

# Practical no:- 1

## Aim :-

Write a LEX program to recognize valid strings using regular expression:  $(a|b)^*abb$

## Test case :-

Valid strings: abababb, abb, bbbabb

Invalid strings: baba, bba, bbb, ababba

## Tools :-

Software: 1. Flex Software (Compiler) ; Specifications :- Windows 10, 64 bit, exe.

Hardware: 2. Computer System ; Specification :- Windows 10, 500 GB HDD/ 250 GB SSD, i3 processor

## Theory :-

Regular Expression :-

The lexical analyzer needs to scan and identify only a finite set of valid string/ token/ lexeme that belong to the language in hand. Regular expression is an important notation for specifying pattern . Each pattern matches a sort of strings. So regular expression serve as a names for a set of strings.

Regular expression can also be described as a sequence of pattern that define a string. Regular expression is used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

## Operations :-

The various operations on languages are

1) Union of two languages L and M is written as

$$L \cup M = \{S \mid S \text{ is in } L \text{ or } S \text{ is in } M\}$$

2) Concatenation of two language L and M is written as

$$LM = \{St \mid S \text{ is in } L \text{ and } t \text{ is in } M\}$$

3) Kleane closure of a language L is written as  $L^*$

$L^*$  = zero or more occurrence of language L

Flex software is compiler which is used to compile lex code. We will apply some regular expression to fulfil give condition.

## Lex Program :-

%{

```

#include <stdio.h>

int yywrap(void); // consistent declaration

%}

%%

^[ab]*abb$  { printf("Valid string: %s\n", yytext); }
.           { printf("Invalid string: %s\n", yytext); }

%%

int yywrap(void) {
    return 1; // Indicate no more input files
}

int main(int argc, char **argv) {
    yylex();
    return 0;
}

```

**Output :-**

```
C:\Users\prati\OneDrive\Desktop\LexProgram>program1.exe
abababb
Valid string: abababb

abb
Valid string: abb

bbbabb
Valid string: bbbabb

baba
Invalid string: b
Invalid string: a
Invalid string: b
Invalid string: a

bba
Invalid string: b
Invalid string: b
Invalid string: a

bbb
Invalid string: b
Invalid string: b
Invalid string: b

ababba
Invalid string: a
Invalid string: b
Invalid string: a
Invalid string: b
Invalid string: b
```

### **Conclusion :-**

By executing this program, the lex program recognize given string by using regular expression. Implemented successfully and all test cases verified.

## Practical no:- 2

### Aim :-

Write a LEX program to recognize tokens

- 1) Identifier
- 2) Integer Constant ( Decimal, Octal, Hexadecimal)
- 3) Real Constants (Floating point and Exponential format)

### Test Case :-

- a. Integer Constant e.g., +15, -5, 0123, 0X1A
- b. Floating point format e.g., +0.5, 236.0, .567, -26.89
- c. Exponential format e.g., 1.6e-19, -0.5E+2, +1.7E4, .26e-7

### Tools :-

1. LEX Compiler ; Specification :- Windows 10, 64 bit
2. Compiler System ; Specification :- Window 10, 500 GB HDD / 250 GB SSD, i3 processor ; Quantity :- 1

### Theory :-

- a) Regular Expression for Decimal Constant

Rules: 1. must have at least one digit

2. may be +ve or -ve

3. No spaces, comma, period (.) are allowed.

Regular Expression :-  $(+|-)?(\text{Digit})^+$

- b) Regular Expression for real Constant (Flouting point notation)

Rules: 1. must have atleast one digit

2. may be +ve or -ve

3. must contain decimal point

4. at least are digit after decimal point

5. No (-) (,) (.)

Regular Expression :-  $(+|-)?(\text{Digit})^* . (\text{Digit})^+$

In Lexical analysis, the source code is scanned and broken down into these individual tokens, which are then passed on the next stage of the compilation or interpretation process. Some of them can be described as follows :-

1) Identifier :-  $^{\wedge} [a-zA-Z\_][a-zA-Z0-9\_]* \$$

Here '^' - matches the beginning of the string

'[a-zA-Z\\_]' - matches any uppercase letter lowercase letter, or underscore

'[a-zA-Z0-9\\_]' - matches zero or more occurrences of any letter, lowercase letter, digit or underscore

'\$' - matches the end of string

2) Integer Constant :- Regular expression for decimal constant can be-

a. must have at least one digit

b. may be have +ve or -ve

c. No space and commas period (.) are allowed

$(+|-)? (\text{Digit}) +$

3) Real Constant - This will include two categories

i) Floating point - must contain decimal paint.

ii) Exponential point - must contain mantissa and Exponent regular expression for real constant is seperated by 'E' or 'e'

$\text{exponent} [e E] [+ -] . \{ \text{digit} \} + \{ \text{digit} \} + \text{"."} \{ \text{digit} \} * \{ \text{exponent} \} ? | \{ \text{digit} \} + \{ \text{exponent} \}$

### Lex Program :-

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

%option noyywrap

DIGIT [0-9]
HEX_DIGIT [0-9a-fA-F]
FLOAT_SUFFIX [fF][iL]

%%

[a-zA-Z_][a-zA-Z0-9_]*           printf("Valid Identifier: %s\n", yytext);
0[+-]?[1-9][0-9]*              printf("Decimal Integer Constant: %s\n", yytext);
0[0-7]+                          printf("Octal Integer Constant: %s\n", yytext);
0[xX]{HEX_DIGIT}+              printf("Hexadecimal Integer Constant: %s\n", yytext);
```

```

[+-]?{DIGIT}+"."{DIGIT}+{FLOAT_SUFFIX}?  printf("Floating Point Constant: %s\n", yytext);
"."{DIGIT}+{FLOAT_SUFFIX}?                printf("Floating Point Constant: %s\n", yytext);
[+-]?{DIGIT}+"."{DIGIT}+[eE][+-]?{DIGIT}+  printf("Exponential Format Constant: %s\n", yytext);
.|[a-zA-Z0-9_]*|{DIGIT}[a-zA-Z0-9_]*      printf("Invalid Identifier: %s\n", yytext);

%%

```

```

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

```

### Output :-

```

C:\Users\prati\OneDrive\Desktop\LexProgram2>flex b.l
C:\Users\prati\OneDrive\Desktop\LexProgram2>gcc lex.yy.c -o b.exe
C:\Users\prati\OneDrive\Desktop\LexProgram2>b.exe
+15
Decimal Integer Constant: +15

-5
Decimal Integer Constant: -5

46
Decimal Integer Constant: 46

0123
Octal Integer Constant: 0123

0X1A
Hexadecimal Integer Constant: 0X1A

+0.5
Floating Point Constant: +0.5

236.0
Floating Point Constant: 236.0

.567
Floating Point Constant: .567

1.6e-19
Exponential Format Constant: 1.6e-19

```

```
-0.5E+2  
Exponential Format Constant: -0.5E+2  
  
abc  
Valid Identifier: abc  
  
_12a  
Valid Identifier: _12a  
  
&12a  
Invalid Identifier: &12a  
  
89ij  
Invalid Identifier: 89ij
```

### Conclusion :-

We successfully able to learn the concept of lexical analysis like how it breaks the source code into different types of tokens and also implemented lex program to recognize tokens.

- 1) Identifier
- 2) Integer Constant
- 3) Real Constant

## Practical no:- 3

### Aim :-

Source code is a text file. Constants used in the program are strings. Write a program to read integer constants as a string and convert it decimal number. Integer constants are represented in decimal, octal, and hexadecimal. It may be positive or negative.

### Test Case :-

- a. Decimal Constant : +15, -5, 123
- b. Octal constant : +0123, 0117, -0777
- c. Hexadecimal Constant : -0X1A, 0X1B, 0x2a, 1c3d

### Tools :-

- Software :- Lex Compiler
- Hardware :- Computer System  
Specifications :- Windows 10, 500 GB HDD / 250 GB SSD, i3 processor

### Theory :-

Integer Constants - An integer constant can be a decimal, octal, or hexadecimal constant. Constants refer to fixed values that the program may not alter during its execution. The constants can be of the basic data types like an integer constant, a floating constant, a character constant or a string literal.

In this practical we used Integer Constants are represented in Decimal, Octal and Hexadecimal, it can be positive or negative.

Octal - Integer Constant represented in decimal octal. A prefix specifies the base or radix: 0 for octal representation and it specifies a base: 8 for octal.

Hexadecimal - Integer Constant represented in Decimal Hexadecimal constant :- A prefix specifies a base or radix, "0x" or "0X" for hexadecimal constants, and base for hexadecimal is 16.

Default Integer Constants consider (base 10) Decimal.

### Lex Program :-

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
%}  
  
%option noyywrap
```



DIGIT [1-9]

HEX\_DIGIT [0-9a-fA-F]

%%

```
0|[[+]?[1-9][0-9]* {  
    int decimal = atoi(yytext);  
    printf("Decimal: %d\n", decimal);  
}
```

```
-?0[0-7]+ {  
    int octal = strtol(yytext, NULL, 8);  
    printf("Decimal: %d\n", octal);  
}
```

```
-?0[xX]{HEX_DIGIT}+ {  
    int hexadecimal = strtol(yytext, NULL, 16);  
    printf("Decimal: %d\n", hexadecimal);  
}
```

%%

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

```
int my_wrap() {  
    return 1; // Indicate no more input  
}
```

**Output :-**

```
C:\Users\prati\OneDrive\Desktop\LexProgram3>flex a.l
C:\Users\prati\OneDrive\Desktop\LexProgram3>gcc lex.yy.c
C:\Users\prati\OneDrive\Desktop\LexProgram3>a.exe
+23
Decimal: 23

-90
Decimal: -90

0123
Decimal: 83

0x2a
Decimal: 42

0X1B
Decimal: 27
```

### C Program :-

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

int main() {
    char num[20];
    double decimal;

    printf("Enter an integer constant (in decimal, octal, hexadecimal, or floating-point format): ");
    scanf("%s", num);

    int is_negative = 0;
    if (num[0] == '-') {
        is_negative = 1;
        memmove(num, num + 1, strlen(num));
    }

    int base = 10;
    if (num[0] == '0') {
```

```

if (num[1] == 'x' || num[1] == 'X') {
    base = 16;

    memmove(num, num + 2, strlen(num));
}
else if (strchr(num, '.') != NULL) {
    base = 10;
}
else {
    base = 8;
}
}

decimal = 0;
int i = 0;
while (num[i] != '\0') {
    double digit;
    if (isdigit(num[i])) {
        digit = num[i] - '0';
    }
    else if (isalpha(num[i])) {
        digit = toupper(num[i]) - 'A' + 10;
    }
    else if (num[i] == '.') {
        i++;
        double factor = 0.1;
        while (num[i] != '\0') {
            digit += (num[i] - '0') * factor;
            factor *= 0.1;
            i++;
        }
        break; // Exit the loop after processing the fractional part
    }
    else {
        printf("Invalid character: %c\n", num[i]);
    }
}

```

```

        return 1;
    }

    if (digit >= base) {
        printf("Invalid digit for base %d: %c\n", base, num[i]);
        return 1;
    }

    decimal = decimal * base + digit;
    i++;
}

if (is_negative) {
    decimal = -decimal;
}

printf("The decimal equivalent of %s is %f\n", num, decimal);

return 0;
}

```

### **Conclusion :-**

The Lex program designed to read integer constants as strings and convert them into decimal numbers has successfully addressed the aim of the task. By recognizing er constants in various format such as Decimal, Octal and Hexadecimal and handling both positive and negative values, the program demonstrates to capability to accurately process and convert numerical inputs from text files.

## Practical no:- 4

### Aim :-

Write a program to eliminate left recursion from the grammar.

### Test Case :-

1.  $E : E+E \mid E-E \mid id$
2.  $S : (L) \mid a$   
 $L : L,S \mid S$

### Tools :-

1. Lex Compiler (language compiler)

Specifications :- Any GDB Compiler (MinGW)

2. Computer System.

Specifications :- Windows 10, 150 GB HDD / 8 GB RAM, i3 processor

Quantity :- 1

### Theory :-

A Grammar  $G (V, T, P, S)$  is left recursive if it has a production in the form

$A \rightarrow A\alpha \mid \beta$

The above is left recursive because the left of production is occurring of a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with.

$A \rightarrow B A'$

$A' \rightarrow \alpha A' \mid \epsilon$

1. Left Recursion can be eliminated by introducing new non-terminal.  $A'$  such that, this process can also define as elimination of direct left recursion.

Left recursion grammar

$A \rightarrow A\alpha \mid \beta$

Removal of left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

2. Eliminate indirect left recursion

Step one describes a rules to eliminate direct left recursion from a production. To eliminate left recursion from an entire grammar may be more difficult because of indirect left recursion.

For e.g.,

$A \rightarrow Bxy \mid x$

$B \rightarrow CD$

$C \rightarrow A|C$

$D \rightarrow d$

For e.g.,  $E \rightarrow E+T \mid T$

Comparing it with  $a \rightarrow A\alpha \mid \beta$

$A = E \quad \alpha = +T \quad \beta = T$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$E \rightarrow T E'$

$A' \rightarrow \alpha A' \mid \epsilon$

$E' \rightarrow + T E' \mid \epsilon$

C compiler will compile the code which used for eliminate left recursion.

### **Lex Program :-**

```
%{  
  
#include<stdio.h>  
  
#include<stdlib.h>  
  
#include<string.h>  
  
  
  
#define MAX_RULES 10  
  
#define MAX_LEN 50  
  
  
  
char productions[MAX_RULES][MAX_LEN];  
char nonTerminals[MAX_RULES];  
int numProductions = 0;  
  
void eliminateLeftRecursion() {  
    printf("\nGrammar Without Left Recursion:\n\n");  
    for (int i = 0; i < numProductions; i++) {  
        char pro[MAX_LEN], alpha[MAX_LEN], beta[MAX_LEN];
```

```

int j, index = 3;
strcpy(pro, productions[i]);
char nonTerminal = pro[0];
if (nonTerminal == pro[index]) {
    int hasLeftRecursion = 0;
    for (j = index + 1; pro[j] != '\0'; j++) {
        if (pro[j] == '|') {
            hasLeftRecursion = 1;
            break;
        }
    }
    if (hasLeftRecursion) {
        index++;
        int k = 0;
        // Extracting alpha and beta
        for (j = index; pro[j] != '|'; j++) {
            alpha[k++] = pro[j];
        }
        alpha[k] = '\0';
        k = 0;
        for (j = j + 1; pro[j] != '\0'; j++) {
            beta[k++] = pro[j];
        }
        beta[k] = '\0';
        printf(" %c -> %s%c\n", nonTerminal, beta, nonTerminal);
        printf(" %c' -> %s%c'\n", nonTerminal, alpha, nonTerminal);
        printf(" %c' -> epsilon\n", nonTerminal);
    } else {
        printf(" %s\n", productions[i]);
    }
} else {
    printf(" %s\n", productions[i]);
}
}

```

```
}
```

```
int main() {  
    printf("Enter the number of production rules: ");  
    scanf("%d", &numProductions);  
  
    printf("Enter the production rules in the form 'E->E|a':\n");  
  
    for (int i = 0; i < numProductions; i++) {  
        scanf("%s", productions[i]);  
        nonTerminals[i] = productions[i][0];  
    }  
    eliminateLeftRecursion();  
    return 0;  
}  
%}
```

```
%%
```

```
[A-Za-z]->[A-Za-z]+ {  
    printf("Production: %s\n", yytext);  
    strcpy(productions[numProductions], yytext);  
    nonTerminals[numProductions] = yytext[0];  
    numProductions++;  
}
```

```
.\n ; // Ignore any other characters or newlines
```

```
%%
```

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

```
int yyerror(char *s) {  
    printf("Error: %s\n", s);  
}
```



```

return 0;
}

```

### Output :-

```

C:\Users\prati\OneDrive\Desktop\LexProgram4>c.exe
Enter the number of production rules: 1
Enter the production rules in the form 'E->E|a':
E->E+E|E-E|id

Grammar Without Left Recursion:

E -> E-E|idE'
E' -> +EE'
E' -> epsilon

C:\Users\prati\OneDrive\Desktop\LexProgram4>c.exe
Enter the number of production rules: 2
Enter the production rules in the form 'E->E|a':
S->(L)|a
L->L,S|S

Grammar Without Left Recursion:

S->(L)|a
L -> SL'
L' -> ,SL'
L' -> epsilon

```

### C Program :-

```

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define SIZE 20

int main() {
    char pro[SIZE], alpha[SIZE], beta[SIZE];
    int nonterminal, i, j, index = 5; // index adjusted for ' -> '
    printf("Enter the production as E -> E|a:");
    scanf("%s", pro);
    nonterminal = pro[0];
    if (nonterminal == pro[index]) {
        int alphaIndex = 0, betaIndex = 0;
        int isAlpha = 1; // Flag to differentiate between alpha and beta
    }
}

```

```

for (i = index + 3; pro[i] != '\0'; i++) {
    if (pro[i] == '+') {
        isAlpha = 0; // Beta part starts after '+'
        continue;
    }
    if (pro[i] == '|') {
        alpha[alphaIndex] = '\0';
        isAlpha = 1; // Reset flag for next term after '|'
        alphaIndex = 0;
        continue;
    }
    if (isAlpha) {
        alpha[alphaIndex++] = pro[i];
    } else {
        beta[betaIndex++] = pro[i];
    }
}
alpha[alphaIndex] = '\0';
beta[betaIndex] = '\0';

printf("\nGrammar without left recursion:\n\n");
printf("%c->%s%c\n", nonterminal, beta, nonterminal);
printf("%c->%s%c\n", nonterminal, alpha, nonterminal);
} else {
    printf("\nThis grammar is not left recursion.\n");
}
return 0;
}

```

### Conclusion :-

By implementing this practical we learnt about the process to eliminate left recursion from the given grammar, and understand the concept of elimination of left recursion from grammar.

## Practical no:- 5

### Aim :-

Design recursive descent parser for the grammar

$E \rightarrow +EE \mid -EE \mid a \mid b$

### Tools :-

1. C Compilers ; Specification :- Turbo, GCC, acc, any ; Quantity :- 1
2. Complex System ; Specification :- Windows, 4 GB, RAM 1 TB HDD ; Quantity :- 1

### Theory :-

A recursive descent parser is a top-down parsing technique where a parser tree is built from top and constructed down to leave this type of parser starts with highest level of grammar & recursively expands non-terminals until a terminal symbol is reached.

To implement this grammar using recursive descent parser, we need to follow steps to implement parser :-

1. Analyze grammar, identify terminals, non-terminals, production rule
2. Write function to parse each non-terminal
3. Use tokenizer to convert input string into list of tokens.
4. Pass list tokens as input, function should return whether string can be derived from grammar or not.

### Lex Program :-

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
%}  
  
%%  
"+" { return '+'; }  
"- " { return '-'; }  
"a" { return 'a'; }  
"b" { return 'b'; }  
"|" { return '|'; }  
"\n" { return '\n'; }  
. { fprintf(stderr, "Invalid character: %c\n", yytext[0]); exit(1); }  
%%
```

```
int lookahead;
```

```
void match(int token) {  
    if (lookahead == token)  
        lookahead = yylex();  
    else {  
        fprintf(stderr, "Parsing unsuccessful\n");  
        exit(1);  
    }  
}
```

```
void E();
```

```
void E1();
```

```
void E() {  
    if (lookahead == 'a' || lookahead == 'b') {  
        printf("E -> %c\n", lookahead);  
        match(lookahead);  
    }  
    else if (lookahead == '+' || lookahead == '-') {  
        printf("E -> %c\n", lookahead);  
        match(lookahead);  
        E();  
        E1();  
    }  
    else {  
        fprintf(stderr, "Syntax error\n");  
        exit(1);  
    }  
}
```

```
void E1() {  
    if (lookahead == 'a' || lookahead == 'b' || lookahead == '+' || lookahead == '-') {
```

```

        E();
        E1();
    }
}

int yywrap() {
    return 1; // Indicate that there are no more files to scan
}

int main() {
    lookahead = yylex();
    E();
    if (lookahead != '\n') {
        fprintf(stderr, "Parsing unsuccessful\n");
        exit(1);
    }
    printf("Parsing successful\n");
    return 0;
}

```

### Output :-

```

C:\Users\prati\OneDrive\Desktop\LexProgram5>c.exe
a
E -> a
Parsing successful

C:\Users\prati\OneDrive\Desktop\LexProgram5>c.exe
b
E -> b
Parsing successful

C:\Users\prati\OneDrive\Desktop\LexProgram5>c.exe
+a+b
E -> +
E -> a
E -> +
E -> b
Parsing successful

C:\Users\prati\OneDrive\Desktop\LexProgram5>c.exe
b+
E -> b
Parsing unsuccessful

C:\Users\prati\OneDrive\Desktop\LexProgram5>c.exe
a++
E -> a
Parsing unsuccessful

```

## C Program :-

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_LEN 100


// Function prototypes

void E();

void match(char expected);


// Global variables

char input[MAX_LEN];

int current = 0;


// Function to parse the non-terminal E

void E() {
    if (input[current] == '+') {
        match('+');

        E();

        E();
    } else if (input[current] == '-') {
        match('-');

        E();

        E();
    } else if (input[current] == 'a') {
        match('a');
    } else if (input[current] == 'b') {
        match('b');
    } else {
        printf("The input string is not valid according to the grammar.\n");
        exit(1);
    }
}
```

```

}

// Function to match the current input character with the expected character
void match(char expected) {
    if (input[current] == expected) {
        current++;
    } else {
        printf("The input string is not valid according to the grammar.\n");
        exit(1);
    }
}

int main() {
    printf("Enter an expression: ");
    scanf("%s", input);

    E(); // Start parsing from the start symbol E

    if (input[current] == '\0') {
        printf("The input string is valid according to the grammar.\n");
    } else {
        printf("The input string is not valid according to the grammar.\n");
    }

    return 0;
}

```

### **Conclusion :-**

The program to implement recursive descent parser for grammar designed successfully.

## Practical no :- 6

### Aim :-

Design a program for basic calculator using YACC or BISON.

### Tools :-

1. Compile ; Specification :- YACC OF BISON
2. Computer System ; Specification :- Windows, 4 GB RAM, 1 TB HDD ; Quantity :- 1

### Theory :-

YACC ( Yet Another Compiler Compiler ) & BISON are tool that generate parsers ( Programs analyze structure of text on code ) based on formal design of language grammar commonly used in development of compilers, interpreters & other language based software.

Syntax –

```
% {  
    //declaration section.  
% }  
% %  
% token definition  
    grammar declaration  
% %  
int main ( ) {  
yy parse ( );  
return 1;  
}
```

### Lex Program :-

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
double yylval;  
%}  
  
%%  
[0-9]+      { yylval = atof(yytext); return '0'; }
```



```

[0-9]+[.][0-9]+ { yyival = atof(yytext); return '0'; }
"+"          { return '+'; }
"-"          { return '-'; }
"*"          { return '*'; }
"/"          { return '/'; }
[ \t\n]      /* Skip whitespace */
.            { printf("Invalid expression: %s\n", yytext); }

```

```

%%

```

```

int main(void) {
    double result;

    char op;

    printf("Enter operation (operand1 operator operand2):\n");

    while (scanf("%lf %c %lf", &result, &op, &yyival) != EOF) {
        double operand = yyival;
        double answer;
        switch (op) {
            case '+':
                answer = result + operand;
                printf("%s: %.2f\n", "Addition", answer);
                break;
            case '-':
                answer = result - operand;
                printf("%s: %.2f\n", "Subtraction", answer);
                break;
            case '*':
                answer = result * operand;
                printf("%s: %.2f\n", "Multiplication", answer);
                break;
            case '/':
                if (operand != 0) {

```

```

        answer = result / operand;

        printf("%s: %.2f\n", "Division", answer);
    }
    else {
        printf("Error: Division by zero\n");
        continue; // Skip to next iteration
    }

    break;

default:
    printf("Invalid operation\n");
    continue; // Skip to next iteration
}

result = answer;

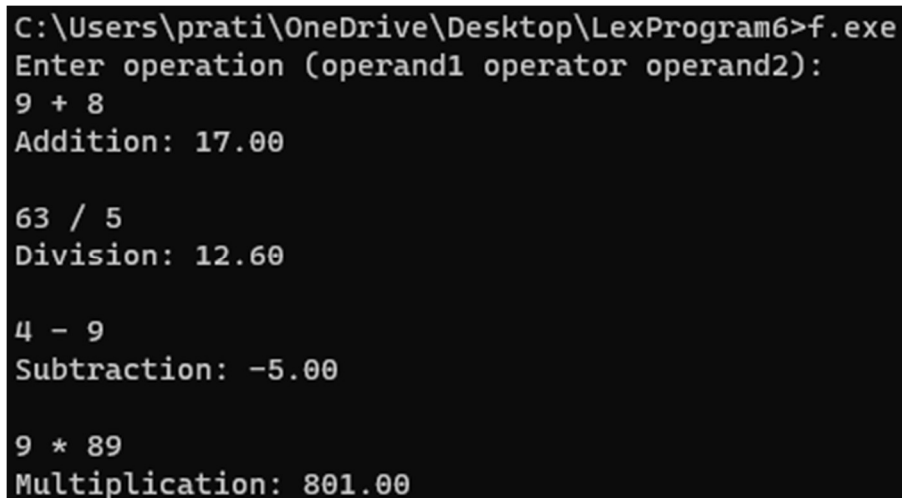
printf("\n"); // Print a newline for spacing between operations
}

return 0;
}

int yywrap() { return 1; }

```

### Output :-



```

C:\Users\prati\OneDrive\Desktop\LexProgram6>f.exe
Enter operation (operand1 operator operand2):
9 + 8
Addition: 17.00

63 / 5
Division: 12.60

4 - 9
Subtraction: -5.00

9 * 89
Multiplication: 801.00

```

### Conclusion :-

The program for basic calculator using YACC or BISON designed successfully.

## Practical no:- 7

### Aim :-

Design a program to evaluate postfix expression.

### Tools :-

1. C Compiler ; Specification :- Turbo C, acc, any
2. Computer System ; Specification :- Windows, 4 GB RAM ; Quantity:- 1

### Theory :-

Postfix Notation: way of writing arithmetic expressions in which each operation follow it's operands

e.g., " 3 + 4 " => " 3 4 + "

Set Steps to evaluate postfix expression i.e., " 3 4 + "

=>Scan expression from left to right :

- push 3 onto stack ( stack : 3 )

- push 4 onto stack ( stack : 4 )

- pop 3 and 4 from stack add them together (  $3 + 4 = 7$  ) and push the result (i.e., 7) back onto stack

Return expression present at the top of stack which is 7.

### Lex Program :-

```
%{  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
  
// Token types  
  
#define NUMBER 256  
  
  
// Declare yylval  
  
double yylval;  
  
  
// Function prototypes  
  
double evalPostfix(const char *expr);  
  
%}
```

```
%option noyywrap yylineno
```

```
%%
```

```
[0-9]+(\.[0-9]+)?    { yylval = atof(yytext); return NUMBER; }
```

```
[-+*/]              { return yytext[0]; }
```

```
[\n\t]              /* ignore whitespace and newline */
```

```
.                  { printf("Invalid character: %c\n", yytext[0]); return -1; }
```

```
%%
```

```
int main() {
```

```
    char expr[100];
```

```
    printf("Enter a postfix expression: ");
```

```
    fgets(expr, sizeof(expr), stdin);
```

```
    double result = evalPostfix(expr);
```

```
    if (result != -1)
```

```
        printf("Result: %g\n", result);
```

```
    return 0;
```

```
}
```

```
double evalPostfix(const char *expr) {
```

```
    double stack[100];
```

```
    int top = -1;
```

```
    for (int i = 0; expr[i] != '\0'; i++) {
```

```
        int token = expr[i];
```

```
        if (token >= '0' && token <= '9') {
```

```
            double number = strtod(&expr[i], NULL);
```

```
            stack[++top] = number;
```

```
            while (expr[i] >= '0' && expr[i] <= '9' || expr[i] == '.') {
```

```
                i++;
```

```

    }

    i--;
} else if (token == '+' || token == '-' || token == '*' || token == '/') {
    if (top < 1) {
        printf("Invalid postfix expression\n");
        return -1;
    }

    double operand2 = stack[top--];
    double operand1 = stack[top--];

    switch (token) {
        case '+':
            stack[++top] = operand1 + operand2;
            break;
        case '-':
            stack[++top] = operand1 - operand2;
            break;
        case '*':
            stack[++top] = operand1 * operand2;
            break;
        case '/':
            if (operand2 == 0) {
                printf("Division by zero\n");
                return -1;
            }
            stack[++top] = operand1 / operand2;
            break;
    }
}

if (top != 0) {
    printf("Invalid postfix expression\n");
}

```

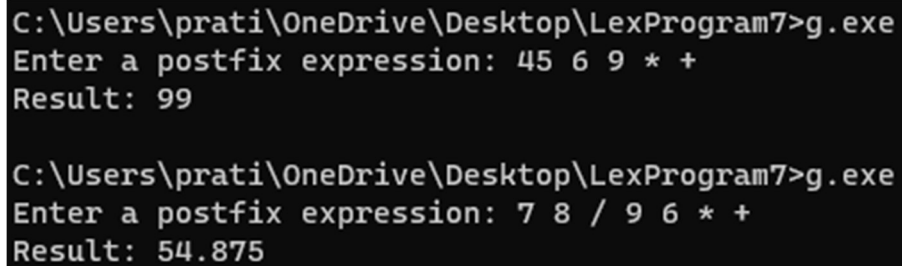
```

        return -1;
    }

    return stack[top];
}

```

### Output :-



```

C:\Users\prati\OneDrive\Desktop\LexProgram7>g.exe
Enter a postfix expression: 45 6 9 * +
Result: 99

C:\Users\prati\OneDrive\Desktop\LexProgram7>g.exe
Enter a postfix expression: 7 8 / 9 6 * +
Result: 54.875

```

### C Program :-

```

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

#define MAX_SIZE 100

// Define a stack structure
typedef struct {
    int items[MAX_SIZE];
    int top;
} Stack;

// Function to initialize the stack
void initialize(Stack *s) {
    s->top = -1;
}

// Function to push an item onto the stack
void push(Stack *s, int value) {
    if (s->top == MAX_SIZE - 1) {

```

```

    printf("Stack Overflow\n");
    exit(EXIT_FAILURE);
}
s->items[++(s->top)] = value;
}

```

// Function to pop an item from the stack

```

int pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack Underflow\n");
        exit(EXIT_FAILURE);
    }
    return s->items[(s->top)--];
}

```

// Function to evaluate the postfix expression

```

int evaluatePostfix(char *expression) {
    Stack stack;
    initialize(&stack);

    for (int i = 0; expression[i] != '\0'; i++) {
        char ch = expression[i];

        if (isdigit(ch)) {
            push(&stack, ch - '0');
        } else {
            int operand2 = pop(&stack);
            int operand1 = pop(&stack);

            switch (ch) {
                case '+':
                    push(&stack, operand1 + operand2);
                    break;
                case '-':

```

```

        push(&stack, operand1 - operand2);
        break;
    case '*':
        push(&stack, operand1 * operand2);
        break;
    case '/':
        push(&stack, operand1 / operand2);
        break;
    default:
        printf("Invalid postfix expression\n");
        exit(EXIT_FAILURE);
    }
}

if (stack.top != 0) {
    printf("Invalid postfix expression\n");
    exit(EXIT_FAILURE);
}

return pop(&stack);
}

int main() {
    char postfix[MAX_SIZE];
    printf("Enter postfix expression: ");
    scanf("%s", postfix);
    int result = evaluatePostfix(postfix);
    printf("Result: %d\n", result);
    return 0;
}

```

### **Conclusion :-**

The program to evaluate postfix expression designed successfully.



## Practical no :- 8

### Aim :-

Write a C Program for a selection sort, generate equivalent 3-address code.

### Tools :-

1. C Compiler ; Specification: - Turbo C, gcc, any
2. Computer System ; Specification: - Windows, 4 GB RAM ; Quantity :- 1

### Theory :-

Selection sort is a simple and easy to understand sorting algorithm that works by repeatedly selecting the smallest element from unsorted portion of the list.

It's repeated for the remaining unsorted portion of the list until the entire list is sorted.

Three address code - It is a type of intermediate code which is easy to generate and can be easily converted to machine code, it makes use of at most three address code and one operator to represent an expression.

### C Program :-

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

```
void selectionSort(int arr[], int n) {
```

```
    int i, j, min_idx;
```

```
    for (i = 0; i < n-1; i++) {
```

```
        min_idx = i;
```

```
        for (j = i+1; j < n; j++) {
```

```
            if (arr[j] < arr[min_idx]) {
```

```
                min_idx = j;
```

```
            }
```

```
        }
```

```
        if (min_idx != i) {
```

```
            int temp = arr[i];
```

```
            arr[i] = arr[min_idx];
```

```
            arr[min_idx] = temp;
```

```
        }
```

```
    }
```

```

}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    selectionSort(arr, n);

    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}

```

### Output :-

```

Original array:
64 25 12 22 11
Sorted array:
11 12 22 25 64

```

### Equivalent 3-address code :-

1. int i, j, min\_idx
2. i=0
3. t1 = n-1
4. label 1: if i >= n goto label 2

```
5. min_idx = i
6. j = i + 1
7. label 3: if j >= n goto label 4
8. t2 = arr[j]
9. t3 = arr[min_idx]
10. if t1 < t2 goto label 5
11. min_idx = j
12. label 5: j = j + 1
13. goto label 3
14. label 4: t4 = arr[min_idx]
15. t5 = arr[i]
16. arr[i] = t4
17. arr[min_idx] = t5
18. i = i + 1
19. goto label 1
20. label 2: exit
```

### **Conclusion :-**

C Program for selection sort and its equivalent 3-address code has been generated successfully.

## Practical no:- 9

### Aim :-

Write a C program to optimize the 3-address code generated.

### Tools :-

1. Compiler ; Specification :- Computer (any), Turbo C, C++ of GEC
2. Computer System ; Specification :- 4 GB RAM, Windows 1 TB HDD ; Quantity :- 1

### Theory :-

3 address code :- It is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most address and one computed at each instruction is sorted in temporary variable generated by compiler.

3 address code optimization :- 3 address code is often used as an intermediate representation of code during optimization phases of the compilation process. The 3-address code allows the compiler to analyze the code perform optimization that can improve the performance of the generated code.

### C Program :-

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

```
#include <stdbool.h>
```

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

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

```
#define MAX_CODE_LINES 100
```

```
#define MAX_CODE_LENGTH 50
```

```
// Structure to represent a three-address code line
```

```
typedef struct {
```

```
    char op; // Operation: '+', '-', '*', '/'
```

```
    int result; // Result variable index
```

```
    int arg1; // Argument 1 variable index or constant value
```

```
    int arg2; // Argument 2 variable index or constant value
```

```
} ThreeAddressCode;
```

```
ThreeAddressCode code[MAX_CODE_LINES]; // Array to store three-address code lines
```

```
int codeCount = 0; // Current count of code lines
```

```
// Function to add a three-address code line to the array
```

```
void addCodeLine(char op, int result, int arg1, int arg2) {  
    code[codeCount].op = op;  
    code[codeCount].result = result;  
    code[codeCount].arg1 = arg1;  
    code[codeCount].arg2 = arg2;  
    codeCount++;  
}
```

```
// Function to perform basic optimizations on three-address code
```

```
void optimizeCode() {  
    for (int i = 0; i < codeCount; i++) {  
        // Check if both arguments are constants  
        if (code[i].arg1 >= 0 && code[i].arg2 >= 0) {  
            // Perform constant folding  
            int value;  
            switch (code[i].op) {  
                case '+':  
                    value = code[i].arg1 + code[i].arg2;  
                    break;  
                case '-':  
                    value = code[i].arg1 - code[i].arg2;  
                    break;  
                case '*':  
                    value = code[i].arg1 * code[i].arg2;  
                    break;  
                case '/':  
                    value = code[i].arg1 / code[i].arg2;  
                    break;  
            }  
            // Replace the current line with a simplified assignment if applicable  
            code[i].op = '=';
```

```

        code[i].arg1 = value;
        code[i].arg2 = -1; // No need for a second argument in an assignment
    }
}

// Function to print the optimized three-address code
void printOptimizedCode() {
    for (int i = 0; i < codeCount; i++) {
        if (code[i].op == '=') {
            printf("t%d = %d\n", code[i].result, code[i].arg1);
        } else {
            printf("t%d = t%d %c t%d\n", code[i].result, code[i].arg1, code[i].op, code[i].arg2);
        }
    }
}

int main() {
    // Example three-address code
    addCodeLine('+', 1, 2, 3);
    addCodeLine('*', 4, 1, 2);
    addCodeLine('-', 5, 4, 3);
    addCodeLine('*', 6, 5, 6);

    printf("Original three-address code:\n");
    printOptimizedCode();

    printf("\nOptimized three-address code:\n");
    optimizeCode();
    printOptimizedCode();

    return 0;
}

```

### Output :-

Original three-address code:

t1 = t2 + t3

t4 = t1 \* t2

t5 = t4 - t3

t6 = t5 \* t6

Optimized three-address code:

t1 = 5

t4 = 2

t5 = 1

t6 = 30

### Conclusion :-

Program in C program to optimize the 3-address code generated is successfully executed.

## Practical no:- 10

### Aim :-

Write a C program for selection sort and generate object code for three address code.

### Tools :-

1. Computer ; Specification :- Turbo C, C++ / GCC
2. Computer System ; Specification :- Windows, 4 GB RAM, 1 TB HDD ; Quantity :- 1

### Theory :-

Three address code is an intermediate code if is used by the optimizing compilers. In 3-address code the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language such three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

### C Program :-

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

```
void selectionSort(int arr[], int n) {  
    int i, j, min_idx;  
  
    for (i = 0; i < n-1; i++) {  
        min_idx = i;  
        for (j = i+1; j < n; j++) {  
            if (arr[j] < arr[min_idx]) {  
                min_idx = j;  
            }  
        }  
        if (min_idx != i) {  
            int temp = arr[i];  
            arr[i] = arr[min_idx];  
            arr[min_idx] = temp;  
        }  
    }  
}
```



```

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    selectionSort(arr, n);

    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}

```

### Output :-

```

Original array:
64 25 12 22 11
Sorted array:
11 12 22 25 64

```

### Object Code :-

```

# Data Section

.data

arr: .word 64, 25, 12, 22, 11

n: .word 5

```

# Text Section

.text

.global main

main:

# Initialize variables

ldr r1, =n @ Load n

ldr r2, =arr @ Load address of arr

mov r3, #0 @ Initialize i

mov r4, #0 @ Initialize j

mov r5, #0 @ Initialize minIndex

L1: cmp r3, r1 @ Compare i with n

bge L2 @ If i >= n, go to L2

mov r5, r3 @ minIndex = i

add r4, r3, #1 @ j = i + 1

L3: cmp r4, r1 @ Compare j with n

bge L4 @ If j >= n, go to L4

ldr r6, [r2, r4, LSL #2] @ Load arr[j]

ldr r7, [r2, r5, LSL #2] @ Load arr[minIndex]

cmp r6, r7 @ Compare arr[j] with arr[minIndex]

bge L6 @ If arr[j] >= arr[minIndex], go to L6

mov r5, r4 @ minIndex = j

L6: add r4, r4, #1 @ j = j + 1

b L3 @ Go to L3

L4: ldr r8, [r2, r5, LSL #2] @ Load arr[minIndex]

ldr r9, [r2, r3, LSL #2] @ Load arr[i]

str r8, [r2, r3, LSL #2] @ arr[i] = arr[minIndex]

str r9, [r2, r5, LSL #2] @ arr[minIndex] = arr[i]

L7: add r3, r3, #1 @ i = i + 1

b L1      @ Go to L1

L2:      @ Halt

### **Conclusion :-**

Program to implement C program for selection sort and generating object code for 3-address code is implemented successfully.