

LAB MANUAL
of
Compiler Design Laboratory
(CS605)
Bachelor of Technology (CSE)
By
Jeet Rathod (22000944)
Third Year, Semester 6
Course In-charge: Prof. Vaibhavi Patel



Department of Computer Science and Engineering
School of Engineering and Technology

Navrachana University,
Vadodara (2024-2025)

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 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 | | 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

- a. 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:

```
Enter the length of the string: 4
Enter the string: aabb
String is valid
```

- b. **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:

```
Enter the length of the string: 3  
Enter the string: aba  
String is valid
```

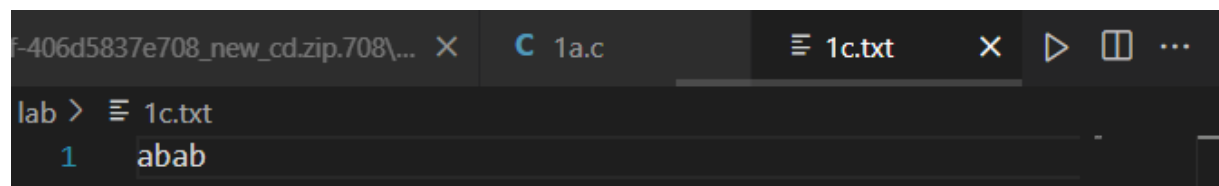
- c. **Aim:** Write a program to recognize strings end with 'ab'. Take the input from text file.

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

```
        {
            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");
}
}
```



The screenshot shows a code editor window with a tab labeled '1a.c'. The editor content shows a prompt 'lab >' followed by the input 'abab' on line 1.

Output:

```
Input string: ababa
String is invalid
```

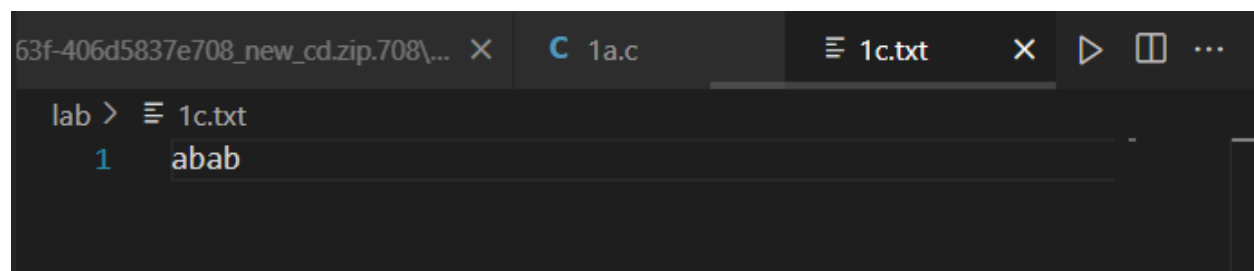
```
Input string: abab
String is valid
```

- d. **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");
    }
}
```



The screenshot shows a code editor window with a tab labeled '1c.txt'. The editor content shows a prompt 'lab >' followed by the text 'abab' on a new line.

Output:

```
Input string: abab
String is valid
```

```
Input string: bbba
String is invalid
```

Program 2

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

Program Code:

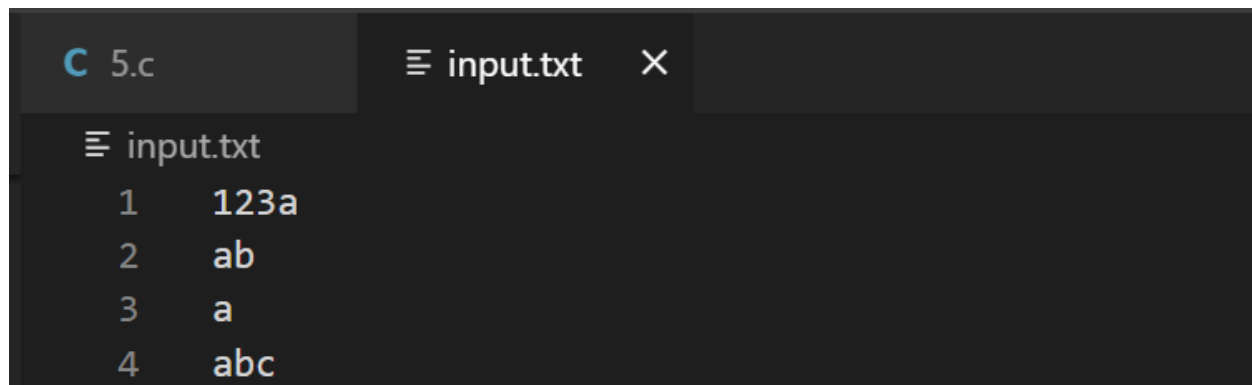
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
void main()
{
    int state = 0, i = 0;
    FILE *fptr;
    fptr = fopen("input.txt", "r");
    char input[100];
    while (fgets(input, 100, fptr) != NULL)
```



```
{
    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;
                }
            }
```

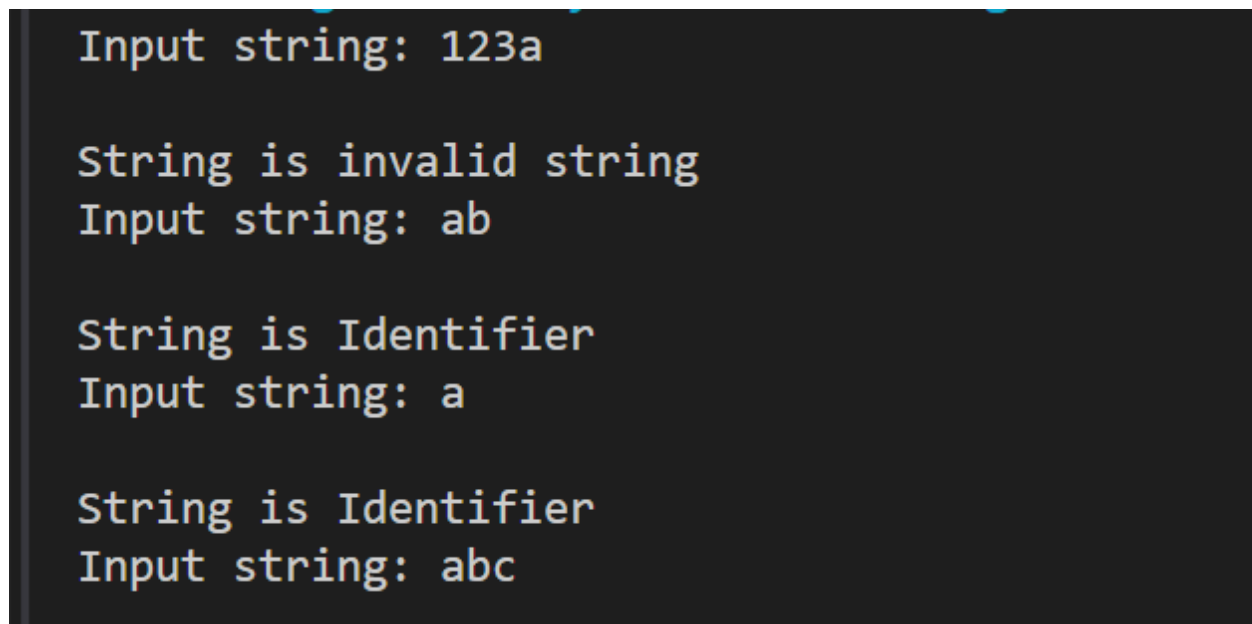
```
        else
        {
            state = 5;
        }
        break;
    case 3:
        if (isalpha(input[i]) || isdigit(input[i]) || input[i] == '_')
        {
            state = 4;
        }
        else
        {
            state = 5;
        }
        break;
    case 4:
        if (isalpha(input[i]) || 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);  
}
```



```
C 5.c  input.txt X  
input.txt  
1 123a  
2 ab  
3 a  
4 abc
```

Output:



```
Input string: 123a  
  
String is invalid string  
Input string: ab  
  
String is Identifier  
Input string: a  
  
String is Identifier  
Input string: abc
```

- b. 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] == '=')
```

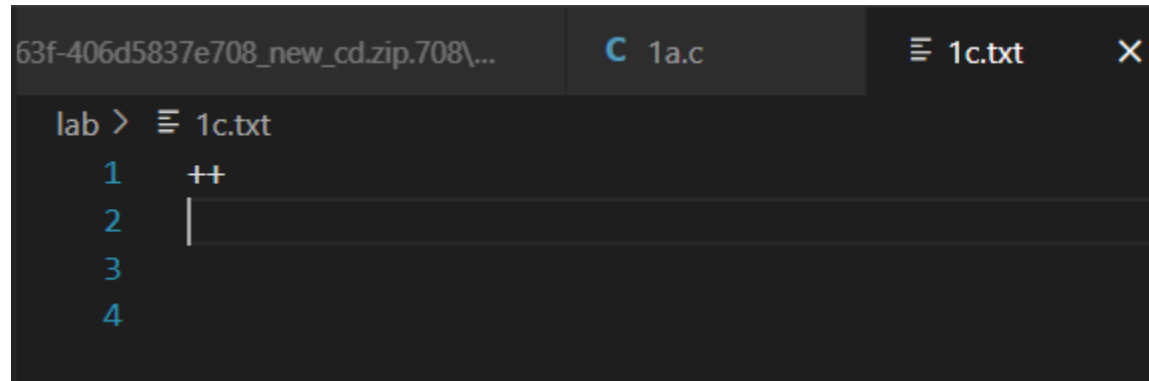
```
        {
            state = 13;
            printf("/= assignment operator");
        }
        else
        {
            state = 14;
            printf("/ arithmetic operator");
        }
        break;
    case 15:
        if (input[i] == '=')
        {
            state = 16;
            printf("%%= 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;
}
```



The screenshot shows a terminal window with a dark background. At the top, there are tabs for files: '63f-406d5837e708_new_cd.zip.708\...', '1a.c', and '1c.txt'. The terminal prompt is 'lab >'. The user has entered '1c.txt', and the file's content is displayed: '++ unary operator' on the first line and 'State is 2' on the second line.

Output:

```
++ unary operator
State is 2
```

c. 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;
}
```

```
63f-406d5837e708_new_cd.zip.708\... X C 1a.c 1c.txt X
lab > 1c.txt
1 123
2 23.e-23
3 23.0
4 900.1234
5
```

Output:

```
The input 123 is a valid integer.
23. is an invalid floating-point input.
Invalid character encountered: -
The input 23 is a valid integer.
The input 23.0 is a valid floating-point number.
The input 900.1234 is a valid floating-point number.
```

d. 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;

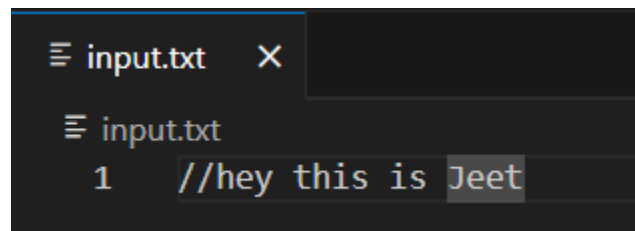
    case 3:
        // Not a comment
        break;
}
i++;
}
```



```
// Determine if the comment is valid or invalid
if (state == 2) {
```

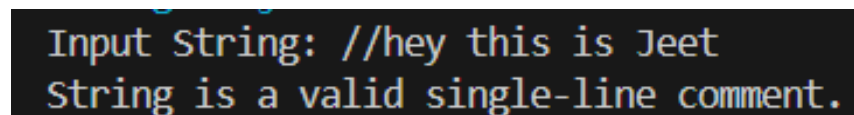
```
    printf("\nString is a valid single-line comment.\n");
} else if (state == 6) {
    printf("\nString is a valid multi-line comment.\n");
} else {
    printf("\nString is an invalid comment.\n");
}

return 0;
}
```



The screenshot shows a text editor window with a tab labeled 'input.txt'. The file contains a single line of text: '//hey this is Jeet'. The text is highlighted in a light blue color.

Output:



The screenshot shows the output of the program. It displays the input string '//hey this is Jeet' and the result of the lexical analysis, which is 'String is a valid single-line comment.'.

e. 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);
}

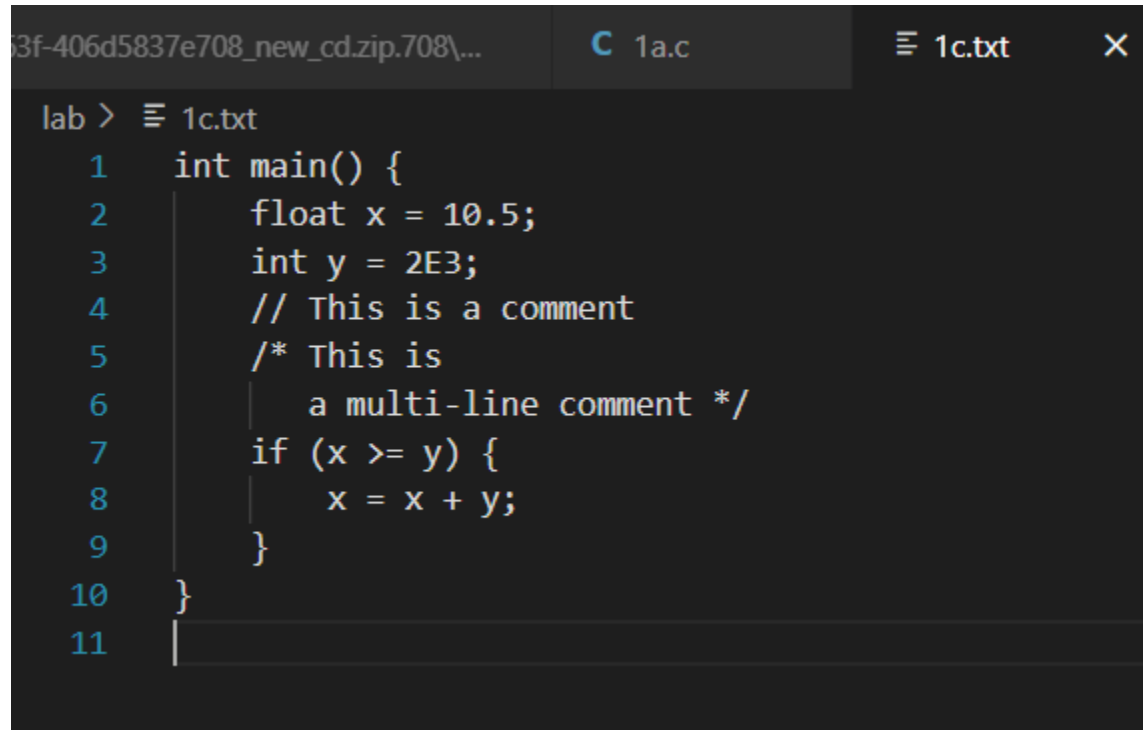
// Function to recognize operators
void recognize_operator(char *buffer, int *index) {
    char op[3] = {buffer[*index], buffer[*index + 1], '\0'};
    char *operators[] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">",
"<=", ">="};

    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;  
    }  
}  
}
```



The screenshot shows a code editor with a dark theme. The top bar displays the file path '53f-406d5837e708_new_cd.zip.708\...' and two tabs: '1a.c' (active) and '1c.txt'. The editor content shows a C program with line numbers 1 through 11. The code includes a multi-line comment and an if statement.

```
lab > 1c.txt  
1  int main() {  
2      float x = 10.5;  
3      int y = 2E3;  
4      // This is a comment  
5      /* This is  
6         a multi-line comment */  
7      if (x >= y) {  
8          x = x + y;  
9      }  
10 }  
11 |
```

Output:

```
int is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
float is a keyword
x is an identifier
= is an operator
10.5 is a valid number
; is a symbol
int is a keyword
y is an identifier
= is an operator
2E3 is a valid number
; is a symbol
// is a single-line comment
/* is the start of a multi-line comment
*/ is the end of a multi-line comment
if is a keyword
( is a symbol
x is an identifier
>= is an operator
y is an identifier
) is a symbol
{ is a symbol
x is an identifier
= is an operator
x is an identifier
+ is an operator
y is an identifier
; is a symbol
} is a symbol
} is a symbol
```

Program 3

- a. 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:

| Feature | LEX | Flex |
|-------------|----------------|------------------------|
| Speed | Slower | Faster |
| Portability | Unix-only | Cross-platform (POSIX) |
| Extensions | Limited | More modern extensions |
| Output Code | Less optimized | Highly optimized |

How It Works

1. Input: Regular expressions and actions (C code) for each pattern.
2. Processing: Generates a C program (`lex.yy.c`) that uses a finite state machine (DFA).
3. Output: The compiled lexer reads input, matches patterns, and executes associated actions.

Use Cases

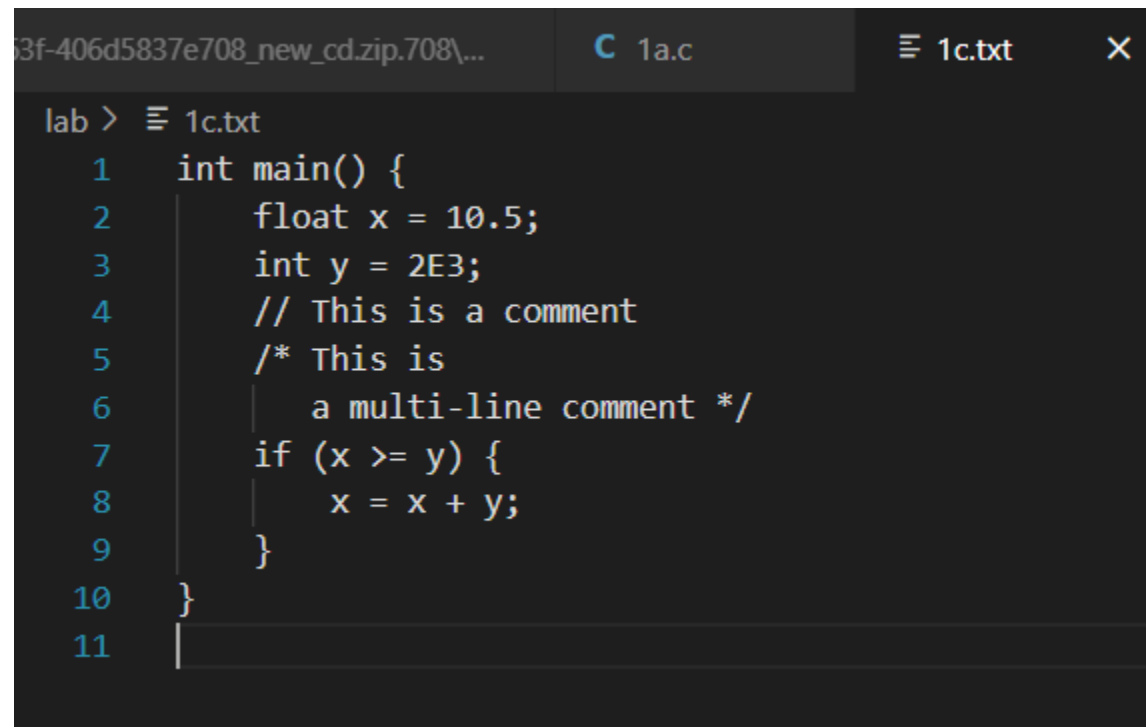
- Programming language compilers (e.g., C, Python).
- Interpreters.
- Code analyzers or format checkers.
- Custom parsers in domain-specific languages.

Program 4

- a. 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);
}
```



```
lab > 1c.txt
1  int main() {
2      float x = 10.5;
3      int y = 2E3;
4      // This is a comment
5      /* This is
6         a multi-line comment */
7      if (x >= y) {
8          x = x + y;
9      }
10 }
11
```

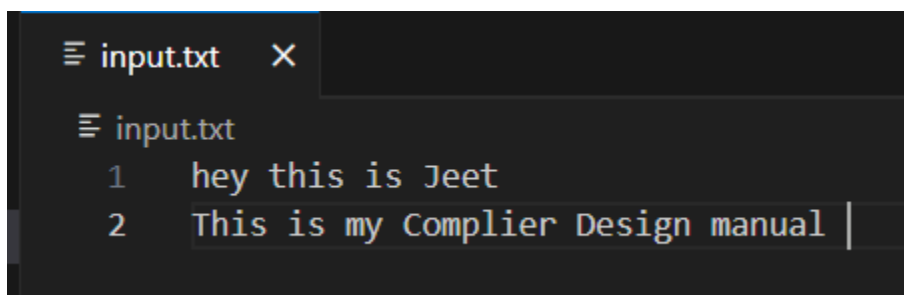

Output:

```
Number of characters: 62
Number of lines: 10
Number of words: 23
```

- b. **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); }
```



```
input.txt X
input.txt
1 hey this is Jeet
2 This is my Compiler Design manual |
```

Output:

```
This file contains...
    15 vowels
    25 consonansts
```

- c. **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);
}
```

```
lab > ≡ 1c.txt
1    123
2    23
3    -2
```

Output:

```
123 is a valid number
23 is a valid number
2 is a valid number
14 is a valid number
```

```
123 is a valid number
23 is a valid number
2 is a valid number
```

- d. **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);  
}
```

```
lab > ≡ 1c.txt  
1 hi  
2 hello  
3 namste  
4
```

Output:

```
done
```

```
lab > ≡ fourth_output.txt  
1 1: hi  
2 2: hello  
3 3: namste  
4
```

- e. **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;
}
```

```
lab > ≡ 1c.txt
1  <html>
2  <!-- First comment -->
3  </html>
4
```

Output:

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

Program 5

- a. 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++; }

\\\/*([^\n]|\\\/)*\\\/  { 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;
}
```

```
lab > 1c.txt
1  #include <stdio.h>
2
3  // this is comment
4
5  int main(){
6      /*
7      multi line comment
8      */
9      printf("hello world\n");
10     return 0;
11 }
12
```

Output:

```
2 Comment(s)
```

```
lab > output.txt
1  #include <stdio.h>
2
3
4
5  int main(){
6
7      printf("hello world\n");
8      return 0;
9  }
10
```

- b. **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;
}
```



```
lab > ≡ 1c.txt
1  #include <stdio.h>
2
3  // this is comment
4
5  int main(){
6      int x = 10;
7      float y = 3.14;
8      char z = 'A';
9
10     if (x == 10) {
11         y = 5.2;
12     }
13     //this is comment
14     while (y>0) {
15         y--;
16     }
17     /*
18     multi line comment
19     */
20     printf("hello world\n");
21     return 0;
22 }
23 |
```

Output:

```
Unrecognized Character: #
Identifier: include
Operator: <
Identifier: stdio
Special Symbol: .
Identifier: h
Operator: >
Operator: /
Operator: /
Identifier: this
Identifier: is
Identifier: comment
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Identifier: x
Operator: =
Number: 10
Special Symbol: ;
Identifier: y
Operator: =
Number: 3.14
Special Symbol: ;
Identifier: z
Operator: =
Character Literal: 'A'
Special Symbol: ;
Special Symbol: (
Identifier: x
Operator: ==
Number: 10
Special Symbol: )
Special Symbol: {
Identifier: y
Operator: =
Number: 5.2
Special Symbol: ;
```

Program 6

- a. 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:

```
Enter set of character to parsed: i+i-i+i$  
  
Success
```

Program 7

a. **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

| Feature | Description |
|-----------------|-------------------------------|
| Grammar Type | Context-Free Grammar |
| Parsing Method | LALR(1) Parser (Bottom-Up) |
| Integration | Works with LEX/Flex |
| Output | C code (y.tab.c) |
| Action Language | C (executed when rules match) |

Use Cases

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

- b. **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:

```
+  
syntax error
```

```
a+b  
valid syntax
```

- c. **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:

```
2+3-8/15*2
5
```

- d. **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:

```
56+43-55*31
56 43 + 55 31 * -
```