

SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE &
ENGINEERING

22000740 – VIDIT SHAH

Laboratory Manual

TABLE OF CONTENT

Sr. No	Experiment Title
1	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> a) Write a program to recognize the valid identifiers and keywords. 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	<p>Implement following programs using Lex.</p> <ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<p>a. To Study about Yet Another Compiler-Compiler(YACC).</p> <p>b. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .</p> <p>c. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.</p> <p>d. Create Yacc and Lex specification files are used to convert infix expression to postfix expression.</p>
---	--

Q1 a Write a program to recognize strings starts with 'a' over {a, b}.

Code:

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

```
int main(){  
    char input[10];  
    int i = 0;  
    printf("Enter input string to check in the automata: ");  
    scanf("%s", input);  
    int state = 0;
```

```
    while(input[i]!='\0'){  
        switch(state){  
            case 0:  
                if (input[i]=='a')  
                {  
                    /* code */  
                    state = 1;  
                }  
                else if (input[i] == 'b')  
                {  
                    /* code */  
                    state = 2;  
                }  
                else  
                {  
                    state = 3;  
                }  
                break;
```

```
case 1:
if (input[i] == 'a' || input[i] == 'b')
{
    /* code */
    state = 1;
}
else
{

    state = 3;
}
break;

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

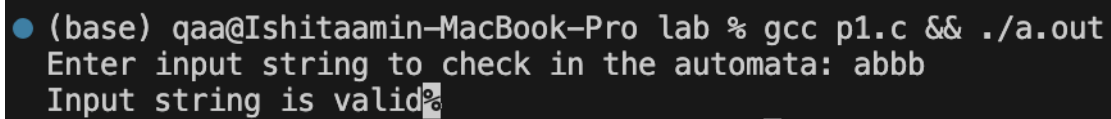
case 3:
state = 3;
}
```

```

        i++;
    }
    if(state == 1) printf("Input string is valid");
    else if(state == 2 || state == 0) printf("Input string is not valid");
    else if(state == 3) printf("String is not recogized");

    return 0;
}

```



```

(base) qaa@Ishitaamin-MacBook-Pro lab % gcc p1.c && ./a.out
Enter input string to check in the automata: abbb
Input string is valid%

```

Output:

Q1 b Write a program to recognize strings end 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;
}

```

```

(base) qaa@Ishitaamin-MacBook-Pro lab % gcc p2.c && ./a.out
Enter the input string: abbba
String is valid!%

```

Output:

Q1 c Write a program to recognize strings end with 'ab'. Take the input from text file.

Code:

```

#include <stdio.h>

#include <stdlib.h>

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

    // Open the file for reading

```

```

FILE *file = fopen("input.txt", "r");

if (file == NULL) {
    printf("Error: Could not open file.\n");
    return 1;
}

// Read the string from the file
fscanf(file, "%s", input);
fclose(file); // Close the file after reading

// DFA logic to check if the string ends with "ab"
while (input[i] != '\0') {
    switch (state) {
        case 0:
            if (input[i] == 'a') {
                state = 1;
            } else if (input[i] == 'b') {
                state = 0;
            } else {
                state = 3; // Invalid character
            }
            break;

        case 1:
            if (input[i] == 'b') {
                state = 2;
            } else if (input[i] == 'a') {
                state = 1;
            } else {
                state = 3;
            }
    }
}

```



```

    }

    break;

case 2:

    if (input[i] == 'a') {

        state = 1;

    } else if (input[i] == 'b') {

        state = 0;

    } else {

        state = 3;

    }

    break;

case 3:

    state = 3;

    break;

}

i++;

}

// Final check: only accept if last two characters were 'a' followed by 'b'

if (state == 2) {

    printf("Input string is valid (ends with 'ab')\n");

} else {

    printf("Input string is not valid (does not end with 'ab')\n");

}

return 0;

}

```

```
(base) qaa@Ishitaamin-MacBook-Pro lab % gcc 1c.c && ./a.out
Input string is not valid (does not end with 'ab')
(base) qaa@Ishitaamin-MacBook-Pro lab % gcc 1c.c && ./a.out
Input string is valid (ends with 'ab')
```

Output:

Q1 d Write a program to recognize strings contains 'ab'. Take the input from text file.

Code:

```
#include <stdio.h>

int main() {
    char input[10];
    int i = 0;
    printf("Enter input string to check in the automata: ");
    scanf("%s", input);
    int state = 0;

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

```

    } else {
        state = 3;
    }
    break;

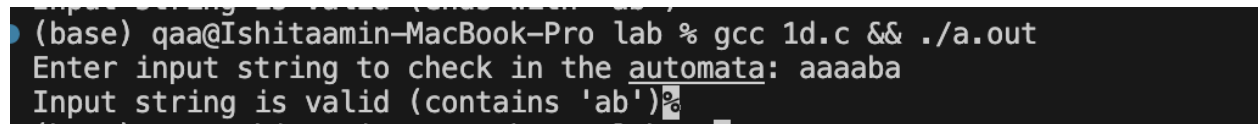
case 1:
    if (input[i] == 'b') {
        state = 2; // Transition to final state on "ab"
    } else if (input[i] == 'a') {
        state = 1;
    } else {
        state = 3;
    }
    break;
case 2:
    if (input[i] == 'a' || input[i] == 'b') {
        state = 2;
    } else {
        state = 3;
    }
    break;
case 3:
    state = 3;
    break;
}
i++;
}
if (state == 2) {
    printf("Input string is valid (contains 'ab')");
} else {

```

```

        printf("Input string is not valid (does not contain 'ab')");
    }
    return 0;
}

```



```

(base) qaa@Ishitaamin-MacBook-Pro lab % gcc 1d.c && ./a.out
Enter input string to check in the automata: aaaaba
Input string is valid (contains 'ab')%

```

Output:

2

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

Code:

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

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

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

```
int isKeyword(char *str) {
```

```
    char *keywords[] = {"int", "float", "char", "double", "return", "if", "else", "for", "while"};
```

```
    int numKeywords = sizeof(keywords) / sizeof(keywords[0]);
```

```
    for (int i = 0; i < numKeywords; i++) {
```

```
        if (strcmp(str, keywords[i]) == 0)
```

```
            return 1;
```

```
    }
```

```
    return 0;
```

```
}
```

```

int isValidIdentifier(char *str) {
    if (!isalpha(str[0]) && str[0] != '_')
        return 0;

    for (int i = 1; str[i] != '\0'; i++) {
        if (!isalnum(str[i]) && str[i] != '_')
            return 0;
    }

    return 1;
}

```

```

int main() {
    FILE *file;
    char filename[] = "input2a.txt";
    char word[100];
    int index = 0;
    char ch;

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

```

```

    printf("Results:\n");

```

```

    while ((ch = fgetc(file)) != EOF) {
        if (isalnum(ch) || ch == '_') {

```

```

        word[index++] = ch;
    } else if (index > 0) {
        word[index] = '\0';

        if (isKeyword(word))
            printf("%s' is a keyword\n", word);
        else if (isValidIdentifier(word))
            printf("%s' is a valid identifier\n", word);
        else
            printf("%s' is NOT a valid identifier\n", word);

        index = 0; // Reset for next word
    }
}

// Handle last word if file doesn't end with whitespace
if (index > 0) {
    word[index] = '\0';

    if (isKeyword(word))
        printf("%s' is a keyword\n", word);
    else if (isValidIdentifier(word))
        printf("%s' is a valid identifier\n", word);
    else
        printf("%s' is NOT a valid identifier\n", word);
}

fclose(file);
return 0;
}

```

```
(base) qaa@Ishitaamin-MacBook-Pro lab % gcc 2a.c && ./a.out
Results:
'int' is a keyword
'myVar' is a valid identifier
'_name' is a valid identifier
'2cool' is NOT a valid identifier
'valid_id' is a valid identifier
'for' is a keyword
'else' is a keyword
'MyVar' is a valid identifier
'float' is a keyword
(base) qaa@Ishitaamin-MacBook-Pro lab %
```

Output:

b) Write a program to recognize the valid operators.

Code:

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

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

```
int main() {
```

```
    char input[10];
```

```
    int i = 0;
```

```

printf("Enter input string to check in the automata: ");
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("Input string is a valid unary operator\n");
else if (state == 102 || state == 105 || state == 108 || state == 110 || state == 112)
    printf("Input string is a valid arithmetic operator\n");
else if (state == 119)
    printf("Input string is a valid relational operator\n");
else if (state == 101 || state == 104 || state == 107 || state == 109 || state == 111 || state == 120)
    printf("Input string is a valid assignment operator\n");
else
    printf("Invalid input\n");

```

```
    return 0;
}
```

```
● (base) qaa@Ishitaamin-MacBook-Pro lab % gcc operator.c && ./a.out
Enter input string to check in the automata: ++
Input string is a valid unary operator
```

Output:

c) Write a program to recognize the valid number.

Code:

```
#include <stdio.h>

#include <ctype.h>

#include <string.h>

int main() {
    FILE *file;

    char filename[] = "number.txt";

    char ch;

    int state = 0;

    int valid_found = 0;

    char word[100];

    int index = 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)) {
            word[index++] = ch;

            state = 1;
        } else if (ch == ' ' || ch == '\n' || ch == '\t') {
            // ignore whitespace
        } else {
            index = 0;

            state = 0;
        }

        break;

    case 1:
        if (isdigit(ch)) {
            word[index++] = ch;

            state = 1;
        } else if (ch == '.') {
            word[index++] = ch;

            state = 2;
        } else if (ch == 'E' || ch == 'e') {
            word[index++] = ch;

            state = 4;
        } else {
            word[index] = '\0';

            printf("Valid number: %s\n", word);

            valid_found = 1;

            index = 0;

            state = 0;

            ungetc(ch, file); // put back the extra character

```

```
}  
break;
```

case 2:

```
if (isdigit(ch)) {  
    word[index++] = ch;  
    state = 3;  
} else {  
    index = 0;  
    state = 0;  
}  
break;
```

case 3:

```
if (isdigit(ch)) {  
    word[index++] = ch;  
    state = 3;  
} else if (ch == 'E' || ch == 'e') {  
    word[index++] = ch;  
    state = 4;  
} else {  
    word[index] = '\\0';  
    printf("Valid number: %s\\n", word);  
    valid_found = 1;  
    index = 0;  
    state = 0;  
    ungetc(ch, file);  
}  
break;
```

case 4:

```
if (ch == '+' || ch == '-') {  
    word[index++] = ch;  
    state = 5;  
} else if (isdigit(ch)) {  
    word[index++] = ch;  
    state = 6;  
} else {  
    index = 0;  
    state = 0;  
}  
break;
```

case 5:

```
if (isdigit(ch)) {  
    word[index++] = ch;  
    state = 6;  
} else {  
    index = 0;  
    state = 0;  
}  
break;
```

case 6:

```
if (isdigit(ch)) {  
    word[index++] = ch;  
    state = 6;  
} else {  
    word[index] = '\0';  
    printf("Valid number: %s\n", word);
```

```

        valid_found = 1;

        index = 0;

        state = 0;

        ungetc(ch, file);
    }

    break;
}

}

// Handle if file ends directly after a number
if ((state == 1 || state == 3 || state == 6) && index > 0) {
    word[index] = '\0';

    printf("Valid number: %s\n", word);

    valid_found = 1;
}

fclose(file);

if (!valid_found) {
    printf("No valid number found.\n");
}

return 0;
}

```

Output:

```
(base) qaa@Ishitaamin-MacBook-P  
Valid number: 123  
Valid number: 45.67  
Valid number: 10e3  
Valid number: 3.14e-2  
Valid number: 12.3e+10  
(base) qaa@Ishitaamin-MacBook-P
```

d) Write a program to recognize the valid comments.

Code:

// Write a program to the comment in the code

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

```
int main() {
```

```
    FILE *file;
```

```
    char filename[] = "comment.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(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;

    }

    i++;
}

if(state == 1) printf("Input string is invalid");
else if(state == 2 || state == 0) printf("Input string is not any comment");
else if(state == 3) printf("String is single line comment");
else if(state == 4 || state==5) printf("String is not a valid comment");
else if(state == 6) printf("String is multiline comment");

return 0;
}

```

```

(base) qaa@Ishitaamin-MacBook-Pro lab % gcc p3.c && ./a.out
String is multiline comment%
(base) qaa@Ishitaamin-MacBook-Pro lab %

```

Output:

e) Program to implement Lexical Analyzer.

Code:

```
// Program to implement Lexical Analyzer
```

```

#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);
}

```

```
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 0.5733 seconds with return value 0
Press any key to continue . . . |
```

}

Output:

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

A **Lexical Analyzer** (or scanner/tokenizer) is a part of a compiler that:

- Reads the input source code (text file)
- Breaks it into **tokens** (smallest meaningful units like keywords, identifiers, numbers, symbols, etc.)
- Removes white spaces and comments
- Sends tokens to the **parser** for syntax analysis

Structure of Lex

```
%{  
    // C declarations (optional)  
}%  
  
%%  
  
// Rules section  
pattern1  action1  
pattern2  action2  
  
%%  
  
// Additional C code (optional)
```

How lex works:

- Write a `.l` file with patterns and actions.
- Run `lex file.l` – it creates a C file `lex.yy.c`.
- Compile with `gcc lex.yy.c -lfl` (link with LEX/FLEX library).
- Run the output program with your input.

Flex -Flex stands for Fast Lexical Analyzer. It is an improved, faster version of LEX. Works similarly to LEX but is more flexible and efficient. Compatible with LEX programs.

4

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

%%

[^\\n\\t]+ { word_count++; char_count += yyleng; } // Count words + their characters
[\\n]      { line_count++; char_count++; }        // Count lines and newline characters
[ \\t]     { char_count++; }                      // Count spaces and tabs
.          { char_count++; }                      // Count other characters (punctuation etc.)

%%

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

```

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Error: Could not open input.txt\n");
        return 1;
    }

    yyin = file; // Set input source for Lex
    yylex();     // Start scanning

    fclose(file); // Close file after scanning

    // Print results
    printf("Total Characters: %d\n", char_count);
    printf("Total Words    : %d\n", word_count);
    printf("Total Lines    : %d\n", line_count);

    return 0;
}

```

Output:

```

● (base) qaa@Ishitaamin-MacBook-Pro 4a % flex count.l
count.l:14: warning, rule cannot be matched
● (base) qaa@Ishitaamin-MacBook-Pro 4a % gcc lex.yy.c -o scan
● (base) qaa@Ishitaamin-MacBook-Pro 4a % ./scan
Total Characters: 32
Total Words      : 6
Total Lines      : 2
○ (base) qaa@Ishitaamin-MacBook-Pro 4a % █

```

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

Code:

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

int countv = 0;
int countc = 0;
%}

%%
[aeiouAEIOU] { countv++; }
[bcd fghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] { countc++; }
. ; // Ignore all other characters
%%

int yywrap() {
    return 1;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Could not open input.txt\n");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    printf("Total Vowels   : %d\n", countv);
    printf("Total Consonants: %d\n", countc);
    return 0;
}
```

```

• (base) qaa@Ishitaamin-MacBook-Pro vowels % lex vowel.l
• (base) qaa@Ishitaamin-MacBook-Pro vowels % gcc lex.yy.c -o scan -ll
• (base) qaa@Ishitaamin-MacBook-Pro vowels % ./scan

```

```

Total Vowels      : 6
Total Consonants: 13

```

Output:

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

Code:

```

%{
#include <stdio.h>
%}

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

. ; // Ignore all other characters
%%

int yywrap() {
    return 1;
}

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

    yylex(); // Start scanning
    fclose(yyin);
    return 0;
}

```

Output:

```
(base) qaa@Ishitaamin-MacBook-Pro 4c % lex num.l
(base) qaa@Ishitaamin-MacBook-Pro 4c % gcc lex.yy.c -o scan -ll
(base) qaa@Ishitaamin-MacBook-Pro 4c % ./scan
Number found: 45.67
Number found: 100

Number found: 42
Number found: 3.14
```

```
> ≡ input.txt
The total is 45.67 dollars and 100 cents.
Ignore words, just 42 and 3.14 are enough.
```

d. Write a Lex program that

adds line numbers to the given file and displays the same into a 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);
```

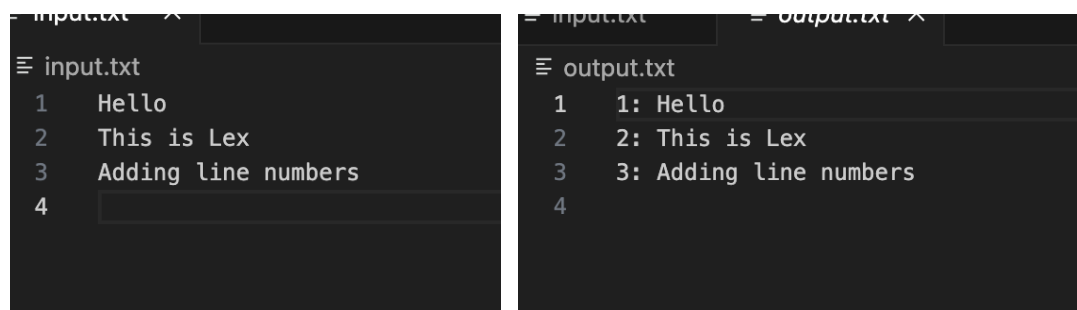
```
    printf("Line numbers added successfully to output.txt\n");  
    return 0;  
}
```

```

(base) qaa@Ishitaamin-MacBook-Pro 4d % flex code.l
(base) qaa@Ishitaamin-MacBook-Pro 4d % gcc lex.yy.c -o scan -ll
(base) qaa@Ishitaamin-MacBook-Pro 4d % ./scan
Line numbers added successfully to output.txt
(base) qaa@Ishitaamin-MacBook-Pro 4d %

```

Output:



```

input.txt
1 Hello
2 This is Lex
3 Adding line numbers
4

output.txt
1 1: Hello
2 2: This is Lex
3 3: Adding line numbers
4

```

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;
}

```

```

le > <> input.html > html > body
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  |   <meta charset="UTF-8">
5  |   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6  |   <title>Document</title>
7  </head>
8  <body>
9  |   <b>hi how are u </b>
10 |   <h1>23</h1>
11 |   <!-- hi how r u -->
12 </body>
13 </html>

```

Output:

```

● (base) qaa@Ishitaamin-MacBook-Pro 4e % flex code.l
● (base) qaa@Ishitaamin-MacBook-Pro 4e % gcc lex.yy.c -o scan -ll
● (base) qaa@Ishitaamin-MacBook-Pro 4e % ./scan
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>
○ (base) qaa@Ishitaamin-MacBook-Pro 4e % █

```

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 comment_count = 0;  
  
FILE *out;  
  
%}  
  
%%  
  
"//".*          { comment_count++; /* Skip single-line comment */ }  
  
"/*"([^\*]*|\*+[^*/])***"+"/*" { comment_count++; /* Skip multi-line comment */ }  
  
.|\\n          { fputc(yytext[0], out); } // Copy other content  
  
%%  
  
int yywrap() {  
    return 1;  
}  
  
int main() {  
    FILE *in = fopen("source.c", "r");  
    out = fopen("cleaned.c", "w");  
  
    if (!in || !out) {  
        printf("Error opening file(s)\n");  
        return 1;  
    }  
  
    yyin = in;
```

```

yylex();

fclose(in);

fclose(out);

printf("Total comments removed: %d\n", comment_count);
printf("Cleaned code written to 'cleaned.c'\n");

return 0;
}

```

```

• (base) qaa@Ishitaamin-MacBook-Pro lab % cd 5a
• (base) qaa@Ishitaamin-MacBook-Pro 5a % flex c.l
• (base) qaa@Ishitaamin-MacBook-Pro 5a % gcc lex.yy.c -o scan
• (base) qaa@Ishitaamin-MacBook-Pro 5a % ./scan
Total comments removed: 3
Cleaned code written to 'cleaned.c'
○ (base) qaa@Ishitaamin-MacBook-Pro 5a % █

```

Output:

```

C cleaned.c > ...
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}

```

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:

```

(base) qaa@Ishitaamin-MacBook-Pro lex % flex lex.l
(base) qaa@Ishitaamin-MacBook-Pro lex % gcc lex.yy.c -o scan
(base) qaa@Ishitaamin-MacBook-Pro lex % ./scan
Enter C code (Ctrl+D to end):

int x = 5;
if (x >= 10) {
    printf("Greater\n");[Token] Keyword          → int
[Token] Identifier      → x
[Token] Operator        → =
[Token] Number          → 5
[Token] Special Symbol  → ;
[Token] Keyword         → if
[Token] Special Symbol  → (
[Token] Identifier      → x
[Token] Operator        → >=
[Token] Number          → 10
[Token] Special Symbol  → )
[Token] Special Symbol  → {
}
[Token] Identifier      → printf
[Token] Special Symbol  → (
[Token] String Literal  → "Greater\n"
[Token] Special Symbol  → )
[Token] Special Symbol  → ;
[Token] Special Symbol  → }

```

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

// Forward declarations

int E();

int E_p();

int T();

int T_p();

int F();

int E() {
    T();
    return E_p();
}
```

```

int E_p() {
    if (input[pos] == '+') {
        match('+');
        T();
        return E_p();
    } else if (input[pos] == '-') {
        match('-');
        T();
        return E_p();
    }
    return 1; // epsilon
}

int T() {
    F();
    return T_p();
}

int T_p() {
    if (input[pos] == '*') {
        match('*');
        F();
        return T_p();
    } else if (input[pos] == '/') {
        match('/');
        F();
        return T_p();
    }
    return 1; // epsilon
}

```



```

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

int parse() {
    E();
    if (input[pos] == '\0') {
        printf("Parsed successfully.\n");
        return 1;
    } else {
        printf("Syntax Error: Unexpected characters at position %d\n", pos);
        exit(1);
    }
}

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

```

Output:

```

Parsed successfully.
(base) qaa@Ishitaamin-MacBook-Pro cd_lab %

```

7.

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

YACC stands for **Yet Another Compiler-Compiler**.

It is a **tool used to generate parsers**, especially **LALR(1)** parsers, for interpreting structured input (like programming languages).

It works alongside **Lex**, which handles **lexical analysis** (tokenizing), while YACC does **syntax analysis** (parsing based on grammar).

YACC helps:

- Convert high-level grammar into a parser automatically.
- Enforce the **syntax rules** of a programming language.
- Act as the middle step in building **interpreters or compilers**.

YACC takes:

- **Tokens** from a lexical analyzer (like Flex/Lex).
- **Grammar rules** written in a BNF-like format.
- **Action code** (usually in C) to execute when rules match.

It outputs:

- A `y.tab.c` file (C code of the parser).
- This parser calls `yylex()` (defined by Lex) to get tokens and applies grammar rules to parse them.

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

code:

Lex file :

```
%{
#include "sample.tab.h" // This header is auto-generated by Bison and includes token
definitions like NUM
}%

%%

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

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

Yacc file:

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

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

%token NUM

%%

S:
    E '\n' { printf("Valid expression\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:

```

(base) qaa@Ishitaamin-MacBook-Pro syntax % bison -d sample.y
(base) qaa@Ishitaamin-MacBook-Pro syntax % flex sample.l
(base) qaa@Ishitaamin-MacBook-Pro syntax % gcc sample.tab.c lex.yy.c -o scan
(base) qaa@Ishitaamin-MacBook-Pro syntax % ./scan
Enter expression: 2+3-4*/7
Error: syntax error
(base) qaa@Ishitaamin-MacBook-Pro syntax %

```

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

Code:

Flex file

```
%{  
  
#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 file

```
%{  
  
#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;  
}
```

```
(base) qaa@Ishitaamin-MacBook-Pro calculator % bison -d calc.y  
(base) qaa@Ishitaamin-MacBook-Pro calculator % flex calc.l  
(base) qaa@Ishitaamin-MacBook-Pro calculator % gcc calc.tab.c lex.yy.c -o calc  
(base) qaa@Ishitaamin-MacBook-Pro calculator % ./calc  
Enter an arithmetic expression:  
2+3/4-1*3  
Result = -1  
(base) qaa@Ishitaamin-MacBook-Pro calculator %
```

Output:

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

Code:

Flex file

```
%{  
  
#include "infix.tab.h"  
  
%}  
  
%%  
  
[0-9]+ { yylval.intval = atoi(yytext); return NUMBER; }  
[ \t] ; /* skip whitespace */  
\n return '\n';  
[-+*/%()] return yytext[0];  
.  
{ printf("Invalid character"); return 0; }  
%%  
  
int yywrap() {  
    return 1;  
}
```

Bison file

```
%{  
  
#include <stdio.h>  
#include <string.h>  
  
int yylex(void);  
void yyerror(const char *s);  
  
char postfix[1024];  
  
%}  
  
%union {
```



```
int intval;  
}
```

```
%token <intval> NUMBER
```

```
%left '+' '-'
```

```
%left '*' '/' '%'
```

```
%type <intval> expr term factor
```

```
%%
```

```
input:
```

```
    | input expr '\n' { printf("Postfix: %s\n", postfix); postfix[0] = '\0'; }  
    ;
```

```
expr:
```

```
    term          { $$ = $1; }  
    | expr '+' term { strcat(postfix, "+ "); }  
    | expr '-' term { strcat(postfix, "- "); }  
    ;
```

```
term:
```

```
    factor        { $$ = $1; }  
    | term '*' factor { strcat(postfix, "* "); }  
    | term '/' factor { strcat(postfix, "/ "); }  
    | term '%' factor { strcat(postfix, "% "); }  
    ;
```

```
factor:
```

```
    NUMBER        { char num[20]; sprintf(num, "%d ", $1); strcat(postfix, num); $$ = $1; }
```

```

    | '(' expr ')' { $$ = $2; }

;

%%

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

int main() {
    printf("Enter expression: ");
    yyparse();
    return 0;
}

```

Output:

```

● (base) qaa@Ishitaamin-MacBook-Pro infix % bison -d infix.y
● (base) qaa@Ishitaamin-MacBook-Pro infix % flex infix.l
● (base) qaa@Ishitaamin-MacBook-Pro infix % gcc infix.tab.c lex.yy.c -o infix
○ (base) qaa@Ishitaamin-MacBook-Pro infix % ./infix
Enter expression: 2+3-6/3*4
Postfix: 2 3 + 6 3 / 4 * -

```