SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

|  |
| --- |
| Laboratory Manual |

Enrollment no. : 22000916

Name: Kairavi Shah

Btech CSE – 6TH SEM

**TABLE OF CONTENT**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Experiment Title** | | | |
|  | | |
| 1 |  | 1. Write a program to recognize strings starts with ‘a’ over {a, b}. 2. Write a program to recognize strings end with ‘a’. 3. Write a program to recognize strings end with ‘ab’. Take the input from text file. 4. Write a program to recognize strings contains ‘ab’. Take the input from text file. |
| 2 |  | 1. Write a program to recognize the valid identifiers and keywords. 2. Write a program to recognize the valid operators. 3. Write a program to recognize the valid number. 4. Write a program to recognize the valid comments. 5. 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. |

# Program 1

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

## Program 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 3:

            state=3;

        }

        i++;

    }

    if(state==1) printf("Input Is Valid");

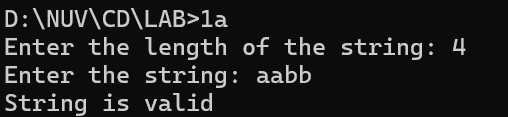
    else if(state==2||state==0) printf("Input is Not Valid");

    else if(state==3) printf("String Is Not Recognized");

    return 0;

}

## Output:



1. Aim: Write a program to recognize strings end with ‘a’.

## Program 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)

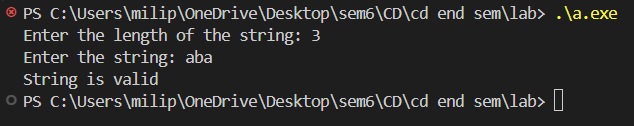
    {

        printf("String is valid");

    }

}

## Output:



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

## Program Code:

#include <stdio.h>

void main()

{

    int state = 0, i = 0;

    FILE \*fptr;

    fptr = fopen("1c.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");

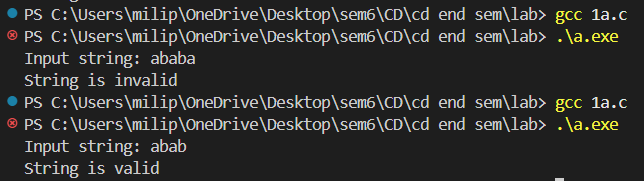
    }

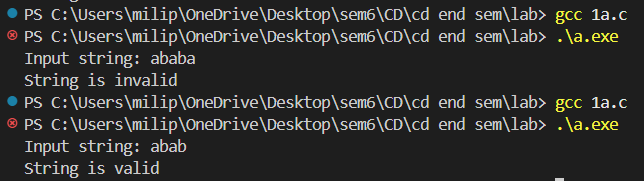
}

A black screen with white text

AI-generated content may be incorrect.

## Output:





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

## Program Code:

#include <stdio.h>

void main()

{

    int state = 0, i = 0;

    FILE \*fptr;

    fptr = fopen("1c.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");

    }

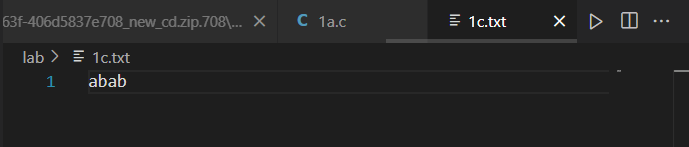
    else

    {

        printf("\nString is invalid");

    }

}



## Output:

A black screen with white text

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

# Program 2

1. Aim: Write a program to recognize the valid identifiers and keywords.

## Program Code:

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

void main()

{

    int state = 0, i = 0;

    FILE \*fptr;

    fptr = fopen("input.txt", "r");

    char input[100];

    while (fgets(input, 100, fptr) != NULL)

    {

        printf("Input string: %s", input);

        i = 0;

        state = 0;

        while (input[i] != '\0' && input[i] != '\n')

        {

            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("\nString is Identifier\n");

        }

        else if (state == 3)

        {

            printf("\nString is a valid Keyword\n");

        }

        else if (state == 5)

        {

            printf("\nString is invalid string\n");

        }

        else

        {

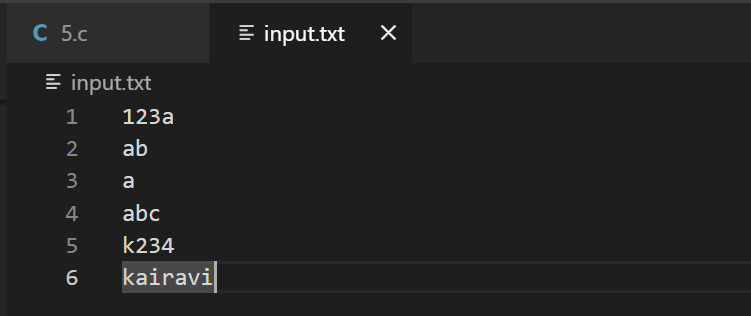
            printf("\nString is empty\n");

        }

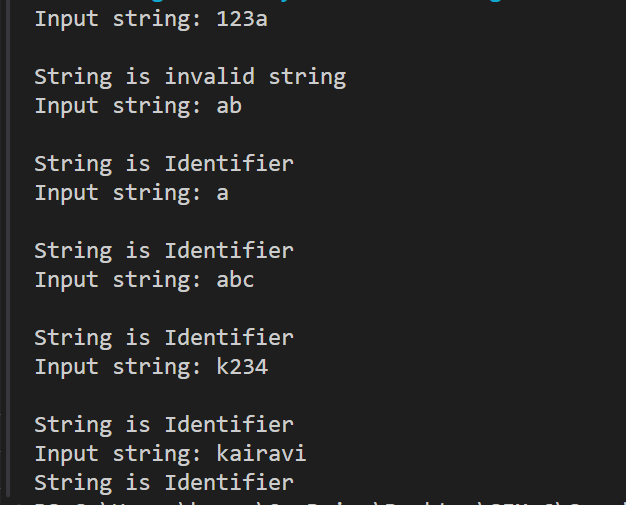
    }

    fclose(fptr);

}



## Output:

****

1. Aim: Write a program to recognize the valid operators.

## Program Code:

#include <stdio.h>

int main()

{

    char input[100];

    int state = 0, i = 0;

    FILE \*file = fopen("1c.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("!= Relational operator");

            }

            break;

        case 36:

            if (input[i] == '=')

            {

                state = 37;

                printf("== Relational operator");

            }

            break;

        default:

            break;

        }

        i++;

    }

    printf("\nState is %d\n", state);

    switch (state)

    {

    case 1:

        break;

    case 5:

        printf("- arithmetic operator\n");

        break;

    case 9:

        printf("\* arithmetic operator\n");

        break;

    case 12:

        printf("/ arithmetic operator\n");

        break;

    case 15:

        printf("%% arithmetic operator\n");

        break;

    case 18:

        printf("& Bitwise operator\n");

        break;

    case 21:

        printf("| Bitwise operator\n");

        break;

    case 24:

        printf("< Relational operator\n");

        break;

    case 28:

        printf("> Relational operator\n");

        break;

    case 32:

        printf("! Logical operator\n");

        break;

    case 34:

        printf("~ Bitwise operator\n");

        break;

    case 35:

        printf("^ Bitwise operator\n");

        break;

    case 36:

        printf("= Assignment operator\n");

        break;

    }

    return 0;

}

A screenshot of a computer

AI-generated content may be incorrect.

## Output:

A black screen with white text

AI-generated content may be incorrect.

1. Aim: Write a program to recognize the valid number.

## Program Code:

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string.h>

int main() {

    char c, buffer[1000], lexeme[1000];

    int i = 0, state = 0, f = 0, j = 0;

    FILE \*fp = fopen("1c.txt", "r");

    if (fp == NULL) {

        printf("Error opening file.\n");

        return 1;

    }

    // Read file content into buffer

    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; // Allow leading + or -

                    lexeme[f++] = c;

                } else if (isspace(c)) {

                    // Skip whitespace

                } else {

                    state = 99; // Invalid character

                }

                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--; // Re-check current character

                }

                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;

            case 99:

                printf("Invalid character encountered: %c\n", c);

                f = 0;

                state = 0;

                break;

        }

        i++;

    }

    // Final token check

    if (f != 0) {

        lexeme[f] = '\0';

        if (state == 1) {

            printf("The input %s is a valid integer.\n", lexeme);

        } else if (state == 3) {

            printf("The input %s is a valid floating-point number.\n", lexeme);

        } else if (state == 6) {

            printf("The input %s is a valid scientific notation number.\n", lexeme);

        }

    }

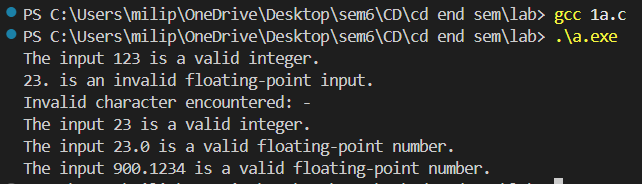
    return 0;

}

A screenshot of a computer

AI-generated content may be incorrect.

## Output:

****

1. Aim: Write a program to recognize the valid comments.

## Program Code:

#include <stdio.h>

int main() {

    int state = 0, i = 0;

    FILE \*fptr;

    char input[100];

    fptr = fopen("input.txt", "r"); // Read input from the file

    if (fptr == NULL) {

        printf("Error opening file.\n");

        return 1;

    }

    fgets(input, 100, fptr); // Read one line

    fclose(fptr);

    printf("Input string: %s", input);

    // State machine to detect comments

    while (input[i] != '\0') {

        switch (state) {

            case 0:

                if (input[i] == '/')

                    state = 1;

                else

                    state = 3; // Not a comment

                break;

            case 1:

                if (input[i] == '/')

                    state = 2; // Single-line comment

                else if (input[i] == '\*')

                    state = 4; // Start of multi-line comment

                else

                    state = 3;

                break;

            case 2:

                // Single-line comment till end of string

                break;

            case 4:

                if (input[i] == '\*')

                    state = 5;

                break;

            case 5:

                if (input[i] == '/')

                    state = 6; // End of multi-line comment

                else if (input[i] != '\*')

                    state = 4; // Go back to content inside comment

                break;

            case 6:

                // Multi-line comment ended

                break;

            case 3:

                // Not a comment

                break;

        }

        i++;

    }

    // Determine if the comment is valid or invalid

    if (state == 2) {

        printf("\nString is a valid single-line comment.\n");

    } else if (state == 6) {

        printf("\nString is a valid multi-line comment.\n");

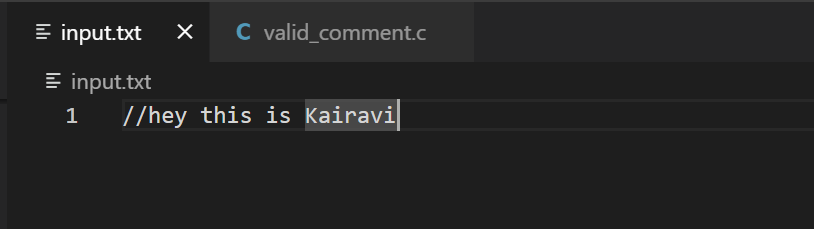
    } else {

        printf("\nString is an invalid comment.\n");

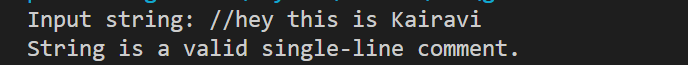
    }

    return 0;

}



## Output:

****

1. Aim: Program to implement Lexical Analyzer.

## Program Code:

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define BUFFER\_SIZE 1000

// Function declarations

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);

int main() {

    FILE \*f1;

    char \*buffer;

    char lexeme[50];

    char c;

    int i = 0, f = 0, state = 0;

    // Open file for reading

    f1 = fopen("1c.txt", "r");

    if (f1 == NULL) {

        printf("Error: Could not open 1c.txt\n");

        return 1;

    }

    // Read the file into memory

    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);

    // Start lexical analysis

    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);

                    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);

                    i = 0;

                    state = 0;

                    f--; // recheck 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--;

                }

                break;

            case 3:

                if (isdigit(c)) {

                    lexeme[i++] = c;

                } else {

                    lexeme[i] = '\0';

                    recognize\_number(lexeme);

                    i = 0;

                    state = 0;

                    f--;

                }

                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--;

                }

                break;

            case 5:

                if (isdigit(c)) {

                    lexeme[i++] = c;

                } else {

                    lexeme[i] = '\0';

                    recognize\_number(lexeme);

                    i = 0;

                    state = 0;

                    f--;

                }

                break;

        }

        f++;

    }

    // Free dynamically allocated memory

    free(buffer);

    return 0;

}

// Function to check if the lexeme is a keyword or identifier

void check\_keyword\_or\_identifier(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"

    };

    int is\_keyword = 0;

    for (int i = 0; i < 35; i++) {

        if (strcmp(lexeme, keywords[i]) == 0) {

            is\_keyword = 1;

            break;

        }

    }

    if (is\_keyword)

        printf("%s is a keyword\n", lexeme);

    else

        printf("%s is an identifier\n", lexeme);

}

// Function to recognize a number

void recognize\_number(char \*lexeme) {

    printf("%s is a valid number\n", lexeme);

}

// Function to recognize operators

void recognize\_operator(char \*buffer, int \*index) {

    char op[3] = {buffer[\*index], buffer[\*index + 1], '\0'};

    char \*operators[] = {"+", "-", "\*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">="};

    for (int i = 0; i < 12; i++) {

        if (strcmp(op, operators[i]) == 0) {

            printf("%s is an operator\n", op);

            (\*index)++;

            return;

        }

    }

    // Single-character operator

    printf("%c is an operator\n", buffer[\*index]);

}

// Function to recognize comments

void recognize\_comment(char \*buffer, int \*index) {

    if (buffer[\*index] == '/' && buffer[\*index + 1] == '/') {

        printf("// is a single-line comment\n");

        (\*index) += 2;

        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] == '/') {

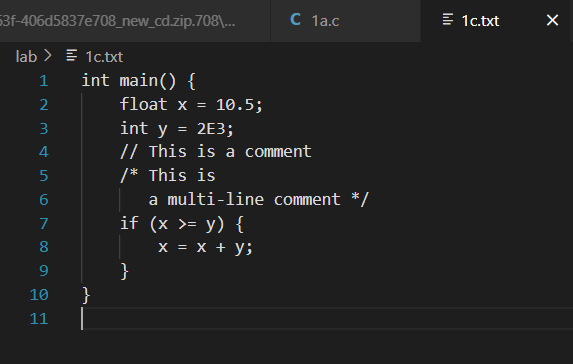
            printf("\*/ is the end of a multi-line comment\n");

            (\*index) += 2;

        }

    }

}



## Output:

A screenshot of a computer program

AI-generated content may be incorrect.

# Program 3

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

**What is a Lexical Analyzer Generator?**

A Lexical Analyzer Generator is a tool that automates the creation of a Lexical Analyzer (lexer), which is the first phase of a compiler. Its job is to scan the source code and convert it into a stream of tokens—the basic building blocks like keywords, operators, identifiers, etc.

**What is LEX?**

LEX (short for Lexical Analyzer Generator) was one of the earliest tools developed to help create lexical analyzers. It uses regular expressions to specify token patterns and generates C code that recognizes these patterns.

* Developed in the 1970s as part of the Unix toolchain.
* Usually used with Yacc (Yet Another Compiler Compiler) for syntax analysis.

**Structure of a LEX program:**

%{

// C declarations

%}

%%

[0-9]+ { printf("NUMBER\n"); }

[a-zA-Z]+ { printf("WORD\n"); }

. { printf("UNKNOWN\n"); }

%%

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

}

**What is Flex?**

Flex (Fast Lexical Analyzer Generator) is a free and faster alternative to LEX, and it is more commonly used today.

* Flex is open-source and generates more efficient code.
* It is backward-compatible with LEX but offers extra features and optimizations.
* It produces a C source file (e.g., lex.yy.c) which can be compiled with GCC.

**Advantages of Flex over LEX:**

A table of text on a white background

AI-generated content may be incorrect.

**How It Works**

1. Input: Regular expressions and actions (C code) for each pattern.

1. Processing: Generates a C program (lex.yy.c) that uses a finite state machine (DFA).

1. Output: The compiled lexer reads input, matches patterns, and executes associated actions.

**Use Cases**

* Programming language compilers (e.g., C, Python).
* Interpreters.
* Code analyzers or format checkers.
* Custom parsers in domain-specific languages.

# Program 4

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

## Program 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("1c.txt", "r"); yylex(); printf("Number of characters: %d \nNumber of lines: %d \nNumber of words: %d", c, l, w);

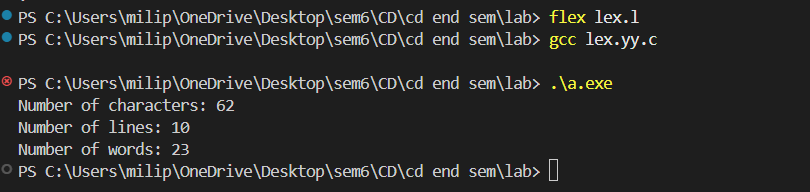
} int yywrap() { return (1);

}

A screenshot of a computer program

AI-generated content may be incorrect.

## Output:



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

## Program Code:

%{

#include<stdio.h>

int consonants = 0, vowels = 0;

%}

%%

[aeiouAEIOU] { vowels++; }

[a-zA-Z] { consonants++; }

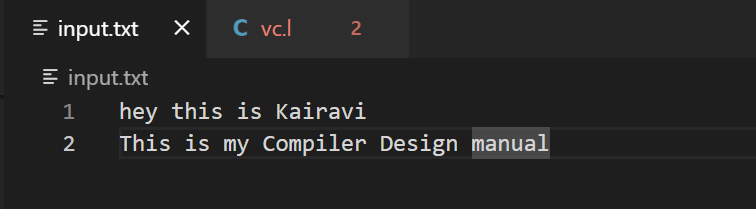
\n

.

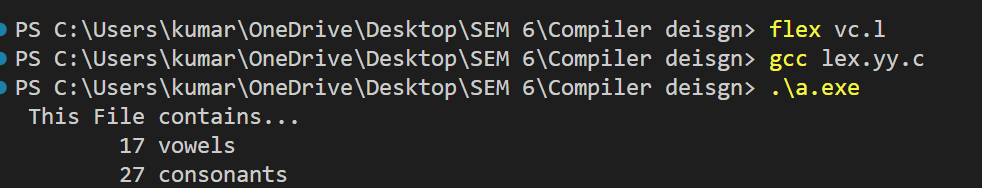
%%

int main() { yyin = fopen("input.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:



1. Aim: Write a Lex program to print out all numbers from the given file.

## Program Code:

%{

#include<stdio.h>

%}

%%

[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)? { printf("%s is a valid number \n", yytext);

}

\n ;

. ;

%%

int main() { yyin = fopen("1c.txt", "r"); yylex(); return 0; }

int yywrap() { return (1);

}

A black screen with a black text box

AI-generated content may be incorrect.

## Output:

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

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

## Program Code:

%{

#include<stdio.h>

int line\_number = 1;

%}

%%

.+ { fprintf(yyout, "%d: %s", line\_number, yytext); line\_number++;

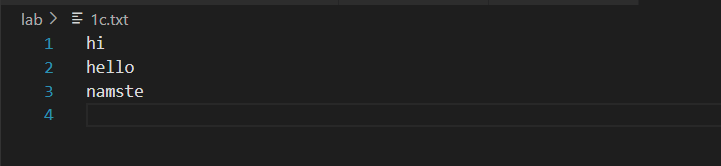
}

%%

int main() { yyin = fopen("1c.txt", "r"); yyout = fopen("fourth\_output.txt", "w"); yylex(); printf("done"); return 0; }

int yywrap() { return (1);

}



## Output:

A black background with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

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

## Program Code:

%{

#include <stdio.h>

int num = 0;

%}

%%

\<[A-Za-z0-9]+\>     { printf("%s is a valid markup tag\n", yytext); }

"<!--"([^<]|<[^!]|<^-]|<!--[^-])\*"-->"  { num++; }

\n                  ;  // ignore newlines

.                   ;  // ignore all other characters

%%

int main() {

    yyin = fopen("1c.txt", "r");

    if (!yyin) {

        perror("Cannot open file");

        return 1;

    }

    yylex();

    printf("Number of comments are: %d\n", num);

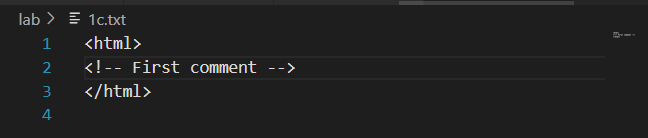
    return 0;

}

int yywrap() {

    return 1;

}



## Output:

![A black screen with white text

AI-generated content may be incorrect.

# Program 5

1. Aim: 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 files.

## Program Code:

%{

#include <stdio.h>

int cmt = 0;

%}

%%

\/\/[^\n]\*               { cmt++; }

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

.|\n                  { fprintf(yyout, "%s", yytext); }

%%

int main() {

    yyin = fopen("1c.txt", "r");

    yyout = fopen("output.txt", "w");

    if (!yyin || !yyout) {

        perror("File error");

        return 1;

    }

    yylex();

    printf("%d Comment(s)\n", cmt);

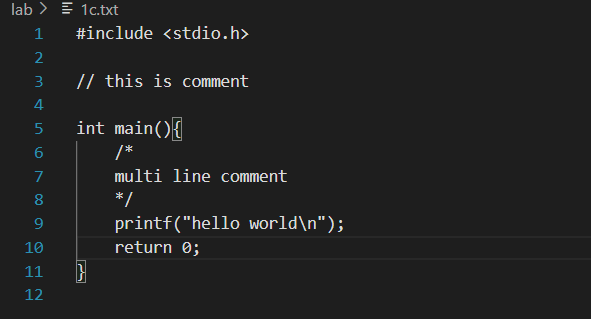
    return 0;

}

int yywrap() {

    return 1;

}



## Output:

A black background with white text

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

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

## Program Code:

%{

#include <stdio.h>

#include <string.h>

int is\_keyword(const char \*str);

// Declare yyin

extern FILE \*yyin;

%}

%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() {

    yyin = fopen("1c.txt", "r");

    if (!yyin) {

        perror("Failed to open 1c.txt");

        return 1;

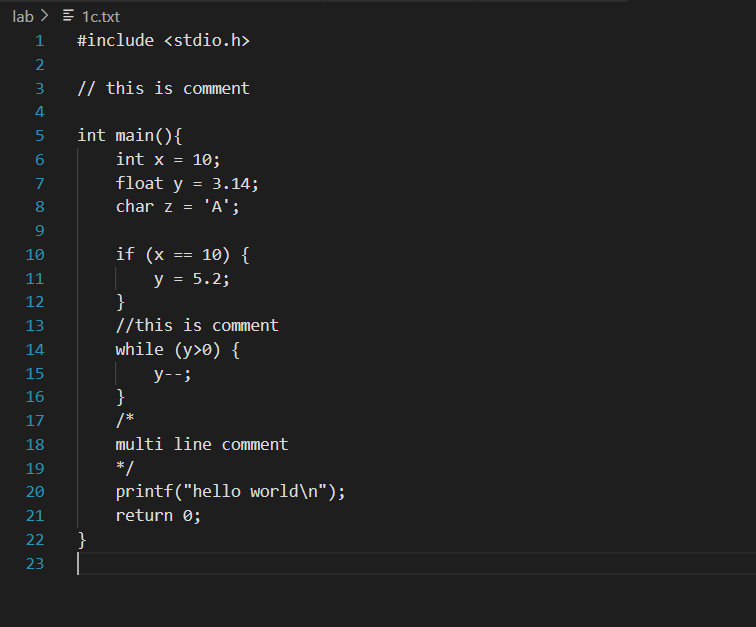
    }

    yylex();

    fclose(yyin);

    return 0;

}



## Output:

A screen shot of a computer program

AI-generated content may be incorrect.

# Program 6

1. Aim: Program to implement Recursive Descent Parsing in C.

## Program Code:

#include <stdio.h>

#include <stdlib.h>

char s[20];

int i=1;

char l;

int match(char t)

{

if(l==t){

l=s[i];

i++;

}

else{

printf("syntax error");

exit(1);

}

}

int E\_()

{

if(l=='+'){

match('+');

match('i');

E\_();

}

else if(l=='-'){

match('-');

match('i');

E\_();

}

else

return(1);

}

int E()

{

if (l=='i'){

match('i');

E\_();

}

}

int main()

{

printf("Enter set of character to parsed: ");

scanf("%s",&s);

l = s[0];

E();

if (l=='$'){

printf("\nSuccess\n");

}

else{

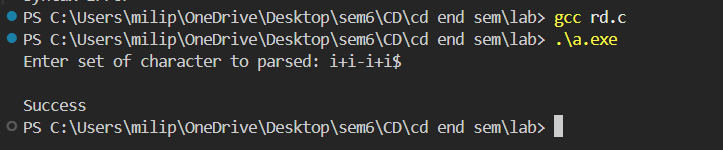
printf("Syntax Error");

}

return 0;

}

## Output:



# Program 7

1. Aim: To Study about Yet Another Compiler-Compiler (YACC).

**What is YACC?**

YACC (Yet Another Compiler-Compiler) is a tool used to generate parsers, which are part of the syntax analysis phase of a compiler. It takes a formal grammar (usually written in BNF- like syntax) and produces C code that can parse input sequences according to that grammar.

* Developed by Stephen C. Johnson in the 1970s at Bell Labs.
* Works closely with LEX/Flex, which handles lexical analysis (tokenization).
* YACC focuses on syntax parsing (checking structure of token sequences).

**Components of YACC**

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

%{

// C declarations

%}

%%

// Grammar rules with actions

%%

// Supporting C code (like main)

**How YACC Works**

1. Input: A context-free grammar (CFG) with actions (usually in C).
2. Output: A parser in C that uses LALR(1) parsing (Look-Ahead LR).
3. Integration: Uses token definitions from LEX (via yylex()).

Executes specific C actions when grammar rules match.

**Key Features**

A table of text on a white background

AI-generated content may be incorrect.

**Use Cases**

* Building compilers/interpreters
* Scripting languages
* Code validators or analyzers
* Structured data parsers (e.g., config files, DSLs)

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

## Program Code:

Lex file:

%{

#include <stdlib.h>

void yyerror(char \*);

#include "yacc.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; }

Yacc file:

%{

#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 { }

%%

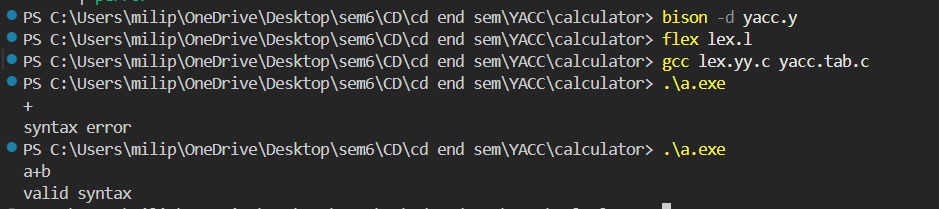
void yyerror(char \*s)

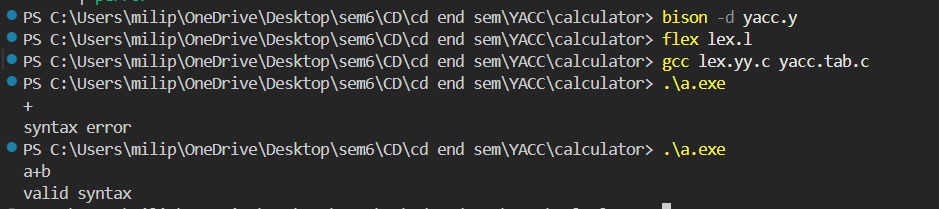
{ fprintf(stderr, "%s\n", s); }

int main()

{ yyparse(); return 0; }

## Output:





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

## Program Code:

Lex file

%{

#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 file

%{

 #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; }

  | T '/' F { $$ = $1 / $3; }

  | F     { $$ = $1; }

F : NUM { $$ = $1; }

%%

void yyerror(char \*s) {

 fprintf(stderr, "%s\n", s);

}

int main() {

 yyparse();return 0;

}

## Output:

A screen shot of a computer screen

AI-generated content may be incorrect.

1. Aim: Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

## Program Code:

Lex file:

%{

#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;

}

Yacc file:

%{

 #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:

