1. Differnet Types of Token

Aim: Write a C program to identify different types of Tokens in a given Program.

Program:

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

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
	if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
		ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
		ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
		ch == '[' || ch == ']' || ch == '{' || ch == '}')
		return (true);
	return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
	if (ch == '+' || ch == '-' || ch == '*' ||
		ch == '/' || ch == '>' || ch == '<' ||
		ch == '=')
		return (true);
	return (false);
}
// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
	if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
		str[0] == '3' || str[0] == '4' || str[0] == '5' ||
		str[0] == '6' || str[0] == '7' || str[0] == '8' ||
		str[0] == '9' || isDelimiter(str[0]) == true)
		return (false);
	return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
	if (!strcmp(str, "if") || !strcmp(str, "else") ||
		!strcmp(str, "while") || !strcmp(str, "do") ||
		!strcmp(str, "break") ||
		!strcmp(str, "continue") || !strcmp(str, "int")
		|| !strcmp(str, "double") || !strcmp(str, "float")
		|| !strcmp(str, "return") || !strcmp(str, "char")
		|| !strcmp(str, "case") || !strcmp(str, "char")
		|| !strcmp(str, "sizeof") || !strcmp(str, "long")
		|| !strcmp(str, "short") || !strcmp(str, "typedef")
		|| !strcmp(str, "switch") || !strcmp(str, "unsigned")
		|| !strcmp(str, "void") || !strcmp(str, "static")
		|| !strcmp(str, "struct") || !strcmp(str, "goto"))
```

```
                return (true);
        return (false);
}
// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
        int i, len = strlen(str);
        if (len == 0)
                return (false);
        for (i = 0; i < len; i++) {
                if (str[i] != '0' && str[i] != '1' && str[i] != '2'
                        && str[i] != '3' && str[i] != '4' && str[i] != '5'
                        && str[i] != '6' && str[i] != '7' && str[i] != '8'
                        && str[i] != '9' || (str[i] == '-' && i > 0))
                        return (false);
        }
        return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
        int i, len = strlen(str);
        bool hasDecimal = false;

        if (len == 0)
                return (false);
        for (i = 0; i < len; i++) {
                if (str[i] != '0' && str[i] != '1' && str[i] != '2'
                        && str[i] != '3' && str[i] != '4' && str[i] != '5'
                        && str[i] != '6' && str[i] != '7' && str[i] != '8'
                        && str[i] != '9' && str[i] != '.' ||
                        (str[i] == '-' && i > 0))
                        return (false);
                if (str[i] == '.')
                        hasDecimal = true;
        }
        return (hasDecimal);
}
// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
        int i;
        char* subStr = (char*)malloc(sizeof(char) * (right - left + 2));

        for (i = left; i <= right; i++)
                subStr[i - left] = str[i];
        subStr[right - left + 1] = '\0';
        return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
        int left = 0, right = 0;
        int len = strlen(str);
```

```
        while (right <= len && left <= right) {
            if (isDelimiter(str[right]) == false)
                right++;

            if (isDelimiter(str[right]) == true && left == right) {
                if (isOperator(str[right]) == true)
                    printf("'%c' IS AN OPERATOR\n", str[right]);

                right++;
                left = right;
            } else if (isDelimiter(str[right]) == true && left != right
                    || (right == len && left != right)) {
                char* subStr = subString(str, left, right - 1);

                if (isKeyword(subStr) == true)
                    printf("'%s' IS A KEYWORD\n", subStr);

                else if (isInteger(subStr) == true)
                    printf("'%s' IS AN INTEGER\n", subStr);

                else if (isRealNumber(subStr) == true)
                    printf("'%s' IS A REAL NUMBER\n", subStr);

                else if (validIdentifier(subStr) == true
                        && isDelimiter(str[right - 1]) == false)
                    printf("'%s' IS A VALID IDENTIFIER\n", subStr);

                else if (validIdentifier(subStr) == false
                        && isDelimiter(str[right - 1]) == false)
                    printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
                left = right;
            }
        }
        return;
    }


    // DRIVER FUNCTION
    int main()
    {
        // maximum length of string is 100 here
        char str[100] = "int a = b + 1c; ";

        parse(str); // calling the parse function

        return (0);
    }
```

Output:

2. Implement a Lexical Analyzer

Aim: Write a Lex Program to implement a Lexical Analyzer using Lex tool.

Program:

```
%{
int COMMENT=0;
%}
identifier  [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;} {printf("\n\t %s is a COMMENT",yytext);}
{identifier}\(  {if(!COMMENT)printf("\nFUNCTION  \n\t%s",yytext);}
\{  {if(!COMMENT)printf("\n BLOCK BEGINS");}
\}  {if(!COMMENT)printf("BLOCK ENDS ");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
```

```
file=fopen("var.c","r");
if(!file)
{ printf("could not open the file");
exit(0); }
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{   return(1); }
```

Program2 : var.c

```
#include<stdio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

Execution:

Step1: Run Program2

```
$vi var.c
$gcc var.c
$./a.out
Sum:3
```

Step2: Run Program 1

```
        $vi lex1.l
        $lex lex1.l
        $gcc lex.yy.c
        $./a.out
```

#include<stdio.h> is a preprocessor directive
void is a keyword
FUNCTION
main(
)
BLOCK BEGINS
int is a keyword
a IDENTIFIER,
b IDENTIFIER,
c IDENTIFIER;
a IDENTIFIER
= is an ASSIGNMENT OPERATOR
1 is a NUMBER ;

b IDENTIFIER

= is an ASSIGNMENT OPERATOR
2 is a NUMBER ;

c IDENTIFIER
= is an ASSIGNMENT OPERATOR
a IDENTIFIER+
b IDENTIFIER;
FUNCTION
printf(
"Sum:%d" is a STRING,
c IDENTIFIER
)
;
BLOCK ENDS

Output:

3. Simulate Lexical Analyzer

Aim: Write a C program to Simulate Lexical Analyzer to validating a given input String.

Program:

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

int main()
{
    char arithmetic[5] = {'+', '-', '*', '/', '%'};
    char relational[4] = {'<', '>', '!', '='};
    char bitwise[5] = {'&', '^', '~', '|'};
    char str[2] = {' ', ' '};

    printf("Enter value to be identified: ");
    scanf("%s", &str);

    int i;

    if (((str[0] == '&' || str[0] == '|') && str[0] == str[1]) || (str[0] ==
'!' && str[1] == '\0'))
    {
        printf("\nIt is Logical operator");
    }

    for (i = 0; i < 4; i++)
    {
        if (str[0] == relational[i] && (str[1] == '=' || str[1] == '\0'))
        {
            printf("\nIt is Relational Operator");
            break;
        }
    }

    for (i = 0; i < 4; i++)
    {
        if ((str[0] == bitwise[i] && str[1] == '\0') ||
            ((str[0] == '<' || str[0] == '>') && str[1] == str[0]))
        {
            printf("\nIt is Bitwise Operator");
            break;
        }
    }

    if (str[0] == '?' && str[1] == ':')
    {
        printf("\nIt is Ternary Operator");
    }

    for (i = 0; i < 5; i++)
    {
        if ((str[0] == '+' || str[0] == '-') && str[0] == str[1])
```

```
            {
                printf("\nIt is Unary Operator");
                break;
            }
            else if ((str[0] == arithmetic[i] && str[1] == '=') || (str[0] == '='
    && str[1] == ' '))
            {
                printf("\nIt is Assignment Operator");
                break;
            }
            else if (str[0] == arithmetic[i] && str[1] == '\0')
            {
                printf("\nIt is Arithmetic Operator");
                break;
            }
        }

        return 0;
    }
```

Output:

4. Brute Force Technique of Top-Down Parsing

Aim: Write a C program to implement the Brute force technique of Top-down Parsing.
Consider the CFG: S -> cAd, A -> ab | a .

Program:

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

bool parse(char *);

int main() {
    char input[100];

    printf("Enter a string to parse: ");
    scanf("%s", input);

    if (parse(input)) {
        printf("String is successfully parsed.\n");
    } else {
        printf("Error in parsing string.\n");
    }

    return 0;
}

bool parse(char *input) {
    int index = 0;

    if (input[index] == 'c') {
        index++;
        if (input[index] == 'a') {
            index++;
            if (input[index] == 'b' || input[index] == 'd') {
                index++;
                if (input[index] == 'd' || input[index] == '\0') {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Output:

5. Recursive Descent Parser

Aim: To implement a Recursive Descent Parser in C that checks whether a given arithmetic expression follows a predefined Context-Free Grammar (CFG) and provides step-by-step derivations of its parsing process.

Program:

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

#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();

const char *cursor;
char string[64];

int main() {
    puts("CFG for implementing Recursive Descent Parser:\n"
         "E -> T E'\n"
         "E' -> + T E' | e\n"
         "T -> F T'\n"
         "T' -> * F T' | e\n"
         "F -> ( E ) | i\n");

    printf("Enter the string: ");
    scanf("%s", string);
    cursor = string;

    puts(string);
    puts("Input      Action");
    puts("--------------------------------------------------------------------------------");

    if (E() && *cursor == '\0') {
        puts("--------------------------------------------------------------------------------");
        puts("String is successfully parsed");
        return 0;
    } else {
        puts("--------------------------------------------------------------------------------");
        puts("Error in parsing String");
        return 1;
    }
}

int E() {
    printf("%-16s E -> T E'\n", cursor);
    if (T()) {
        if (Edash())
            return SUCCESS;
        else
            return FAILED;
    }

    return FAILED;
}
```

```
int Edash() {
    if (*cursor == '+') {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;
        if (T()) {
            if (Edash())
                return SUCCESS;
            else
                return FAILED;
        }
        return FAILED;
    }
    printf("%-16s E' -> $\n", cursor);
    return SUCCESS;
}

int T() {
    printf("%-16s T -> F T'\n", cursor);
    if (F()) {
        if (Tdash())
            return SUCCESS;
        else
            return FAILED;
    }
    return FAILED;
}

int Tdash() {
    if (*cursor == '*') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) {
            if (Tdash())
                return SUCCESS;
            else
                return FAILED;
        }
        return FAILED;
    }
    printf("%-16s T' -> $\n", cursor);
    return SUCCESS;
}

int F() {
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) {
            if (*cursor == ')') {
                cursor++;
                return SUCCESS;
            }
            return FAILED;
        }
        return FAILED;
    } else if (*cursor == 'i') {
        cursor++;
        printf("%-16s F -> i\n", cursor);
```

```
            return SUCCESS;
        }
        return FAILED;
    }
```

Output:

6. First and Follow set generator

Aim: To implement a First and Follow set generator for a given Context-Free Grammar (CFG) using C, which helps in constructing parsing tables for compiler design.

Program:

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

// Functions to calculate Follow
void followfirst(char, int, int);

void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;

// Stores the final result of the First Sets
char calc_first[10][100];

// Stores the final result of the Follow Sets
char calc_follow[10][100];

int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main() {
    int jm = 0, km = 0, i, choice;
    char c, ch;
    count = 8;

    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");

    int kay;
    char done[count];
    int ptr = -1;

    // Initializing the calc_first array
    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
```

```
        }
    }

    int point1 = 0, point2, x;

    for (k = 0; k < count; k++) {
        c = production[k][0];
        point2 = 0;
        x = 0;

        // Checking if First of c has already been calculated
        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                x = 1;

        if (x == 1)
            continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        for (i = 0 + jm; i < n; i++) {
            int lark = 0, chk = 0;

            for (lark = 0; lark < point2; lark++) {
                if (first[i] == calc_first[point1][lark]) {
                    chk = 1;
                    break;
                }
            }

            if (chk == 0) {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }

        printf("}\n");
        jm = n;
        point1++;
    }

    printf("\n");
    printf("----------\n\n");

    char donee[count];
    ptr = -1;

    // Initializing the calc_follow array
    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
```

```
                calc_follow[k][kay] = '!';
            }
        }

        point1 = 0;
        int land = 0;

        for (e = 0; e < count; e++) {
            ck = production[e][0];
            point2 = 0;
            x = 0;

            // Checking if Follow of ck has already been calculated
            for (kay = 0; kay <= ptr; kay++)
                if (ck == donee[kay])
                    x = 1;

            if (x == 1)
                continue;

            land += 1;

            // Function call
            follow(ck);
            ptr += 1;

            // Adding ck to the calculated list
            donee[ptr] = ck;
            printf(" Follow(%c) = { ", ck);
            calc_follow[point1][point2++] = ck;

            // Printing the Follow Sets of the grammar
            for (i = 0 + km; i < m; i++) {
                int lark = 0, chk = 0;
                for (lark = 0; lark < point2; lark++) {
                    if (f[i] == calc_follow[point1][lark]) {
                        chk = 1;
                        break;
                    }
                }
                if (chk == 0) {
                    printf("%c, ", f[i]);
                    calc_follow[point1][point2++] = f[i];
                }
            }

            printf(" }\n\n");
            km = m;
            point1++;
        }
    }

    void follow(char c) {
        int i, j;

        // Adding "$" to the follow set of the start symbol
        if (production[0][0] == c) {
            f[m++] = '$';
```

```
    }

    for (i = 0; i < 10; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    // Calculate the first of the next Non-Terminal in the
production
                    followfirst(production[i][j + 1], i, (j + 2));
                }

                if (production[i][j + 1] == '\0' && c != production[i][0]) {
                    // Calculate the follow of the Non-Terminal in the L.H.S.
of the production
                    follow(production[i][0]);
                }
            }
        }
    }
}


void findfirst(char c, int q1, int q2) {
    int j;

    // The case where we encounter a Terminal
    if (!(isupper(c))) {
        first[n++] = c;
    }

    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0)) {
                    // Recursion to calculate First of New Non-Terminal we
encounter after epsilon
                    findfirst(production[q1][q2], q1, (q2 + 1));
                } else {
                    first[n++] = '#';
                }
            } else if (!isupper(production[j][2])) {
                first[n++] = production[j][2];
            } else {
                // Recursion to calculate First of New Non-Terminal we
encounter at the beginning
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2) {
    int k;

    // The case where we encounter a Terminal
    if (!(isupper(c)))
        f[m++] = c;
```

```
        else {
            int i = 0, j = 1;
            while (calc_first[i][j] != '!') {
                if (calc_first[i][j] != '#')
                    f[m++] = calc_first[i][j];
                else {
                    if (production[c1][c2] == '\0')
                        follow(production[c1][0]);
                    else
                        followfirst(production[c1][c2], c1, c2 + 1);
                }
                j++;
            }
        }
    }
```

Output:

7a. Left recursion from the given expression grammar

Aim: To eliminate left recursion from the given expression grammar using C++. The program identifies left-recursive productions and converts them into right-recursive forms while preserving the language.

Program:

```
#include <bits/stdc++.h>
using namespace std;

void removeLeftRecursion(string str) {
    int ind = 0;
    string s1 = "", s2 = "";

    // Find the index of '/'
    for (int i = 0; i < str.size(); i++) {
        if (str[i] == '/') {
            ind = i;
            break;
        }
    }

    // Check for invalid production
    if (!isupper(str[0])) {
        cout << "Invalid Production";
    }
    // Check for left recursion
    else if (str[0] == str[3] && ind != 0) {
        s1 = string(1, str[0]) + " -> " + string(1, str[ind + 1]) + string(1,
str[0]) + "'";
        s2 = string(1, str[0]) + "'" + " -> " + str.substr(4, str.size() - ind)
+ string(1, str[0]) + "'" + string(1, str[ind]) + "ε";

        cout << s1 << "\n";
        cout << s2 << "\n";
    }
    else {
        cout << str << "\n";
    }
}

int main() {
    int n;
    cout << "Enter your number of productions: ";
    cin >> n;

    string str[n];
    cout << "Enter a grammar: \n";

    for (int i = 0; i < n; i++) {
        cin >> str[i];
    }

    cout << "\nGrammar after removing left recursion:\n";
    for (int i = 0; i < n; i++) {
        removeLeftRecursion(str[i]);
```

```
        }

        return 0;
    }
```

Output:

7b. Left Factoring

Aim: To implement a C program that eliminates left factoring from a given grammar. The program identifies common prefixes in production rules and refactors them to ensure proper parsing.

Program:

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

int main() {
    char gram[20], part1[20], part2[20], modifiedGram[20], newGram[20];
    int i, j = 0, k = 0, l = 0, pos;

    // Taking input
    printf("Enter Production : A->");
    gets(gram);

    // Splitting the production into two parts
    for (i = 0; gram[i] != '|'; i++, j++) {
        part1[j] = gram[i];
    }
    part1[j] = '\0';

    for (j = ++i, i = 0; gram[j] != '\0'; j++, i++) {
        part2[i] = gram[j];
    }
    part2[i] = '\0';

    // Finding common prefix
    for (i = 0; i < strlen(part1) || i < strlen(part2); i++) {
        if (part1[i] == part2[i]) {
            modifiedGram[k] = part1[i];
            k++;
            pos = i + 1;
        }
    }

    // Creating new production for left factoring
    for (i = pos, j = 0; part1[i] != '\0'; i++, j++) {
        newGram[j] = part1[i];
    }
    newGram[j++] = '|';

    for (i = pos; part2[i] != '\0'; i++, j++) {
        newGram[j] = part2[i];
    }

    modifiedGram[k] = 'X';  // Adding new non-terminal
    modifiedGram[++k] = '\0';
    newGram[j] = '\0';

    // Displaying result
    printf("\nGrammar Without Left Factoring:\n");
    printf(" A -> %s\n", modifiedGram);
    printf(" X -> %s\n", newGram);
```

```
        return 0;
    }
```

Output:

8. Predictive Parser

Aim: Write a C program to implement predictive parser

Program:

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

#define MAX 20

int stack[MAX], top = -1;

void push(int item) {
    if (top >= MAX - 1) {
        printf("STACK OVERFLOW\n");
        exit(1);
    }
    stack[++top] = item;
}

int pop() {
    if (top == -1) {
        printf("UNDERFLOW\n");
        exit(1);
    }
    return stack[top--];
}

char convert(int item) {
    switch (item) {
        case 0: return 'E';
        case 1: return 'e';
        case 2: return 'T';
        case 3: return 't';
        case 4: return 'F';
        case 5: return 'i';
        case 6: return '+';
        case 7: return '*';
        case 8: return '(';
        case 9: return ')';
        case 10: return '$';
        default: return '?';
    }
}

int main() {
    int m[10][10] = {0}; // Initialize entire parsing table to 0
    int i, j, k;
    char ips[MAX];
    int ip[MAX], a, b, t;

    // Fill parsing table manually (Ensure completeness)
    m[0][0] = m[0][3] = 21;
    m[1][1] = 621;
    m[1][4] = m[1][5] = -2;
    m[2][0] = m[2][3] = 43;
```

```
    m[3][1] = m[3][4] = m[3][5] = -2;
    m[3][2] = 743;
    m[4][0] = 5;
    m[4][3] = 809;

    printf("\nGiven CFG:\n E -> E+T / T \n T -> T*F / F \n F -> (E) / i\n");
    printf("Enter the input string ending with $ (e.g., i+i$, i+(i+i)$, etc.):
");
    scanf("%s", ips);

    // Convert input string characters to corresponding integer values
    for (i = 0; ips[i]; i++) {
        switch (ips[i]) {
            case 'E': k = 0; break;
            case 'e': k = 1; break;
            case 'T': k = 2; break;
            case 't': k = 3; break;
            case 'F': k = 4; break;
            case 'i': k = 5; break;
            case '+': k = 6; break;
            case '*': k = 7; break;
            case '(': k = 8; break;
            case ')': k = 9; break;
            case '$': k = 10; break;
            default:
                printf("Invalid character in input!\n");
                return 1;
        }
        ip[i] = k;
    }
    ip[i] = -1; // Mark end of input

    // Initialize stack with start symbol and end symbol
    push(10); // $
    push(0);  // E

    printf("\nStack\t\tInput\n");

    while (1) {
        // Print stack contents
        printf("\t");
        for (j = 0; j <= top; j++)
            printf("%c", convert(stack[j]));

        // Print input string
        printf("\t\t");
        for (k = i; ip[k] != -1; k++)
            printf("%c", convert(ip[k]));

        printf("\n");

        // Success condition: Both stack and input match $
        if (stack[top] == ip[i]) {
            if (ip[i] == 10) {
                printf("\tSUCCESS\n");
                return 0;
            } else {
                top--;
```

```
                i++;
            }
        } else if (stack[top] <= 4 && stack[top] >= 0) { // Non-terminal case
            a = stack[top];
            b = ip[i] - 5;

            // Ensure `b` is within valid range
            if (b < 0 || b >= 10) {
                printf("\tERROR: Invalid input symbol\n");
                return 1;
            }

            t = m[a][b];
            top--;

            if (t > 0) {
                int temp[10], count = 0;

                // Store values in temp array in correct order
                while (t > 0) {
                    temp[count++] = t % 10;
                    t /= 10;
                }

                // Push in reverse order
                for (int x = count - 1; x >= 0; x--) {
                    push(temp[x]);
                }
            } else {
                printf("\tERROR\n");
                return 1;
            }
        } else { // Error case
            printf("\tERROR\n");
            return 1;
        }
    }
}
```

Output:

9. LR Parsing Algorithm

Aim: Write a C Program for implementation of LR Parsing algorithm to accept a given input string.

Program:

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

//Global Variables
int z = 0, i = 0, j = 0, c = 0;

// Modify array size to increase
// length of string to be parsed
char a[16], ac[20], stk[15], act[10];

// This Function will check whether
// the stack contain a production rule
// which is to be Reduce.
// Rules can be E->2E2 , E->3E3 , E->4
void check()
{
    // Copying string to be printed as action
    strcpy(ac,"REDUCE TO E -> ");
     // c=length of input string
    for(z = 0; z < c; z++)
    {
        //checking for producing rule E->4
        if(stk[z] == '4')
        {
            printf("%s4", ac);
            stk[z] = 'E';
            stk[z + 1] = '\0';

            //printing action
            printf("\n$%s\t\t%s$\t", stk, a);
        }
    }
    for(z = 0; z < c - 2; z++)
    {
        //checking for another production
        if(stk[z] == '2' && stk[z + 1] == 'E' &&
                                stk[z + 2] == '2')
        {
            printf("%s2E2", ac);
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t\t%s$\t", stk, a);
            i = i - 2;
        }
    }
    for(z=0; z<c-2; z++)
    {
        //checking for E->3E3
```

```
            if(stk[z] == '3' && stk[z + 1] == 'E' &&
                                    stk[z + 2] == '3')
            {
                printf("%s3E3", ac);
                stk[z]='E';
                stk[z + 1]='\0';
                stk[z + 2]='\0';
                printf("\n$%s\t\t%s$\t", stk, a);
                i = i - 2;
            }
        }
        return ; //return to main
    }
    //Driver Function
    int main()
    {
        char str[50];
        printf("Consider the CFG  -\nE->2E2 \nE->3E3 \nE->4\n");
        printf("\nEnter input string from given CFG: ");
         scanf("%s",str);
        // a is input string
        strcpy(a,str);
        // strlen(a) will return the length of a to c
        c=strlen(a);
        // "SHIFT" is copied to act to be printed
        strcpy(act,"SHIFT");
        // This will print Labels (column name)
        printf("\nStack \t\tInput \t\t Action");

        // This will print the initial
        // values of stack and input
        printf("\n$ \t\t%s$ \t", a);

        // This will Run upto length of input string
        for(i = 0; j < c; i++, j++)
        {
            // Printing action
            printf("\t%s", act);

            // Pushing into stack
            stk[i] = a[j];
                stk[i + 1] = '\0';

            // Moving the pointer
            a[j]=' ';

            // Printing action
            printf("\n$%s\t\t%s$\t", stk, a);

            // Call check function ..which will
            // check the stack whether its contain
            // any production or not
            check();
        }

        // Rechecking last time if contain
        // any valid production then it will
        // replace otherwise invalid
```

```
        check();

        // if top of the stack is E(starting symbol)
        // then it will accept the input
        if(stk[0] == 'E' && stk[1] == '\0')
            printf("Accept\n");
        else //else reject
            printf("Reject\n");
    }
```

Output:

10. Shift Reduce Parser using Stack Data Structure

Aim: Write a C program for the implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.

Program:

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

char ip_sym[15], stack[15];
int ip_ptr = 0, st_ptr = 0, len, i;
char temp[2], temp2[2];
char act[15];

void check();

int main() {
    printf("\n\n\t Shift Reduce Parser\n");
    printf("\n\t***** ****** ******");
    printf("\nGrammar\n");
    printf("E -> E + E\nE -> E / E\n");
    printf("E -> E * E\nE -> a / b\n");

    printf("\nEnter the Input Symbol: ");
    scanf("%s", ip_sym);

    printf("\n\n\t Stack Implementation Table");
    printf("\n Stack\t\t Input Symbol\t\t Action");
    printf("\n $\t\t\t%s$\t\t\t --", ip_sym);

    strcpy(act, "shift");
    temp[0] = ip_sym[ip_ptr];
    temp[1] = '\0';
    strcat(act, temp);

    len = strlen(ip_sym);

    for (i = 0; i <= len - 1; i++) {
        stack[st_ptr] = ip_sym[ip_ptr];
        stack[st_ptr + 1] = '\0';
        ip_sym[ip_ptr] = ' ';
        ip_ptr++;

        printf("\n$%s\t\t%s$\t\t\t%s", stack, ip_sym, act);

        strcpy(act, "shift");
        temp[0] = ip_sym[ip_ptr];
        temp[1] = '\0';
        strcat(act, temp);

        check();
        st_ptr++;
    }
    st_ptr++;
```

```
    check();

    return 0;
}

void check() {
    int flag = 0;
    temp2[0] = stack[st_ptr];
    temp2[1] = '\0';

    // Reduce E → a / b
    if ((!strcmp(temp2, "a")) || (!strcmp(temp2, "b"))) {
        stack[st_ptr] = 'E';

        if (!strcmp(temp2, "a"))
            printf("\n$%s\t\t%s$\t\t\tE -> a", stack, ip_sym);
        else
            printf("\n$%s\t\t%s$\t\t\tE -> b", stack, ip_sym);

        flag = 1;
    }

    // Continue shifting for operators
    if ((!strcmp(temp2, "+")) || (!strcmp(temp2, "*")) || (!strcmp(temp2,
"/"))) {
        flag = 1;
    }

    // Reduce E → E + E / E * E / E / E
    if ((!strcmp(stack, "E+E")) || (!strcmp(stack, "E/E")) || (!strcmp(stack,
"E*E"))) {
        strcpy(stack, "E");
        st_ptr = 0;

        if (!strcmp(stack, "E+E"))
            printf("\n$%s\t\t%s$\t\t\tE -> E + E", stack, ip_sym);
        else if (!strcmp(stack, "E/E"))
            printf("\n$%s\t\t%s$\t\t\tE -> E / E", stack, ip_sym);
        else
            printf("\n$%s\t\t%s$\t\t\tE -> E * E", stack, ip_sym);

        flag = 1;
    }

    // Accept condition
    if (!strcmp(stack, "E") && ip_ptr == len) {
        printf("\n$%s\t\t%s$\t\t\tAccept", stack, ip_sym);
        exit(0);
    }

    // Reject condition
    if (flag == 0) {
        printf("\n%s\t\t%s\t\tReject", stack, ip_sym);
    }

    return;
}
```

Output:

11. Calculator using LEX and YACC tool

Aim: Simulate the calculator using LEX and YACC tool.

Algorithm:

Step1: A Yacc source program has three parts as follows:

Declarations %% translation rules %% supporting C routines

Step2: Declarations Section: This section contains entries that:

i. Include standard I/O header file.

ii. Define global variables.

iii. Define the list rule as the place to start processing.

iv. Define the tokens used by the parser. v. Define the operators and their precedence.

Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

Step5: Main- The required main program that calls the yyparse subroutine to start the program.

Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.

Step7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.

Step8: calc.lex contains the rules to generate these tokens from the input stream.

Program:

//Implementation of calculator using LEX and YACC

LEX PART:

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;
      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%

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

YACC PART:

```
%{
   #include<stdio.h>
   int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%

ArithmeticExpression: E{

        printf("\nResult=%d\n",$$);

        return 0;

       };
E:E'+'E {$$=$1+$3;}
```

```
|E'-'E {$$=$1-$3;}

|E'*'E {$$=$1*$3;}

|E'/'E {$$=$1/$3;}

|E'%'E {$$=$1%$3;}

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

| NUMBER {$$=$1;}
;
%%

void main()
{
  printf("\nEnter Any Arithmetic Expression which can have operations Addition,
Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");
  yyparse();
 if(flag==0)
  printf("\nEntered arithmetic expression is Valid\n\n");
}

void yyerror()
{
  printf("\nEntered arithmetic expression is Invalid\n\n");
  flag=1;
}
```

Output: