
}

```

```
{  
printf("Input string: %s", input);  
fclose(fptr);
```

```
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 if (input[i] == 'a')
        {
            state = 1;
        }
        else
            state = 0;
        break;
    case 2:
        if (input[i] == 'a')
        {
```

```

        state = 1;
    }
    else
        state = 0;
    break;
}
i++;
}

if (state == 2)
{
    printf("\nString is valid");
}
else
{
    printf("\nString is invalid");
}
}

```

Output:

```

Input string: bbsvaadbafvvab
String is valid

```

**Aim4:** Write a program to recognize strings contains 'ab'. Take the input from text file.

## Procedure: Source

### Code:

```
#include <stdio.h>

void main()
{
    int state = 0, i = 0;
    FILE *fptr;
    fptr = fopen("input.txt", "r");
    char input[100];
    fgets(input, 100, fptr);
    printf("Input string: %s", input);
    fclose(fptr);
    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 if (input[i] == 'a')
        {
            state = 1;
        }
        else
            state = 0;
        break;
    case 2:
        state = 2;
        break;
    }
    i++;
}

if (state == 2)
{
    printf("\nString is valid");
}
else
{
    printf("\nString is invalid");
}
}

```



**Output:**

```
Input string: bbsvaadbafvvab
```

```
String is valid
```

```
C:\Users\Kunal\Desktop\Project\src\main\java\com\example\StringValidator.java:11: warning: [deprecation] isAlphanumeric() in java.lang.String has been deprecated
```

## Practical-2

**Aim1:** Write a program to recognize the valid identifiers.

Procedure: Source

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
void main()
{
    int state = 0, i = 0;
    FILE *fptr;
    fptr = fopen("input1.txt", "r");
    char input[100];
    fgets(input, 100, fptr);
    printf("Input string: %s", input);
    fclose(fptr);

    while (input[i] != '\0')
    {
        switch (state)
        {
            case 0:
                if (input[i] == 'i')
                {
                    state = 1;
                }
                else if (input[i] == '_' || isalpha(input[i]))
                {
                    state = 4;
                }
                else
                {
                    state = 5;
                }
                break;
            case 1:
                if (input[i] == 'n')
                {
                    state = 2;
                }
            }
```

```
else if (input[i] == '_' || isalpha(input[i]))  
{
```

```

        state = 4;
    }
    else
    {
        state = 5;
    }
    break;

case 2:
    if (input[i] == 't')
    {
        state = 3;
    }
    else if (input[i] == '_' || isalpha(input[i]))
    {
        state = 4;
    }
    else
    {
        state = 5;
    }
    break;
case 3:
    if (isalpha(input[i]) || isdigit(input[i]) || input[i] == '_')
    {
        state = 4;
    }
    else
    {
        state = 5;
    }
    break;
case 4:
    if (isalpha(input[i]) || isdigit(input[i]) || input[i] == '_')
    {
        state = 4;
    }
    else
    {
        state = 5;
    }
    break;
}
i++;
}

```

```

if (state == 4)
{
    printf("\n String is Identifier");
}
else if (state == 3)
{
    printf("\n String is a valid Keyword");
}
else if (state == 5)
{
    printf("\n String is invalid string");
}
else
{
    printf("\n String is empty");
}
}

```

## Output:

```

C:\Users\Kenil Patel\Desktop\second sem third year degree\Compiler Design\practical\practical2>gcc first.c -o first
C:\Users\Kenil Patel\Desktop\second sem third year degree\Compiler Design\practical\practical2>first
Input string: intdsggf_dsgsd
String is Identifier
C:\Users\Kenil Patel\Desktop\second sem third year degree\Compiler Design\practical\practical2>

```

**Aim2:** Write a program to recognize the valid operators.

## Procedure: Source Code:

```

#include <stdio.h>

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

    FILE *file = fopen("operator.txt", "r");
    if (file == NULL)
    {
        printf("Error opening file.\n");
        return 1;
    }
}

```

```
fscanf(file, "%s", input);
```

```

fclose(file);

while (input[i] != '\0')
{
    switch (state)
    {
    case 0:
        if (input[i] == '+')
        {
            state = 1;
        }
        else if (input[i] == '-')
        {
            state = 5;
        }
        else if (input[i] == '*')
        {
            state = 9;
        }
        else if (input[i] == '/')
        {
            state = 12;
        }
        else if (input[i] == '%')
        {
            state = 15;
        }
        else if (input[i] == '&')
        {
            state = 18;
        }
        else if (input[i] == '|')
        {
            state = 21;
        }
        else if (input[i] == '<')
        {
            state = 24;
        }
        else if (input[i] == '>')
        {
            state = 28;
        }
        else if (input[i] == '!')
        {

```

```

        state = 32;
    }
    else if (input[i] == '~')
    {
        state = 34;
    }
    else if (input[i] == '^')
    {
        state = 35;
    }
    else if (input[i] == '=')
    {
        state = 36;
    }
    break;

case 1:
    if (input[i] == '+')
    {
        state = 2;
        printf("++ unary operator");
    }
    else if (input[i] == '=')
    {
        state = 3;
        printf("+= assignment operator");
    }
    else
    {
        state = 4;
        printf("+ arithmetic operator");
    }
    break;

case 5:
    if (input[i] == '-')
    {
        state = 6;
        printf("-- unary operator");
    }
    else if (input[i] == '=')
    {
        state = 7;
        printf("-= assignment operator");
    }
    }

```



```

else
{
    state = 8;
    printf("+ arithmetic operator");
}
break;

case 9:
if (input[i] == '=')
{
    state = 10;
    printf("*= assignment operator");
}
else
{
    state = 11;
    printf("* arithmetic operator");
}
break;

case 12:
if (input[i] == '=')
{
    state = 13;
    printf("/= assignment operator");
}
else
{
    state = 14;
    printf("/ arithmetic operator");
}
break;

case 15:
if (input[i] == '=')
{
    state = 16;
    printf("%= assignment operator");
}
else
{
    state = 17;
    printf("% arithmetic operator");
}
break;

```

```

case 18:
    if (input[i] == '&')
    {
        state = 19;
        printf("&& Logical operator");
    }
    else
    {
        state = 20;
        printf("% Bitwise operator");
    }
    break;

case 21:
    if (input[i] == '|')
    {
        state = 22;
        printf("|| Logical operator");
    }
    else
    {
        state = 23;
        printf("| Bitwise operator");
    }
    break;

case 24:
    if (input[i] == '<')
    {
        state = 25;
        printf("<< Bitwise operator");
    }
    else if (input[i] == '=')
    {
        state = 27;
        printf("<= Relational operator");
    }
    else
    {
        state = 26;
        printf("< Relational operator");
    }
    break;

```

```

case 28:
    if (input[i] == '>')
    {
        state = 29;
        printf(">> Bitwise operator");
    }
    else if (input[i] == '=')
    {
        state = 30;
        printf(">= Relational operator");
    }
    else
    {
        state = 31;
        printf("> Relational operator");
    }
    break;

case 32:
    if (input[i] == '=')
    {
        state = 33;
        printf("!= Assignment operator");
    }
    break;

case 36:
    if (input[i] == '=')
    {
        state = 37;
        printf("== Relational operator");
    }
    break;

default:
    break;
}

i++;
}

printf("\nState is %d\n", state);
if (state == 1)
{
    printf("+ arithmetic operator\n");
}

```

```

}
else if (state == 5)
{
    printf("- arithmetic operator\n");
}
else if (state == 9)
{
    printf("* arithmetic operator\n");
}
else if (state == 12)
{
    printf("/ arithmetic operator\n");
}
else if (state == 15)
{
    printf("% arithmetic operator\n");
}
else if (state == 18)
{
    printf("& Bitwise operator\n");
}
else if (state == 21)
{
    printf("| Bitwise operator\n");
}
else if (state == 24)
{
    printf("< Relational operator\n");
}
else if (state == 28)
{
    printf("> Relational operator\n");
}
else if (state == 32)
{
    printf("! Logical operator\n");
}
else if (state == 34)
{
    printf("~ Bitwise operator\n");
}
else if (state == 35)
{
    printf("^ Bitwise operator\n");
}

```

```

else if (state == 36)
{
    printf("= Assignment operator\n");
}
return 0;
}

```

Output:

```

++ unary operator
State is 2

```

**Aim3:** Write a program to recognize the valid number.

Procedure: Source

Code:

```

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void main()
{
    char c, buffer[1000], lexeme[1000];
    int i = 0, state = 0, f = 0, j = 0;
    FILE *fp = fopen("digitInput.txt", "r");
    while ((c = fgetc(fp)) != EOF && j < 1000)
    {
        buffer[j++] = c;
    }
    buffer[j] = '\0';
    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))
    {
    }
    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);
}
}

```

Output:

```
The input 100 is a valid integer.
```

**Aim4:** Write a program to recognize the valid comments.

Procedure: Source

Code:

```

#include <stdio.h>
void main()
{
    int state = 0, i = 0;
    FILE *fptr;
    fptr = fopen("input.txt", "r");
    char input[100];
    fgets(input, 100, fptr);
    printf("Input string: %s", input);
    fclose(fptr);
    while (input[i] != '\0')
    {

```

```

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

case 4:
    if (input[i] == '*')
    {
        state = 5;
    }
    else
    {
        state = 4;
    }
    break;
case 5:
    if (input[i] == '/')
    {
        state = 6;
    }
    else
    {
        state = 4;
    }

```

```

        }
        break;
    case 6:
        if (input[i] != '\0')
        {
            state = 3;
        }
        break;
    }
    i++;
}
if (state == 2)
{
    printf("\n String is valid single line comment");
}
else if (state == 6)
{
    printf("\nString is a valid multiline comment");
}
else
{
    printf("\n String is invalid comment ");
}
}

```

Output:

```

Input string: /*ffddsagasg***ad*s*ag***231313146*/
String is a valid multiline comment

```

**Aim5:** Write a program to implement Lexical Analyzer

Procedure:

**Source Code:**

```

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

#define BUFFER_SIZE 1000

```

```

void check_keyword_or_identifier(char *lexeme);
void recognize_number(char *lexeme);
void recognize_operator(char *buffer, int *index);
void recognize_comment(char *buffer, int *index);

```

```

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

    f1 = fopen("input2.txt", "r");
    if (f1 == NULL)
    {
        printf("Error: Could not open input.txt\n");
        return;
    }

    fseek(f1, 0, SEEK_END);
    long file_size = ftell(f1);
    rewind(f1);

    buffer = (char *)malloc(file_size + 1);

    fread(buffer, 1, file_size, f1);
    buffer[file_size] = '\0';
    fclose(f1);

    while (buffer[f] != '\0')
    {
        c = buffer[f];
        switch (state)
        {
            case 0:
                if (isalpha(c) || c == '_')
                {
                    state = 1;
                    lexeme[i++] = c;
                }
                else if (isdigit(c))
                {
                    state = 2;

```

```

        lexeme[i++] = c;
    }
    else if (c == '/' && (buffer[f + 1] == '/' || buffer[f + 1] == '*'))
    {
        recognize_comment(buffer, &f);
        state = 0;
    }
    else if (strchr("+-*/%=<>!", c))
    {
        recognize_operator(buffer, &f); // Pass buffer and index for operator handling
        state = 0;
    }
    else if (strchr(";,{}()", c))
    {
        printf("%c is a symbol\n", c);
        state = 0;
    }
    else if (isspace(c))
    {
        state = 0;
    }
    break;

case 1:
    if (isalnum(c) || c == '_')
    {
        lexeme[i++] = c;
    }
    else
    {
        lexeme[i] = '\0';
        check_keyword_or_identifier(lexeme);
        lexeme[i] = '\0';
        i = 0;
        state = 0;
        f--; // Go back to recheck the current character
    }
    break;

case 2:
    if (isdigit(c))
    {
        lexeme[i++] = c;
    }
    else if (c == '.')

```

```

    {
        state = 3;
        lexeme[i++] = c;
    }
    else if (c == 'E' || c == 'e')
    {
        state = 4;
        lexeme[i++] = c;
    }
    else
    {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--; // Go back to recheck the current character
    }
    break;

case 3:
    if (isdigit(c))
    {
        lexeme[i++] = c;
    }
    else
    {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--; // Go back to recheck the current character
    }
    break;

case 4:
    if (isdigit(c) || c == '+' || c == '-')
    {
        state = 5;
        lexeme[i++] = c;
    }
    else
    {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;

```

```

        state = 0;
        f--; // Go back to recheck the current character
    }
    break;

case 5:
    if (isdigit(c))
    {
        lexeme[i++] = c;
    }
    else
    {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--; // Go back to recheck the current character
    }
    break;
}
f++; // Move forward in the buffer
}

free(buffer);
}

void check_keyword_or_identifier(char *lexeme)
{
    int i = 0;
    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 (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);
}

```

```

}

void recognize_number(char *lexeme)
{
    printf("%s is a valid number\n", lexeme);
}

void recognize_operator(char *buffer, int *index)
{
    char operators[][3] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">="};
    char op[3] = {buffer[*index], buffer[*index + 1], '\0'};
    int i = 0;

    // Handle two-character operators (e.g., "==" or ">=")
    for (i = 0; i < 12; i++)
    {
        if (strcmp(op, operators[i]) == 0)
        {
            printf("%s is an operator\n", op);
            (*index)++; // Skip the next character since we used two chars
            return;
        }
    }

    // Handle single-character operators
    printf("%c is an operator\n", buffer[*index]);
}

void recognize_comment(char *buffer, int *index)
{
    if (buffer[*index] == '/' && buffer[*index + 1] == '/')
    {
        printf("// is a single-line comment\n");
        while (buffer[*index] != '\n' && buffer[*index] != '\0')
            (*index)++;
    }
    else if (buffer[*index] == '/' && buffer[*index + 1] == '*')
    {
        printf("/* is the start of a multi-line comment\n");
        (*index) += 2;
        while (!(buffer[*index] == '*' && buffer[*index + 1] == '/') && buffer[*index] != '\0')
            (*index)++;
        if (buffer[*index] == '*' && buffer[*index + 1] == '/')
        {
            printf("*/ is the end of a multi-line comment\n");
        }
    }
}

```



```
        (*index) += 2;
    }
}
}
```

### Output:

```
int main(){
b=1000, a = 10;
return 0;

}
```

```
int is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
b is an identifier
= is an operator
100 is a valid number
, is a symbol
a is an identifier
= is an operator
10 is a valid number
; is a symbol
return is a keyword
0 is a valid number
; is a symbol
} is a symbol
```

## Practical-3

**Aim1:** To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)

### Procedure:

When we write a program, the compiler has to first break it down into smaller parts like keywords, numbers, operators, and variable names. This process is called lexical analysis, and it's done by a component called the lexical analyzer. Instead of writing all the code ourselves to identify these parts, we can use a tool called LEX — or its faster version, Flex — to do the job for us. We simply define the patterns (like what a number looks like or what makes something a keyword), and LEX/Flex creates a C program that can recognize those patterns in any input.

In this experiment, we used LEX or Flex to build a small program that scans text and picks out specific tokens — for example, identifying numbers, keywords, or symbols. It helped me understand how a compiler starts its work by reading the raw code and making sense of it piece by piece. This experiment was really helpful because it showed me how regular expressions are used in real tools, and how compilers begin processing code from the very first line.

## Practical-4

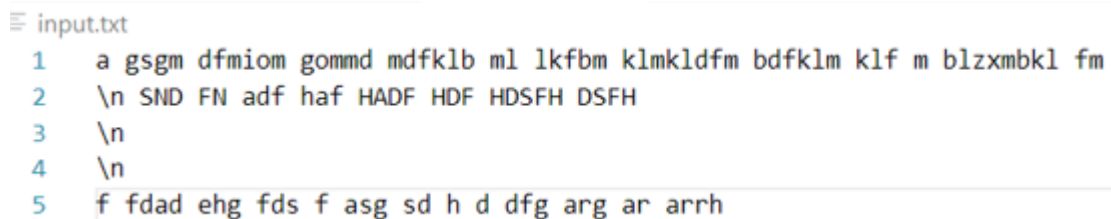
**Aim1:** Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.

### Procedure:

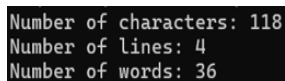
#### Source Code:

```
%{
#include<stdio.h>
int l=0, w=0, c=0;
int in_word=0;
%}
%%
[a-zA-Z] {c++; in_word = 1;}
\n {l++; if (in_word) {w++;in_word=0;}}
[ \t]+ {if (in_word){w++; in_word=0;}}
. {if (in_word){w++; in_word = 0;}}
%%
void main(){
    yyin = fopen("input.txt", "r");
    yylex();
    printf("Number of characters: %d \nNumber of lines: %d \nNumber of words: %d", c, l,w);
}
int yywrap(){return (1);}
```

### Output:



```
≡ input.txt
1  a gsgm dfmion gommd mdfklb ml lkfbm klmklfdm bdfklm klf m blzmbkl fm
2  \n SND FN adf haf HADF HDF HDSFH DSFH
3  \n
4  \n
5  f fdad ehg fds f asg sd h d dfg arg ar arrh
```



```
Number of characters: 118
Number of lines: 4
Number of words: 36
```

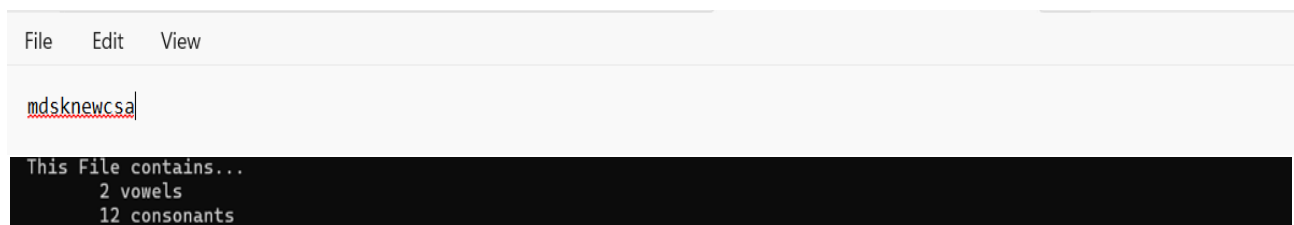
**Aim2:** Write a Lex program to take input from text file and count number of vowels and consonants.

## Procedure:

### Source Code:

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

## Output:



```
File Edit View
mdsknewcsa|
This File contains...
 2 vowels
12 consonants
```

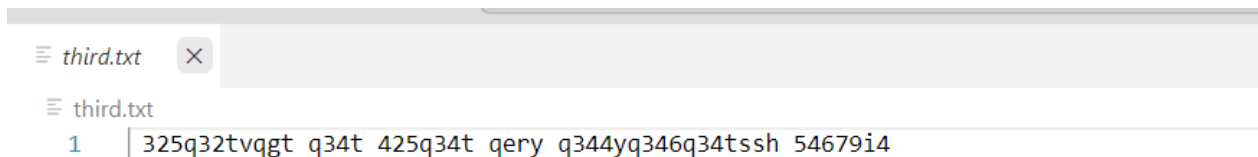
**Aim3:** Write a Lex program to print out all numbers from the given file.

## Procedure:

### Source Code:

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

## Output:

A screenshot of a text editor window. The title bar shows 'third.txt' and a close button. The editor content shows the text '325q32tvqgt q34t 425q34t qery q344yq346q34tssh 54679i4' on a single line, with a line number '1' on the left margin.

```
325 is valid number
32 is valid number
34 is valid number
425 is valid number
34 is valid number
344 is valid number
346 is valid number
34 is valid number
54679 is valid number
4 is valid number
```

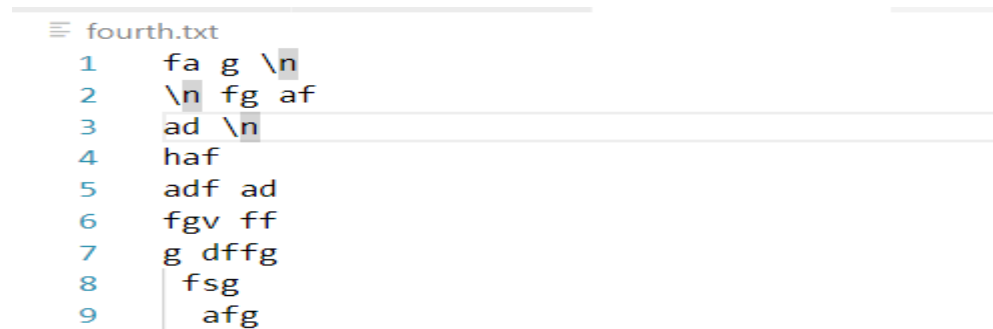
**Aim4:** Write a Lex program which adds line numbers to the given file and display the same into different file.

## Procedure:

### Source Code:

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

## Output:



```
≡ fourth.txt
1 fa g \n
2 \n fg af
3 ad \n
4 haf
5 adf ad
6 fgv ff
7 g dffg
8 fsg
9 afg
```

≡ fourth\_output.txt

```
1 1: fa g \n
2 2: \n fg af
3 3: ad \n
4 4: haf
5 5: adf ad
6 6: fgv ff
7 7: g dffg
8 8: fsg
9 9: afg
```

**Aim5:** Write a Lex program to printout all markup tags and HTML comments in file.

Procedure:

**Source Code:**

```
%{
#include<stdio.h>
int num=0;
%}
%%
"<"[A-Za-z0-9]+>" printf("%s is a valid markup tag\n", yytext);
"<!--"[^-->]*"-->" num++;
\n ;
. ;
%%
int main()
{
    yyin = fopen("fifth.txt", "r");
    yylex();
    printf("Number of comments are: %d", num);
    return 0;
}
int yywrap(){return (1);}
```

## Output:

≡ fifth.txt

```
1  <html>
2  <head>
3  </head>
4  <!-- this is html comment -->
5  </html>
6
```

```
<html> is a valid markup tag
<head> is a valid markup tag
Number of comments are: 1
```



## Practical-5

**Aim1:** 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.

### Procedure:

#### Source Code:

```
%{
#include<stdio.h>
int cmt = 0;
%}
%%
"/".* { fprintf(yyout, "\n"); cmt++;}
"/*"([^\]|+[^/])"*/" { fprintf(yyout, "\n"); cmt++;}
.\n { fprintf(yyout, "%s", yytext);}
%%
void main(){
    yyin = fopen("input1.txt", "r");
    yyout = fopen("output1.txt", "w");
    yylex();
    printf("%d Comment ", cmt);
}
int yywrap(){return(1);}
```

### Output:

```
≡ input1.txt
1
2 // This is the single line comment
3 void main(){
4
5 }
```

**Aim2:** Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

#### Procedure:

#### Source Code:

## second.l

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

int is_keyword(const char *str);
%}

%option noyywrap

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

[ \t\n]+          ;

"=="|"!="|"<="|">="|"="|"+"| "-"|"*"|" "/"|"%"|"&&"|"||"|"!"|"<"|">"    { printf("Operator:
%s\n", yytext); }

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

\"([^\"]|\\.)*\\"    { printf("String Literal: %s\n", yytext); }

\\.\'              { printf("Character Literal: %s\n", yytext); }

[{}()\[\];,:.]      { printf("Special Symbol: %s\n", yytext); }

[a-zA-Z_][a-zA-Z0-9_]*    {
    if (is_keyword(yytext))
        printf("Keyword: %s\n", yytext);
    else
        printf("Identifier: %s\n", yytext);
}

.                    { printf("Unrecognized Character: %s\n", yytext); }
```

%%

```
int is_keyword(const char *str) {
    const char *keywords[] = {
        "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
        "else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long",
        "register", "return", "short", "signed", "sizeof", "static", "struct", "switch",
        "typedef", "union", "unsigned", "void", "volatile", "while", NULL
    };
    for (int i = 0; keywords[i] != NULL; i++) {
        if (strcmp(keywords[i], str) == 0)
            return 1;
    }
    return 0;
}

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

### Output:

```
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Identifier: x
Operator: =
Number: 10.5
Special Symbol: ;
Identifier: ch
Operator: =
Character Literal: 'a'
Special Symbol: ;
Special Symbol: (
Identifier: x
Operator: >
Number: 5
Special Symbol: )
Special Symbol: {
Identifier: x
Operator: +
Operator: =
Number: 2
Special Symbol: ;
Special Symbol: }
Identifier: printf
Special Symbol: (
String Literal: "Hello World\n"
Special Symbol: )
Special Symbol: ;
Number: 0
Special Symbol: ;
Special Symbol: }
```

## Practical-6

**Aim1:** Program to implement Recursive Descent Parsing in C.

Procedure: Source

Code:

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

char s[20];
int i = 1;
char l;
int match(char l);
int E1();
int E()
{
    if (l == 'i')
    {
        match('i');
        E1();
    }
    else
    {
        printf("Error parsing string");
        exit(1);
    }
    return 0;
}

int E1()
{
    if (l == '+')
    {
        match('+');
        match('i');
        E1();
    }
    else
    {
        return 0;
    }
}
```

```
int match(char t)
```

```

{
    if (l == t)
    {
        l = s[i];
        i++;
    }
    else
    {
        printf("Syntax Error");
        exit(1);
    }
    return 0;
}
void main()
{
    printf("Enter the string: ");
    scanf("%s", &s);
    l = s[0];
    E();
    if (l == '$')
    {
        printf("parsing successful");
    }
    else
    {
        printf("Error while parsing the string\n");
    }
}

```

**Output:**

## Practical-7

**Aim1:** To Study about Yet Another Compiler-Compiler(YACC).

### Procedure:

YACC, which stands for Yet Another Compiler-Compiler, is a tool that helps us write the part of a compiler that checks if a program follows the correct structure or grammar. Writing this part manually can be pretty complex, but YACC makes it much easier by letting us define grammar rules, and then it automatically creates the code to check those rules. It's often used with another tool called Lex, which reads the input and splits it into meaningful pieces called tokens — like numbers, symbols, or keywords.

In this experiment, we use YACC to build a small program that can read and evaluate simple math expressions like “2 + 3 \* 4”. First, we define the rules that describe how such expressions should be written. YACC takes those rules and creates a parser, which can then figure out if a given input is valid and even calculate the result. This experiment helped me understand how a compiler breaks down and understands code, and how Lex and YACC work together to make that process easier.

**Aim2:** Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, \* and / .

### Procedure:

#### Source Code:

##### first.l

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "b.tab.h"
%}
%%
[0-9]+ {yylval = atoi(yytext); return NUM;}
[a-zA-Z_][a-zA-Z_0-9]* {return id;}
[-+*\n] {return *yytext;}
[ \t] { }
. yyerror("invalid character");
%%
int yywrap() {
```

```
    return 0;
}
```

first.y

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token NUM
%token id
%%
S: E '\n' { printf("valid syntax"); return(0); }
E: E '+' T { }
  | E '-' T { }
  | T      { }
T: T '*' F { }
  | F      { }
F: NUM     { }
  | id     { }
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
    return 0;
}
```

Output:



**Aim3:** Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.

**Procedure:**

**Source Code:**

**lex.l**

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

yacc.y

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token NUM
%%
S: E '\n' { printf("%d\n", $1); return(0); }
E: E '+' T { $$ = $1 + $3; }
  | E '-' T { $$ = $1 - $3; }
  | T      { $$ = $1; }
T: T '*' F { $$ = $1 * $3; }
  | F      { $$ = $1; }
F: NUM    { $$ = $1; }
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
```

```
yyparse();
```

```
    return 0;
}
```

Output:

**Aim4:** Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

**Procedure:**

**Source Code:**

**fourth.l**

```
%{
#include<stdio.h>
#include "first.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;
}
```

**fourth.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:**

```

256+65+24-5
256 65 + 24 + 5 -

```