EX.NO.1 Implementation of Lexical Analyzer

AIM:

To write a C program to implement Lexical Analyzer.

ALGORITHM:

1. Start the program
2. Include necessary header files.
3. The ctype header file is to load the file with predicate is digit.
4. The define directive defines the buffer size, numerics, assignment operator, relational operator.
5. Initialize the necessary variables.
6. To return index of new string S, token t using insert() function.
7. Initialize the length of every string.
8. Check the necessary condition.
9. Call the initialize() function. This function loads the keywords into the symbol table.
10. Check the conditions such as white spaces, digits, letters and alphanumerics.
11. To return the index of entry for string S, or 0 if S is not found using the lookup( ) function.
12. Check this until EOF is found.
13. Otherwise initialize the token value to be none.
14. In the main function if lookahead equals numeric then the global variable tokenval gives the value of attribute num.
15. Check the necessary conditions such as arithmetic operators, parenthesis, identifiers, assignment operators, and relational operators.
16. Stop the program

SOURCE CODE:

```
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LENGTH 100
bool isDelimiter(char chr)
{
   return (chr == ' ' || chr == '+' || chr == '-' ||
         chr == '*' || chr == '/' || chr == ',' ||
         chr == ';' || chr == '%' || chr == '>' ||
```

```
        chr == '<' || chr == '=' || chr == '(' ||
        chr == ')' || chr == '[' || chr == ']' ||
        chr == '{' || chr == '}');
}
bool isOperator(char chr)
{
   return (chr == '+' || chr == '-' || chr == '*' ||
        chr == '/' || chr == '>' || chr == '<' ||
        chr == '=');
}
bool isValidIdentifier(char* str)
{
   return (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]));
}

bool isKeyword(char* str)
{
 const char* keywords[] = { "auto", "break", "case", "char", "const", "continue", "default", "do",
"double", "else", "enum", "extern", "float", "for", "goto", "if",  "int", "long", "register", "return",
"short", "signed", "sizeof", "static",  "struct", "switch", "typedef", "union",  "unsigned", "void",
"volatile", "while" };
   for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
      if (strcmp(str, keywords[i]) == 0) {
         return true;
      }
   }
   return false;
}
bool isInteger(char* str)
{
   if (str == NULL || *str == '\0') {
return false;
   }
   int i = 0;
   while (isdigit(str[i])) {
      i++;
```

```
    }
    return str[i] == '\0';
}

char* getSubstring(char* str, int start, int end)
{
    int length = strlen(str);
    int subLength = end - start + 1;
    char* subStr = (char*)malloc((subLength + 1) * sizeof(char));
    strncpy(subStr, str + start, subLength);
    subStr[subLength] = '\0';
    return subStr;
}

int lexicalAnalyzer(char* input)
{
    int left = 0, right = 0;
    int len = strlen(input);
    while (right <= len && left <= right) {
        if (!isDelimiter(input[right]))
            right++;
        if (isDelimiter(input[right]) && left == right) {
            if (isOperator(input[right]))
                printf("Token: Operator, Value: %c\n", input[right]);
            right++;
            left = right;
        }
        else if (isDelimiter(input[right]) && left != right
                 || (right == len && left != right)) {
            char* subStr = getSubstring(input, left, right - 1);
            if (isKeyword(subStr))
                printf("Token: Keyword, Value: %s\n", subStr);
            else if (isInteger(subStr))
                printf("Token: Integer, Value: %s\n", subStr);
            else if (isValidIdentifier(subStr) && !isDelimiter(input[right - 1]))
                printf("Token: Identifier, Value: %s\n", subStr);
            else if (!isValidIdentifier(subStr) && !isDelimiter(input[right - 1]))
                printf("Token: Unidentified, Value: %s\n", subStr);
            left = right;
        }
```

```
    }
    return 0;
}

int main()
{
    char lex_input[MAX_LENGTH] = "int a = b + c";
    printf("For Expression \"%s\":\n", lex_input);
    lexicalAnalyzer(lex_input);
    printf(" \n");
    char lex_input01[MAX_LENGTH] = "int x=ab+bc+30+switch+ 0y ";
    printf("For Expression \"%s\":\n", lex_input01);
    lexicalAnalyzer(lex_input01);
    return (0);
}
```

Output:

```
For Expression 1:
Token: Keyword, Value: int
Token: Identifier, Value: a
Token: Operator, Value: =
Token: Identifier, Value: b
Token: Operator, Value: +
Token: Identifier, Value: c

For Expression 2:
Token: Keyword, Value: int
Token: Identifier, Value: x
Token: Operator, Value: =
Token: Identifier, Value: ab
Token: Operator, Value: +
Token: Identifier, Value: bc
Token: Operator, Value: +
Token: Integer, Value: 30
Token: Operator, Value: +
Token: Keyword, Value: switch
Token: Operator, Value: +
Token: Unidentified, Value: 0y
```

RESULT:

Thus a C program is implemented successfully for Lexical Analyzer.

EX.NO.2 Conversion from Regular Expression to NFA

AIM:

To write a C program to convert the regular expression to NFA.

ALGORITHM:

1. Start the program.
2. Declare all necessary header files.
3. Define the main function.
4. Declare the variables and initialize variables r & c to '0'.
5. Use a for loop within another for loop to initialize the matrix for NFA states.
6. Get a regular expression from the user & store it in 'm'.
7. Obtain the length of the expression using strlen() function and store it in 'n'.
8. Use for loop upto the string length and follow steps 8 to 12.
9. Use switch case to check each character of the expression
10. If case is '*', set the links as 'E' or suitable inputs as per rules.
11. If case is '+', set the links as 'E' or suitable inputs as per rules.
12. .Check the default case, i.e.,for single alphabet or 2 consecutive alphabets and set the links to respective alphabet.
13. End the switch case.
14. Use for loop to print the states along the matrix.
15. Use a for loop within another for lop and print the value of respective links.
16. Print the states start state as '0' and final state.
17. End the program.

SOURCE CODE:

```
#include<stdio.h>

#include<string.h>
```

```
int main() {
        char reg[20];
        int q[20][3], i = 0, j = 1, len, a, b;
        for (a = 0; a < 20; a++)
        for (b = 0; b < 3; b++)
        q[a][b] = 0;
        scanf("%s", reg);
        printf("Given regular expression: %s\n", reg);
        len = strlen(reg);
        while (i < len) {
        if (reg[i] == 'a' && reg[i + 1] != '|' && reg[i + 1] != '*') {
        q[j][0] = j + 1;
        j++;
        }
        if (reg[i] == 'b' && reg[i + 1] != '|' && reg[i + 1] != '*') {
        q[j][1] = j + 1;
        j++;
        }
        if (reg[i] == 'e' && reg[i + 1] != '|' && reg[i + 1] != '*') {
        q[j][2] = j + 1;
        j++;
        }
        if (reg[i] == 'a' && reg[i + 1] == '|' && reg[i + 2] == 'b') {
```

```
q[j][2] = ((j + 1) * 10) + (j + 3);
j++;
q[j][0] = j + 1;
j++;
q[j][2] = j + 3;
j++;
q[j][1] = j + 1;
j++;
q[j][2] = j + 1;
j++;
i = i + 2;
}
if (reg[i] == 'b' && reg[i + 1] == '|' && reg[i + 2] == 'a') {
q[j][2] = ((j + 1) * 10) + (j + 3);
j++;
q[j][1] = j + 1;
j++;
q[j][2] = j + 3;
j++;
q[j][0] = j + 1;
j++;
q[j][2] = j + 1;
j++;
```

```
i = i + 2;
}
if (reg[i] == 'a' && reg[i + 1] == '*') {
q[j][2] = ((j + 1) * 10) + (j + 3);
j++;
q[j][0] = j + 1;
j++;
q[j][2] = ((j + 1) * 10) + (j - 1);
j++;
}
if (reg[i] == 'b' && reg[i + 1] == '*') {
q[j][2] = ((j + 1) * 10) + (j + 3);
j++;
q[j][1] = j + 1;
j++;
q[j][2] = ((j + 1) * 10) + (j - 1);
j++;
}
if (reg[i] == ')' && reg[i + 1] == '*') {
q[0][2] = ((j + 1) * 10) + 1;
q[j][2] = ((j + 1) * 10) + 1;
j++;
}
```

```
        i++;
        }
    printf("\n\tTransition Table\n");
    printf("_____\n");
        printf("Current State |\tInput |\tNext State");
    printf("\n_____\n");
        for (i = 0; i <= j; i++) {
        if (q[i][0] != 0) printf("\n  q[%d]\t     |   a   |  q[%d]", i, q[i][0]);
        if (q[i][1] != 0) printf("\n  q[%d]\t     |   b   |  q[%d]", i, q[i][1]);
        if (q[i][2] != 0) {
        if (q[i][2] < 10) printf("\n  q[%d]\t     |   e   |  q[%d]", i, q[i][2]);
        else printf("\n  q[%d]\t          |   e   |  q[%d] , q[%d]", i, q[i][2] / 10, q[i][2] % 10);
        }
        }
    printf("\n_____\n");
        return 0;
}
```

OUTPUT:

Given regular expression: (a|b)*

Transition Table

```
_____
Current State |   Input | Next State
_____

  q[0]              |   e   |  q[7] , q[1]
  q[1]              |   e   |  q[2] , q[4]
```

q[2]	a	q[3]
q[3]	e	q[6]
q[4]	b	q[5]
q[5]	e	q[6]
q[6]	e	q[7] , q[1]

RESULT:

Thus the C program to convert regular expression to NFA has been executed and the output has been verified successfully.

EX.NO : 3 Conversion from NFA to DFA

AIM:

To write a program to convert NFA to DFA

ALGORITHM:

1. Start the program
2. Assign an input string terminated by end of file, DFA with start
3. The final state is assigned to F
4. Assign the state to S
5. Assign the input string to variable C
6. While C!=e of do

   S=move(s,c)
   C=next char
7. If it is in ten return yes else no
8. Stop the program

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
int n[11], I, j, c, k, h, l, h1, f, h2, temp1[12], temp2[12], count = 0, ptr = 0;
char a[20][20], s[5][8], st[5], tr[5][2], ecl[5][8];
int flag;
```

```
void ecls(int b[10], int x) {
    int i = 0;
    int k = -1;
    flag = 0;
    while (i < x) {
    n[++k] = b[i];
    i = b[i];
    h = k + 1;
    a:
    for (int j = i; j <= 11; j++) {
                if (a[i][j] == 'e') {
                n[++k] = j;
                }
                if (j == 11 && h <= k) {
                i = n[h];
                h++;
                goto a;
                }
    }
    i++;
    }
    l++;
}
void mova(int g) {
    h1 = 0;
    for (int i = 0; i < 7; i++) {
    if (ecl[g][i] == 3) {
          temp1[h1++] = 4;
    }
    if (ecl[g][i] == 8) {
          temp1[h1++] = 9;
    }
    }
   printf("\nmove(%c,a):", st[g]);
    for (int i = 0; i < h1; i++) {
      printf("%d", temp1[i]);
    }
    f = 0;
    ecls(temp1, h1);
}
```

```
void movb(int g) {
    h2 = 0;
    for (int i = 0; i < 7; i++) {
    if (ecl[g][i] == 5) {
          temp2[h2++] = 6;
    }
    if (ecl[g][i] == 9) {
          temp2[h2++] = 10;
    }
    if (ecl[g][i] == 10) {
          temp2[h2++] = 11;
    }
    }
   printf("\nmove(%c,b):", st[g]);
    for (int i = 0; i < h2; i++) {
      printf("%d", temp2[i]);
    }
    f = 1;
    ecls(temp2, h2);
}

int main() {
    printf("\nthe no. of states in NFA (a/b)*abb are: 11");
    for (int i = 0; i <= 11; i++) {
    for (int j = 0; j <= 11; j++) {
              a[i][j] = '\0';
    }
    }
    a[1][2] = 'e';
    a[1][8] = 'e';
    a[2][3] = 'e';
    a[2][5] = 'e';
    a[3][4] = 'a';
    a[5][6] = 'b';
    a[4][7] = 'e';
    a[6][7] = 'e';
    a[7][8] = 'e';
    a[7][2] = 'e';
    a[8][9] = 'a';
    a[9][10] = 'b';
```

```
    a[10][11] = 'b';
   printf("\n\nThe transition table is as follows:\n");
    printf("   ");
    for (int i = 1; i <= 11; i++) {
       printf("%d ", i);
    }
   printf("\n");
    for (int i = 1; i <= 11; i++) {
       printf("%d  ", i);
    for (int j = 1; j <= 11; j++) {
           printf("%c ", a[i][j]);
    }
       printf("\n");
    }
    return 0;
}
```

OUTPUT:

The no. of states in NFA(a/b)*abb are:11

The transition table is as Follows

1	2	3	4	5	6	7	8	9	10	11
1	e		e							
2		e		e						
3		a								
4				e						
5				b	e					
6					e					
7		e			e					
8										
9					a					
10					b					
11					b					

RESULT:

Thus a C program is written successfully to convert NFA to DFA.

EX.NO: 4 Elimination of Ambiguity, Left Recursion and Left Factoring

AIM:

To Write a program elimination of Ambiguity, Left Recursion, and Left Factoring.

4a) Algorithm for ambiguity:-

1) Precedence:

a) Determine the precedence of different operators (e.g., +, -, *, /, etc.).

b) Operators with higher precedence should be evaluated first.

c) Adjust the order of productions to reflect operator precedence.

d) Use left recursion for left-associative operators and right recursion for right-associative operators.

2) Associativity:

a) Determine whether operators are left-associative or right-associative.

For example: +, -, *, and / are left-associative operators.

^ is a right-associative operator.

b) Adjust the order of productions to reflect operator associativity.

c) Use left recursion for left-associative operators and right recursion for right-associative operators.

SOURCE CODE:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#define MAX_PRODUCTIONS 100
typedef struct {
    char lhs;
    char rhs[20];
} Production;
bool isAmbiguous(Production productions[], int numProductions, char* input) {
    for (int i = 0; i < numProductions; i++) {
        for (int j = i + 1; j < numProductions; j++) {
            if (productions[i].lhs == productions[j].lhs) {
```

```
            char string1[100], string2[100];
            strcpy(string1, input);
            strcpy(string2, input);
            char* result1 = strstr(string1, &productions[i].lhs);
            char* result2 = strstr(string2, &productions[j].lhs);
            if (result1 && result2) {
                *result1 = productions[i].rhs[0];
                *result2 = productions[j].rhs[0];
                if (strcmp(string1, string2) != 0) {
                    return true;
                }
            }
        }
    }
    return false;
}
int main() {
    Production productions[MAX_PRODUCTIONS] = {
        {'S', "SaS"},
        {'S', ""},
        {'A', "AaB"},
        {'A', "a"},
        {'B', "Abb"},
        {'B', "b"}
    };
    int numProductions = 6;
    char input[100];
    printf("Enter a string: ");
    scanf("%s", input);
    bool ambiguous = isAmbiguous(productions, numProductions, input);
```

```
    if (ambiguous) {
        printf("The grammar is ambiguous for the input string '%s'.\n", input);
    } else {
        printf("The grammar is unambiguous for the input string '%s'.\n", input);
    }

    return 0;
}
```

OUTPUT:

Enter a string: aab

The grammar is unambiguous for the input string 'aab'.

4b) Algorithm for left recursion:-

1) Read the production rules (in the form of A -> α | β | ...).

2) For each production, split the right-hand side using the pipe

(|) symbol.

3) If any part starts with the same non-terminal symbol as the left-hand side, it has left recursion.

4) If left recursion is found:

4a) Create new productions to eliminate it:

4b) Replace A -> α | β with:

A -> βA'

A' -> αA' | ε (where ε represents an empty string)

5) If no left recursion, keep the original production.

6)Display the updated productions.

SOURCE CODE:

```
#include<stdio.h>

#include<string.h>
```

```
void main() {

char input[100],l[50],r[50],temp[10],tempprod[20],productions[25][50];

int i=0,j=0,flag=0,consumed=0;

printf("Enter the productions: ");

scanf("%1s->%s",l,r);

printf("%s",r);

while(sscanf(r+consumed,"%[^|]s",temp) == 1 && consumed <= strlen(r)) {

if(temp[0] == l[0]) {

flag = 1;

sprintf(productions[i++],"%s->%s%s'\0",l,temp+1,l);

} else

sprintf(productions[i++],"%s'->%s%s'\0",l,temp,l);

consumed += strlen(temp)+1;

}

if(flag == 1) {

sprintf(productions[i++],"%s->ε\0",l);

printf(" The productions after eliminating Left Recursion are:\n");

for(j=0;j<i;j++)

printf("%s\n",productions[j]);

} else
```

```
printf("The Given Grammar has no Left Recursion");

}
```

OUTPUT:

Enter the productions: E->E+E|T

E+E|T The productions after eliminating Left Recursion are:

E->+EE'

E'->TE'

E->ε

4c) Algorithm for left factoring:-

1. Read the production rule (e.g., A -> α | β) from the user.
2. Separate the left-hand side (LHS) and right-hand side (RHS) of the production.
3. Compare the characters of part1 and part2.
4. Find the longest common prefix (up to the first differing character).
5. Store this common prefix in modifiedGram.
6. Create a new non-terminal symbol (e.g., X).
7. Form the modified productions:

   A -> modifiedGram X

   X -> newGram | ε (where ε represents an empty string)
8. Store these modified productions in modifiedGram and newGram.
9. Display the modified productions without left factoring.

SOURCE CODE:

```
#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, pos;
    printf("Enter Production: A -> ");
    scanf("%s", gram);
    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';
    for (i = 0; i < strlen(part1) || i < strlen(part2); i++) {
    if (part1[i] == part2[i]) {
            modifiedGram[k] = part1[i];
            k++;
            pos = i + 1;      }    }
    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';
    modifiedGram[++k] = '\0';
    newGram[j] = '\0';
    printf("\nGrammar Without Left Factoring:\n");
    printf("A -> %s\n", modifiedGram);
    printf("X -> %s\n", newGram);
    return 0;
}
```

OUTPUT:

Enter Production : A->bE+acF|bE+f

Grammar Without Left Factoring : :

A->bE+X

X->acF|f

RESULT:

Thus a C program is written successfully for the elimination of Ambiguity, LeF]ft Recursion and Left Factoring.

EX.NO: 5 FIRST AND FOLLOW computation

AIM:

To calculate the first and Follow of the given expression

ALGORITHM:

1. Initialize necessary variables and data structures to store the productions, FIRST sets, and FOLLOW sets.
2. Prompt the user to input the number of productions.
3. Take input for each production and store them.
4. Iterate over each production rule.
5. If the first symbol after the arrow (`->`) is a terminal or epsilon, add it to the FIRST set of the left-hand side non-terminal.
6. If the first symbol is a non-terminal, recursively compute its FIRST set and add it to the FIRST set of the left-hand side non-terminal.
7. Start by setting FOLLOW sets of all non-terminals to empty.
8. The FOLLOW set of the start symbol usually contains the end-of-input marker `$`.
9. Iterate over each production rule.
10. For each non-terminal in the right-hand side of production:
    If it's followed by another non-terminal or epsilon, add the FIRST set of that non-terminal to its FOLLOW set.
    If it's the last symbol or followed by terminals, add the FOLLOW set of the left-hand side non-terminal to its FOLLOW set.
11. Print the computed FIRST sets for each non-terminal.
12. Print the computed FOLLOW sets for each non-terminal.

SOURCE CODE:

```
#include <stdio.h>

#include <conio.h>

void main() {

    int nop, x=0, y=0, l=0, k=0, c=0, s=0, z=0;

    char p[10][10], fi[10][10], fo[10][10];

    for(x=0; x<10; x++) {

    for(y=0; y<10; y++) {
```

```
            p[x][y]='0';
            fi[x][y]='0';
            fo[x][y]='0';
}
}
printf("Enter the no of productions: ");
scanf("%d", &nop);
printf("\nEnter the productions:\n");
for(x=0; x<nop; x++) {
scanf("%s", p[x]);
}
printf("\nFirst\n");
for(y=0; y<nop; y++) {
   printf("First(%c)={", p[y][0]);
if(p[y][2] == 'e' || (p[y][2] >= 'a' && p[y][2] <= 'z') || (p[y][2] >= 'A' && p[y][2] <= 'Z')) {
       printf("%c", p[y][2]);
       fi[y][k++]=p[y][2];
}
else if(p[y][2] == '(' || (p[y][2] >= 'a' && p[y][2] <= 'z') || (p[y][2] >= 'A' && p[y][2] <= 'Z')) {
       printf("%c", p[y][2]);
```

```
            fi[y][k++]=p[y][2];
    }
       printf("}\n");
    }
   printf("\nFollow\n");
    for(x=0; x<nop; x++) {
       printf("Follow(%c)={", p[x][0]);
    if(x == 0) {
            printf("$");
    }
       printf("}\n");
    }
    getch();
}
```

OUTPUT:

Enter the no. of production:5
E->TE'
E'->+TE'/e
T->FT'
T'->*FT'/e
F->(E) /id

First

First(E)={(,id}
First(E')={+,e}
First(T)={(,id}
First(T')={*,e}
First(F)={(,id}

Follow
Follow(E)={),$}
Follow(E')={),+,$}
Follow(T)={),+,$}
Follow(T')={*,+,),$}
Follow(F)={*,+,),$}

RESULT:

Thus a C program is written successfully for computation of FIRST and FOLLOW sets.

EX.NO :6 Predictive Parsing Table

AIM: To write a C program for the implementation of predictive parsing table

ALGORITHM:

1. Define necessary variables and data structures:
2. Implement stack operations: pop, push, stop, and pobo.
3. Implement the `search` function to find the appropriate production rule from the parsing table.
4. In the `main` function:
5. Initialize the stack and parsing process.
6. Display the parsing table.
7. Take input string from the user.
8. Begin parsing process:
9. Iterate over the input string.
10. Match terminal and non-terminal symbols with the parsing table.
11. Update stack and output based on the parsing table rules.
12. Print the current stack, input, and output.
13. Handle errors if any.
14. Print the final result indicating whether the input string is accepted or not.

SOURCE CODE:

```
 #include <stdio.h>

#include <string.h>

char str[10], out, in, output[10], input[10], temp;

char tl[7] = {'x', '+', '*', '(', ')', '$', '@'};

char ntl[5] = {'e', 'e', 't', 't', 'f'};

int err = 0, flag = 0, i, j, k, m;

char c[5][6][7] = {

    {"te", "error!", "error!", "te", "error!", "error!"},

    {"error!", "te", "error!", "error", "@", "@"},

    {"ft", "error!", "error!", "ft", "error!", "error!"},

    {"error!", "@", "*ft", "error!", "@", "@"},

    {"x", "error!", "error!", "(e)", "error!", "error!"}};

 struct stack

{

    char sic[10];

    int top;

};

 void push(struct stack *s, char p)
```

```
{
    s->sic[++s->top] = p;
    s->sic[s->top + 1] = '\0';
}
 char pop(struct stack *s)
{
    char a;
    a = s->sic[s->top];
    s->sic[s->top--] = '\0';
    return (a);
}
 char stop(struct stack *s)
{
    return (s->sic[s->top]);
}
 void pobo(struct stack *s)
{
    m = 0;
    while (str[m] != '\0')
    m++;
```

```
    m--;
    while (m != -1)
    {
    if (str[m] != '@')
            push(s, str[m]);
    m--;
    }
}
 void search(int l)
{
    for (k = 0; k < 7; k++)
    if (in == tl[k])
            break;
    if (l == 0)
    strcpy(str, c[l][k]);
    else if (l == 1)
    strcpy(str, c[l][k]);
    else if (l == 2)
    strcpy(str, c[l][k]);
    else if (l == 3)
```

```
    strcpy(str, c[l][k]);
    else
    strcpy(str, c[l][k]);
}
 int main()
{
    struct stack s1;
    struct stack *s;
    s = &s1;
    s->top = -1;

    printf("\t\t Parsing Table \t\t");
    for (i = 0; i < 5; i++)
    {
       printf("%c\t", ntl[i]);
    for (j = 0; j < 6; j++)
              if (strcmp(c[i][j], "error!") == 0)
             printf("error!\t");
              else
             printf("%c->%s\t", ntl[i], c[i][j]);
```

```
   printf("\n");
}
push(s, '$');
push(s, 'e');
printf("Enter the input string: ");
scanf("%s", input);
printf("\n\nThe behavior of the parser for the given input string is: \n");
printf("\nStack\tInput\tOutput");
i = 0;
in = input[i];
printf("%s\t", s->sic);
for (int k = i; k < strlen(input); k++)
printf("%c", input[k]);
if (strcmp(str, " ") != 0)
   printf("\t%c->%s", ntl[j], str);
while ((s->sic[s->top] != '$') && err != 1 && strcmp(str, "error!") != 0)
{
strcpy(str, " ");
flag = 0;
for (j = 0; j < 7; j++)
```

```
        if (in == tl[j])
        {
        flag = 1;
        break;
        }
if (flag == 0)
        in = 'x';
flag = 0;
out = stop(s);
for (j = 0; j < 7; j++)
        if (out == tl[j])
        {
        flag = 1;
        break;
        }
if (flag == 1)
{
        if (out == in)
        {
        temp = pop(s);
```

```
        in = input[++i];
        if (strcmp(str, "@") == 0)
        temp = pop(s);
        }
        else
        {
        strcpy(str, "error!");
        err = 1;
        }
}
else
{
        flag = 0;
        for (j = 0; j < 5; j++)
        if (out == ntl[j])
        {
        flag = 1;
        break;
        }
        if (flag == 1)
```

```
            {
            search(j);
            temp = pop(s);
            pobo(s);
            }
            else
            {
            strcpy(str, "error!");
            err = 1;
            }
}
if (strcmp(str, "error!") != 0)
{
       printf("%s\t", s->sic);
            for (int k = i; k < strlen(input); k++)
           printf("%c", input[k]);
            if ((strcmp(str, " ") != 0) && (strcmp(str, "error!") != 0))
           printf("\t%c->%s", ntl[j], str);
}
}
```

```
    if (strcmp(str, "error!") == 0)

    printf("\nThe string is not accepted!!");

    else

       printf("\t\tAccepted.\n\n\nThe string is accepted.");

    return 0;

}
```

OUTPUT:

Parsing table

X	+	*	(	)	$	@
E	E->Te	ERROR!	ERROR!	E->te	ERROR!	ERROR!
E	ERROR!	E->+te	ERROR!	ERROR!	E->@	e->@
T	T->Ft	ERROR!	ERROR!	T->ft	ERROR!	ERROR!
T	ERROR!	T->@	t->*ft	ERROR	t->@	t->@
F	F->x	ERROR!	ERROR!	F-> (E)	ERROR!	ERROR!

Enter the input string: x+X$
The behavior of the parser for given input string is

Stack	input	output
SE	X+X$	
SeT	X+X$	E->Te
Set F	X+X$	T->Ft
Set X	X+X$	F->X
Set	+X$	
Se	+X$	t->@
SeT+	+X$	e->+Te
seT	X$	
SetF	X$	T->Ft

RESULT:

Thus a C program is written successfully for the construction of a Predictive Parsing Table.

EX.NO: 7 **Shift Reduce Parsing**

AIM:

To write a C program for shift-reduce parsing.

ALGORITHM:

1. Define necessary variables and arrays.
2. Implement the `check` function to perform reductions:
3. Check for the presence of patterns in the stack to perform reductions.
4. If the stack contains `4`, reduce to `E`.
5. If the stack contains `2E2` or `3E3`, reduce to `E`.
6. In the `main` function:
7. Display the grammar rules.
8. Initialize the input string `a`.
9. Determine the length of the input string.
10. Display the initial stack, input, and action.
11. Iterate through the input string:
12. Shift symbols from input to stack.
13. Display the current stack, input, and action.
14. Check for reductions.
15. After processing the input string, check for additional reductions.
16. If the stack contains only `E` and is empty, accept the input; otherwise, reject it.

SOURCE CODE:

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int z = 0, i = 0, j = 0, c = 0;

char a[16], ac[20], stk[15], act[10];

void check()

{
```

```
strcpy(ac,"REDUCE TO E -> ");
for(z = 0; z < c; z++)
{
        if(stk[z] == '4')
        {
                printf("%s4", ac);
                stk[z] = 'E';
                stk[z + 1] = '\0';
                printf("\n$%s\t%s$\t", stk, a);
        }
}
for(z = 0; z < c - 2; z++)
{
        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$%s\t%s$\t", stk, a);
```

```
                i = i - 2;
            }
    }
    for(z=0; z<c-2; z++)
    {
            if(stk[z] == '3' && stk[z + 1] == 'E' &&
                                                    stk[z + 2] == '3')
            {
                printf("%s3E3", ac);
                stk[z]='E';
                stk[z + 1]='\0';
                stk[z + 1]='\0';
                printf("\n$%s\t%s$\t", stk, a);
                i = i - 2;
            }
    }
    return ;
}
int main()
{
```

```
printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");

strcpy(a,"32423");

c=strlen(a);

strcpy(act,"SHIFT");

printf("\nstack \t input \t action");

printf("\n$\t%s$\t", a);

for(i = 0; j < c; i++, j++)

{

                printf("%s", act);

                stk[i] = a[j];

                stk[i + 1] = '\0';

                a[j]=' ';

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

                check();

}

check();

if(stk[0] == 'E' && stk[1] == '\0')

                printf("Accept\n");

else //else reject

                printf("Reject\n");
```

}

OUTPUT:

GRAMMAR is -

E->2E2

E->3E3

E->4

stack	input	action
$	32423$	SHIFT
$3	2423$	SHIFT
$32	423$	SHIFT
$324	23$	REDUCE TO E -> 4
$32E	23$	SHIFT
$32E2	3$	REDUCE TO E -> 2E2
$3E	3$	SHIFT
$3E3	$	REDUCE TO E -> 3E3
$E	$	Accept

RESULT:

Thus a C program is written successfully for the implementation of Shift Reduce Parsing.

EX.NO: 8 Computation of LEADING AND TRAILING

AIM:

To write a C program to Compute of Leading and Trailing

ALGORITHM:

1. Get the no of production and calculate the length of each production.
2. With a variable val for checking the valid non-terminals if they are duplicated get all Non-terminals in an array.
3. In each production check the first accurate of terminals and take that terminal and add it to Follow of non terminal in array and exitthe loop.
4. Scan the production and find the last terminal and add it to respective trailing array of associated non terminal &exit the loop.
5. Consider production with a non terminal on right side and check the Follow of that non terminal associated production and also trailing and add it to the Follow and trailing of left side non terminal.
6. Write the Follow and trailing terminals for each non terminal.

SOURCE CODE:

```
#include<stdio.h>

#include<string.h>

#include<conio.h>


int nt, t, top = 0;

char s[50], NT[10], T[10], st[50], l[10][10], tr[50][50];

 int searchnt(char a) {

    int count = -1, i;

    for (i = 0; i < nt; i++) {

    if (NT[i] == a)
```

```
            return i;
    }
    return count;
}
 int searchter(char a) {
    int count = -1, i;
    for (i = 0; i < t; i++) {
    if (T[i] == a)
            return i;
    }
    return count;
}
void push(char a) {
    s[top] = a;
    top++;
}
 char pop() {
    top--;
    return s[top];
}
 void installl(int a, int b) {
    if (l[a][b] == 'f') {
    l[a][b] = 't';
```

```
    push(T[b]);
    push(NT[a]);
    }
}
 void installt(int a, int b) {
    if (tr[a][b] == 'f') {
    tr[a][b] = 't';
    push(T[b]);
    push(NT[a]);
    }
}
 int main() {
    int i, s, k, j, n;
    char pr[30][30], b, c;
    clrscr();
    printf("Enter the no of productions:");
    scanf("%d", &n);
    printf("Enter the productions one by one\n");
    for (i = 0; i < n; i++)
    scanf("%s", pr[i]);
    nt = 0;
    t = 0;
    for (i = 0; i < n; i++) {
```

```
if ((searchnt(pr[i][0])) == -1)
        NT[nt++] = pr[i][0];
}
for (i = 0; i < n; i++) {
for (j = 3; j < strlen(pr[i]); j++) {
        if (searchnt(pr[i][j]) == -1) {
        if (searchter(pr[i][j]) == -1)
        T[t++] = pr[i][j];
        }
}
}
for (i = 0; i < nt; i++) {
for (j = 0; j < t; j++)
        l[i][j] = 'f';
}
for (i = 0; i < nt; i++) {
for (j = 0; j < n; j++) {
        if (NT[(searchnt(pr[j][0]))] == NT[i]) {
        if (searchter(pr[j][3]) != -1)
           installl(searchnt(pr[j][0]), searchter(pr[j][3]));
        else {
        for (k = 3; k < strlen(pr[j]); k++) {
                if (searchnt(pr[j][k]) == -1) {
```

```
                installl(searchnt(pr[j][0]), searchter(pr[j][k]));
                break;
                 }
        }
        }
        }
}
}
while (top != 0) {
b = pop();
c = pop();
for (s = 0; s < n; s++) {
         if (pr[s][3] == b)
        installl(searchnt(pr[s][0]), searchter(c));
}
}
for (i = 0; i < nt; i++) {
   printf("Leading[%c]\t{", NT[i]);
for (j = 0; j < t; j++) {
         if (l[i][j] == 't')
        printf("%c ", T[j]);
}
   printf("}\n");
```

```
}
top = 0;
for (i = 0; i < nt; i++) {
for (j = 0; j < n; j++) {
        if (NT[searchnt(pr[j][0])] == NT[i]) {
        if (searchter(pr[j][strlen(pr[j]) - 1]) != -1)
          installt(searchnt(pr[j][0]), searchter(pr[j][strlen(pr[j]) - 1]));
        else {
        for (k = (strlen(pr[j]) - 1); k >= 3; k--) {
                 if (searchnt(pr[j][k]) == -1) {
                installt(searchnt(pr[j][0]), searchter(pr[j][k]));
                break;
                 }
        }
        }
        }
}
}
while (top != 0) {
b = pop();
c = pop();
for (s = 0; s < n; s++) {
        if (pr[s][3] == b)
```

```
            installt(searchnt(pr[s][0]), searchter(c));
    }
    }
    for (i = 0; i < nt; i++) {
       printf("Trailing[%c]\t{", NT[i]);
    for (j = 0; j < t; j++) {
             if (tr[i][j] == 't')
            printf("%c ", T[j]);
    }
       printf("}\n");
    }
    getch();
    return 0;
}
```

OUTPUT:

Enter the no. of production:5
E->TE’
E’->+TE’/e
T->FT’
T’->*FT’/e
F->(E) /id

Leading
Leading (E)={(,id}
Leading (E’)={+,e}
Leading (T)={(,id}
Leading (T’)={*,e}
Leading (F)={(,id}

Trailing
Trailing (E)={),$}
Trailing (E')={),+,$}
Trailing (T)={),+,$}
Trailing (T')={*,+,),$}
Trailing (F)={*,+,),$}

RESULT:

Thus a C program is written successfully for the computation of Leading and Trailing.

EX.NO.9 Computation of LR(0) items

AIM:

To write a code for LR(0) Parser for Following Production:

E->E+T

T->T*F/F

F->(E)/char

ALGORITHM:

1. Define necessary variables and constants:
2. Implement the `shift` function to handle shifting operations:
3. Add the next state and symbol to the stack.
4. Implement the `reduce` function to handle reduction operations:
5. Determine the rule to be applied based on the reduction type.
6. Pop the necessary symbols from the stack.
7. Push the resulting symbol and state onto the stack.
8. Implement the `parseInput` function to parse the input string:
9. Iterate through the input string character by character.
10. Determine the current symbol.
11. Look up the action in the parsing table.
12. If it's a shift operation, call the `shift` function.
13. If it's a reduction operation, call the `reduce` function.
14. Print the stack, input, and action at each step.
15. Print "Accepted" if the input string is successfully parsed.

SOURCE CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define NUM_STATES 8

#define NUM_SYMBOLS 5

enum Symbols {

    SYMBOL_E = 0,

    SYMBOL_T,

    SYMBOL_F,

    SYMBOL_PLUS,

    SYMBOL_MULTIPLY

};

int parsingTable[NUM_STATES][NUM_SYMBOLS] = {

    {1, 2, 3, -1, -1},

    {-1, -1, -1, 4, -1},

    {-1, -1, -1, -1, 5},

    {6, 2, 3, -1, -1},

    {-1, -1, -1, -1, 7},

    {-1, -1, -1, -1, -1},
```

```
    {1, 2, 3, -1, -1},
    {-1, -1, -1, -1, -1}
};
void shift(int *stateStack, int *symbolStack, int *top, int nextState, int symbol);
void reduce(int *stateStack, int *symbolStack, int *top, int rule);
void parseInput();
int main() {
    parseInput();
    return 0;
}
void shift(int *stateStack, int *symbolStack, int *top, int nextState, int symbol) {
    stateStack[++(*top)] = nextState;
    symbolStack[*top] = symbol;
}
void reduce(int *stateStack, int *symbolStack, int *top, int rule) {
    int i;
    switch (rule) {
    case 1:
            for (i = 0; i < 3; i++) {
            (*top)--;
```

```
        }
        stateStack[*top + 1] = parsingTable[stateStack[*top]][SYMBOL_E];
        symbolStack[*top + 1] = SYMBOL_E;
        (*top)++;
        break;
case 2:
        for (i = 0; i < 3; i++) {
        (*top)--;
        }
        stateStack[*top + 1] = parsingTable[stateStack[*top]][SYMBOL_T];
        symbolStack[*top + 1] = SYMBOL_T;
        (*top)++;
        break;
case 3:
        for (i = 0; i < 3; i++) {
        (*top)--;
        }
        stateStack[*top + 1] = parsingTable[stateStack[*top]][SYMBOL_F];
        symbolStack[*top + 1] = SYMBOL_F;
        (*top)++;
```

```
            break;
    }
}
void parseInput() {
    char input[] = "a+b*c";
    int stateStack[100], symbolStack[100], top = 0;
    stateStack[0] = 0;

    printf("Enter input string: %s\n", input);
    printf("Stack\tInput\tAction\n");
    printf("-----\t-----\t------\n");
    int i = 0;
    while (input[i] != '\0') {
    for (int j = 0; j <= top; j++) {
        printf("%d%c", stateStack[j], symbolStack[j] != -1 ? symbolStack[j] + 'A' : ' ');
    }
    printf("\t%s\t", input + i);
    char currentSymbol = input[i];
    int symbol;
    switch (currentSymbol) {
```

```
            case 'a':
            symbol = SYMBOL_E;
            break;
            case 'b':
            symbol = SYMBOL_T;
            break;
            case 'c':
            symbol = SYMBOL_F;
            break;
            case '+':
            symbol = SYMBOL_PLUS;
            break;
            case '*':
            symbol = SYMBOL_MULTIPLY;
            break;
            default:
           printf("\nInvalid symbol\n");
            return;
}
int action = parsingTable[stateStack[top]][symbol];
```

```
    if (action == -1) {
        printf("\nError\n");
            return;
    } else if (action > 0 && action < NUM_STATES) {
            shift(stateStack, symbolStack, &top, action, symbol);
            i++;
        printf("Shift\n");
    } else {
            reduce(stateStack, symbolStack, &top, abs(action));
        printf("Reduce by rule %d\n", abs(action));
    }
    }
   printf("Accepted\n");
}
```

OUTPUT

Enter any String :- a+b*c

```
0                a+b*c
0a5              +b*c
0F3              +b*c
0T2              +b*c
0E1              +b*c
0E1+6             b*c
```

0E1+6b5 *c

0E1+6F3 *c

0E1+6T9 *c

0E1+6T9*7 c

0E1+6T9*7c5

RESULT:

Thus a C program is written successfully for the computation of LR(0) items.

EX.NO.10 Intermediate Code Generation - Postfix, Prefix

AIM:

To write a C program for the implementation of code generation - postfix, prefix.

Algorithm:

1. Declare a set of operators that you'll encounter in the expression.
2. Initialize an empty stack.
3. Initialize an empty list to store the intermediate code.
4. Scan the infix expression from left to right.
5. For each character in the expression:
6. If it's an operand, add it to the intermediate code list.
7. If it's an operator, follow these steps:
8. Pop all operators from the stack that have greater precedence than the scanned operator (or if the stack is empty or contains a '('). -Push the scanned operator onto the stack.
9. After scanning the entire expression, pop any remaining operators from the stack and add them to the intermediate code list.
10. The final postfix expression can be obtained by concatenating the elements in the intermediate code list.
11. To obtain the prefix expression, reverse the intermediate code list.
12. Output the postfix and prefix representations.

SOURCE CODE:

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

#define MAX_SIZE 100

// Stack structure
typedef struct {
    int top;
    char items[MAX_SIZE];
} Stack;

// Function prototypes
void push(Stack *s, char value);
char pop(Stack *s);
int isOperator(char c);
int precedence(char c);
void infixToPostfix(char *infix, char *postfix);
void infixToPrefix(char *infix, char *prefix);

int main() {
    char infixExpression[MAX_SIZE];
    char postfixExpression[MAX_SIZE];
    char prefixExpression[MAX_SIZE];

    // Input infix expression
    printf("Enter infix expression: ");
    scanf("%s", infixExpression);

    infixToPostfix(infixExpression, postfixExpression);
    infixToPrefix(infixExpression, prefixExpression);

    printf("Postfix expression: %s\n", postfixExpression);
    printf("Prefix expression: %s\n", prefixExpression);

    return 0;
}
```

```
// Function to push an item onto the stack
void push(Stack *s, char 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
char pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    }
    return s->items[s->top--];
}

// Function to check if a character is an operator
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Function to return precedence of operators
int precedence(char c) {
    if (c == '*' || c == '/')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return 0;
}

// Function to convert infix expression to postfix expression
void infixToPostfix(char *infix, char *postfix) {
    Stack s;
    s.top = -1;
    int i = 0, j = 0;

    while (infix[i] != '\0') {
```

```
        if (isalnum(infix[i])) {
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            push(&s, infix[i]);
        } else if (infix[i] == ')') {
            while (s.top != -1 && s.items[s.top] != '(') {
                postfix[j++] = pop(&s);
            }
            if (s.top == -1) {
                printf("Invalid infix expression\n");
                exit(EXIT_FAILURE);
            }
            pop(&s); // Discard the '('
        } else if (isOperator(infix[i])) {
            while (s.top != -1 && precedence(s.items[s.top]) >= precedence(infix[i])) {
                postfix[j++] = pop(&s);
            }
            push(&s, infix[i]);
        }
        i++;
    }

    while (s.top != -1) {
        if (s.items[s.top] == '(') {
            printf("Invalid infix expression\n");
            exit(EXIT_FAILURE);
        }
        postfix[j++] = pop(&s);
    }
    postfix[j] = '\0';
}

// Function to convert infix expression to prefix expression
void infixToPrefix(char *infix, char *prefix) {
    Stack s;
    s.top = -1;
    char temp[MAX_SIZE];
    int i, j = 0;

    // Reverse the infix expression
```

```
    for (i = strlen(infix) - 1; i >= 0; i--) {
        if (infix[i] == '(')
            temp[j++] = ')';
        else if (infix[i] == ')')
            temp[j++] = '(';
        else
            temp[j++] = infix[i];
    }
    temp[j] = '\0';

    infixToPostfix(temp, prefix);

    // Reverse the postfix expression to get the prefix expression
    j = strlen(prefix) - 1;
    for (i = 0; i < j; i++, j--) {
        char tempChar = prefix[i];
        prefix[i] = prefix[j];
        prefix[j] = tempChar;
    }
}
```

OUTPUT-

Enter infix expression: a*b+c/d-e/f+g*h

Postfix expression: ab*cd/+ef/-gh*+

Prefix expression: +*ab-/cd+/ef*gh

RESULT:

Thus a C program is written successfully for the implementation of Intermediate code generation Postfix, Prefix.

EX.NO: 11 Intermediate Code Generation - Quadruple, Triple, Indirect Triple

AIM:

To write a C program for the implementation of code generation - quadruple, triple, indirect triple.

ALGORITHM:

1. Define a set of operators.
2. Initialize an empty stack.
3. Initialize an empty list for intermediate code.
4. Scan the infix expression left to right.
5. For each character:
6. If operand, add to the intermediate code list.
7. If operator:
   Pop operators from the stack with higher precedence.
   Push scanned operator onto stack.
8. After scanning, pop the remaining operators and add them to the intermediate code.
9. Concatenate intermediate code list for postfix expression.
10. Reverse intermediate code list for prefix expression.
11. Convert the intermediate code list to quadruples/triples:
    Use a counter for temporary variables
12. For each element:
    If operand, output directly.
    If operator, create quadruple/triple and add.
13. Output postfix and prefix representations.

SOURCE CODE:

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

#define MAX_SIZE 100

// Structure for Quadruple
typedef struct {
   char op;
   char arg1[MAX_SIZE];
   char arg2[MAX_SIZE];
   char result[MAX_SIZE];
} Quadruple;
```

```
// Structure for Triple
typedef struct {
    char op[3]; // To store operator as a string
    char arg1[MAX_SIZE];
    char arg2[MAX_SIZE];
} Triple;

// Structure for Indirect Triple
typedef struct {
    char op[3]; // To store operator as a string
    int arg1;
    int arg2;
} IndirectTriple;

// Function prototypes
void generateQuadruples(char *expression);
void generateTriples(char *expression);
void generateIndirectTriples(char *expression);

int main() {
    char expression[MAX_SIZE];

    // Input expression
    printf("Enter the expression: ");
    fgets(expression, MAX_SIZE, stdin);

    generateQuadruples(expression);
    generateTriples(expression);
    generateIndirectTriples(expression);

    return 0;
}

// Function to generate quadruples
void generateQuadruples(char *expression) {
    Quadruple quadruples[MAX_SIZE];
    int index = 0;

    // Tokenize the expression
```

```
    char *token = strtok(expression, "+-*/");

    while (token != NULL) {
        Quadruple quad;
        quad.op = expression[strlen(token)];
        strcpy(quad.arg1, token);
        strcpy(quad.arg2, token + 2);
        sprintf(quad.result, "T%d", index + 1);
        quadruples[index++] = quad;
        token = strtok(NULL, "+-*/");
    }

    // Print the quadruples
    printf("Quadruples:\n");
    for (int i = 0; i < index; i++) {
        printf("(%c, %s, %s, %s)\n", quadruples[i].op, quadruples[i].arg1, quadruples[i].arg2,
quadruples[i].result);
    }
}

// Function to generate triples
void generateTriples(char *expression) {
    Triple triples[MAX_SIZE];
    int index = 0;

    // Tokenize the expression
    char *token = strtok(expression, "+-*/");

    while (token != NULL) {
        Triple triple;
        strcpy(triple.op, &expression[strlen(token)]);
        strcpy(triple.arg1, token);
        strcpy(triple.arg2, token + 2);
        triples[index++] = triple;
        token = strtok(NULL, "+-*/");
    }

    // Print the triples
    printf("\nTriples:\n");
    for (int i = 0; i < index; i++) {
```

```
        printf("(%s, %s, %s)\n", triples[i].op, triples[i].arg1, triples[i].arg2);
    }
}

// Function to generate indirect triples
void generateIndirectTriples(char *expression) {
    IndirectTriple indirectTriples[MAX_SIZE];
    int index = 0;

    // Tokenize the expression
    char *token = strtok(expression, "+-*/");

    while (token != NULL) {
        IndirectTriple indirectTriple;
        strcpy(indirectTriple.op, &expression[strlen(token)]);
        indirectTriple.arg1 = index;
        indirectTriple.arg2 = index + 1;
        indirectTriples[index++] = indirectTriple;
        token = strtok(NULL, "+-*/");
    }

    // Print the indirect triples
    printf("\nIndirect Triples:\n");
    for (int i = 0; i < index; i++) {
        printf("(%s, %d, %d)\n", indirectTriples[i].op, indirectTriples[i].arg1, indirectTriples[i].arg2);
    }
}
```

OUTPUT:

Enter the expression: a + b x c / e ↑ f + b x c

Quadruples:
(, a , , T1)
(, b x c , x c , T2)
(, e ↑ f , ↑ f , T3)
(, b x c, x c, T4)

Triples:
(, a ,)

Indirect Triples:
(, 0, 1)

RESULT:

Thus a C program is written successfully for the implementation of Intermediate code generation - quadruple, triple, indirect triple.

EX.NO : 12 A simple code Generator

AIM:

To write a C program to implement a simple code generator.

ALGORITHM:

1. The input consists of an array of strings, where each string represents a three-address statement.
2. Each statement contains an operator (e.g., +, -, *, /), two operands (e.g., a, b), and a result (e.g., t1,x).
3. Initialize a register descriptor to keep track of available registers (e.g., R0, R1, etc.).
4. Set up address descriptors for memory locations (e.g., getReg(y), getReg(z)).
5. For each three-address statement:

   Parse the statement to extract the operator, operands, and result.

   Based on the operator:-Perform the corresponding operation (e.g., addition, multiplication, subtraction, division).
6. Store the result in the specified memory location (e.g., x).
7. Generate machine instructions (assembly-like code) based on the processed statements.
8. Use the register and address descriptors to guide instruction generation.

SOURCE CODE:

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

```
void intermediateCodeGen();
int main() {
intermediateCodeGen();
return 0;
}
void intermediateCodeGen() {
char op[2], arg1[5], arg2[5], result[5];
int i = 0;
char *input[] = {"+ a b t1", "* c d t2", "- t1 t2 t", "= t ? x", NULL};
printf("Intermediate Code Generation:\n");
while (input[i] != NULL) {
sscanf(input[i], "%s%s%s%s", op, arg1, arg2, result);
if (strcmp(op, "+") == 0) {
printf("MOV R0,%s\n", arg1);
printf("ADD R0,%s\n", arg2);
printf("MOV %s,R0\n", result);
} else if (strcmp(op, "*") == 0) {
printf("MOV R0,%s\n", arg1);
printf("MUL R0,%s\n", arg2);
printf("MOV %s,R0\n", result);
} else if (strcmp(op, "-") == 0) {
printf("MOV R0,%s\n", arg1);
printf("SUB R0,%s\n", arg2);
printf("MOV %s,R0\n", result);
} else if (strcmp(op, "/") == 0) {
printf("MOV R0,%s\n", arg1);
printf("DIV R0,%s\n", arg2);
printf("MOV %s,R0\n", result);
} else if (strcmp(op, "=") == 0) {
printf("MOV R0,%s\n", arg1);
printf("MOV %s,R0\n", result);
}
i++;
}
}
```

OUTPUT:

```
Intermediate Code Generation:
MOV R0,a
ADD R0,b
```

```
MOV t1,R0
MOV R0,c
MUL R0,d
MOV t2,R0
MOV R0,t1
SUB R0,t2
MOV t,R0
MOV R0,t
MOV x,R0
```

RESULT:

Thus a C program is written successfully for the implementation of a simple code generator.

EX.NO.13 Implementation of DAG

AIM:

To write a C program to perform the implementation of DAG

ALGORITHM:

1. Define the maximum number of edges (EDGES) in the graph (e.g., 20).
2. Create a structure called Edge to represent an edge with two fields: source and dest.
3. Initialize an array of Edge structures called edges.
4. For each edge (from 0 to EDGES - 1):
5. Generate random source and destination vertices (within the range [0, 9]).
6. Store the source and destination vertices in the corresponding fields of the Edge structure.
7. Repeat this process for all edges.
8. Sort the edges array based on the source vertex in ascending order.
9. Use the qsort function with a custom comparison function (compareEdges) to achieve this.
10. Print the sorted edges in the format: source -> destination;.
11. Display the edges of the directed acyclic graph (DAG).

SOURCE CODE:

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

```
#include<time.h>
#define EDGES 20
typedef struct {
int source;
int dest;
} Edge;
int compareEdges(const void* a, const void* b) {
Edge* edge1 = (Edge*)a;
Edge* edge2 = (Edge*)b;
return (edge1->source - edge2->source);
}
int main() {
int i;
Edge edges[EDGES];
srand(time(NULL));
printf("DIRECTED ACYCLIC GRAPH\n");
for (i = 0; i < EDGES; i++) {
edges[i].source = rand() % 10;
edges[i].dest = rand() % 10;
}
qsort(edges, EDGES, sizeof(Edge), compareEdges);
for (i = 0; i < EDGES; i++) {
printf("%d -> %d;\n", edges[i].source, edges[i].dest);
}

return 0;
}
```

OUTPUT:

```
DIRECTED ACYCLIC GRAPH
0 -> 6;
0 -> 9;
0 -> 5;
0 -> 4;
1 -> 3;
1 -> 9;
1 -> 6;
1 -> 8;
2 -> 6;
2 -> 7;
3 -> 6;
```

3 -> 3;
6 -> 3;
7 -> 4;
8 -> 2;
8 -> 6;
8 -> 4;
8 -> 2;
9 -> 9;
9 -> 2;

RESULT:

Thus a C program is written successfully for the implementation of DAG.

EX.NO.14 Implementation of Global Data Flow Analysis

AIM:

To write a C program for the Implementation of Global Data Flow Analysis.

ALGORITHM:

1. Prompt the user to enter the number of expressions.
2. Input each expression (operator, operand1, operand2, result) and store them.
3. Iterate through each expression.
4. If both operands are constants, calculate the result based on the operator.
5. Update the result of the expression if it's a constant.
6. Iterate through each pair of expressions.
7. If two expressions have the same operator and operands in any order:
8. Mark the second expression as redundant.
9. Update subsequent expressions that use the result of the redundant expression.
10. Print the optimized code, excluding redundant expressions.
11. Call the input function.
12. Call the constant folding function.
13. Call the expression optimization function.
14. Call the output function to print the optimized code.

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
void input();
```

```
void output();
void change(int p, int q, char *res);
void constant();
void expression();
struct expr {
char op[2], op1[5], op2[5], res[5];
int flag;
} arr[10];
int n;
int main() {
input();
constant();
expression();
output();
return 0;
}
void input() {
printf("\nEnter the number of expressions: ");
scanf("%d", &n);
printf("\nEnter the input expressions: \n");
for (int i = 0; i < n; i++) {
scanf("%s %s %s %s", arr[i].op, arr[i].op1, arr[i].op2, arr[i].res);
arr[i].flag = 0;
}
}
void constant() {
int op1, op2, res;
char op, res1[5];
for (int i = 0; i < n; i++) {
if (isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0])) {
op1 = atoi(arr[i].op1);
op2 = atoi(arr[i].op2);
op = arr[i].op[0];
switch(op) {
case '+':
res = op1 + op2;
break;
case '-':
res = op1 - op2;
break;
```

```
case '*':
res = op1 * op2;
break;
case '/':
res = op1 / op2;
break;
}
sprintf(res1, "%d", res);
arr[i].flag = 1;
change(i, i, res1);
}
}
}
void expression() {
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (strcmp(arr[i].op, arr[j].op) == 0) {
if (strcmp(arr[i].op, "+") == 0 || strcmp(arr[i].op, "*") == 0) {
if ((strcmp(arr[i].op1, arr[j].op1) == 0 && strcmp(arr[i].op2, arr[j].op2) == 0) ||
(strcmp(arr[i].op1, arr[j].op2) == 0 && strcmp(arr[i].op2, arr[j].op1) == 0)) {
arr[j].flag = 1;
change(i, j, NULL);
}
} else {
if (strcmp(arr[i].op1, arr[j].op1) == 0 && strcmp(arr[i].op2, arr[j].op2) == 0) {
arr[j].flag = 1;
change(i, j, NULL);
}
}
}
}
}
}
void output() {
printf("\nOptimized code:\n");
for (int i = 0; i < n; i++) {
if (!arr[i].flag) {
printf("%s %s %s %s\n", arr[i].op, arr[i].op1, arr[i].op2, arr[i].res);
}
}
```

```
}
void change(int p, int q, char *res) {
for (int i = q + 1; i < n; i++) {
if (strcmp(arr[q].res, arr[i].op1) == 0) {
if (res == NULL) {
strcpy(arr[i].op1, arr[p].res);
} else {
strcpy(arr[i].op1, res);
}
} else if (strcmp(arr[q].res, arr[i].op2) == 0) {
if (res == NULL) {
strcpy(arr[i].op2, arr[p].res);
} else {
strcpy(arr[i].op2, res);
}
}
}
}
```

OUTPUT:-

```
Enter the number of expressions: 5
Enter the input expressions:
+  4  2  t1
+  a  t1  t2
-   B  a  t3
+  a  6  t4
*  t3 t2 t5
Optimized code:
+ a 6 t2
- B a t3
* t3 t2 t5
```

RESULT:

Thus a C program is written successfully for the Implementation of Global Data Flow Analysis

EX.NO.15 Implement any one storage allocation strategies (heap, stack, static)

AIM:

To implement heap storage allocation strategies using C program.

ALGORITHM:

1. Initialize head pointer to NULL.
2. Display a menu with options for creating, displaying, inserting, deleting elements, and quitting.
3. Repeat until the user chooses to quit
4. Initialize head pointer to NULL.
5. Create a loop:
6. Prompt the user to enter an element.
7. Create a new node with the entered element.
8. Insert the new node at the beginning of the list.
9. Prompt the user if they want to enter more elements.
10. Return the head pointer.
11. Traverse the list:
    Print the data of each node.
12. Prompt the user to enter an element.
13. Create a new node with the entered element.
14. Insert the new node at the beginning of the list.
15. Return the head pointer.
16. Prompt the user to enter the element to delete.
17. Search for the node with the entered element:
18. If found, delete the node:
    Update the links to skip the node.
    Free the memory of the deleted node.
    Print a message indicating successful deletion.
19. If not found, print a message indicating the element was not found.

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
#define TRUE 1
#define FALSE 0
typedef struct Heap
{
int data;
```

```
struct Heap *next;
} node;
node *create();
void display(node *);
node *insert(node *);
void dele(node **);
int main()
{
int choice;
node *head = NULL;
do
{
printf("\nProgram to perform various operations on heap using dynamic memory management");
printf("\n1. Create");
printf("\n2. Display");
printf("\n3. Insert an element in a list");
printf("\n4. Delete an element from list");
printf("\n5. Quit");
printf("\nEnter your choice (1-5): ");
scanf("%d", &choice);
switch (choice)
{
case 1:
head = create();
break;
case 2:
display(head);
break;
case 3:
head = insert(head);
break;
case 4:
dele(&head);
break;
case 5:
exit(0);
default:
printf("Invalid choice, try again.\n");
}
} while (choice != 5);
```

```
return 0;
}
node *create()
{
node *temp, *New, *head;
int val, flag;
char ans = 'y';
temp = NULL;
flag = TRUE;
do
{
printf("\nEnter the element: ");
scanf("%d", &val);
New = (node *)malloc(sizeof(node));
if (New == NULL)
{
printf("Memory is not allocated.");
exit(1);
}
New->data = val;
if (flag == TRUE)
{
head = New;
temp = head;
flag = FALSE;
}
else
{
temp->next = New;
temp = New;
}
printf("Do you want to enter more elements? (y/n): ");
scanf(" %c", &ans);
} while (ans == 'y');
printf("\nThe list is created\n");
return head;
}
void display(node *head)
{
node *temp;
```

```
temp = head;

if (temp == NULL)
{
printf("\nThe list is empty\n");
return;
}

printf("List: ");
while (temp != NULL)
{
printf("%d -> ", temp->data);
temp = temp->next;
}
printf("NULL\n");
}
node *insert(node *head)
{
node *New;
int val;
char ans;

New = (node *)malloc(sizeof(node));
if (New == NULL)
{
printf("Memory is not allocated.");
exit(1);
}
printf("Enter the element you want to insert: ");
scanf("%d", &val);
New->data = val;
New->next = NULL;
if (head == NULL)
{
head = New;
}
else
{
New->next = head;
head = New;
```

```
}
return head;
}
void dele(node **head)
{
node *temp, *prev;
int key;
temp = *head;
if (temp == NULL)
{
printf("\nThe list is empty\n");
return;
}
printf("\nEnter the element you want to delete: ");
scanf("%d", &key);
while (temp != NULL && temp->data != key)
{
prev = temp;
temp = temp->next;
}
if (temp == NULL)
{
printf("Element not found in the list\n");
return;
}
if (temp == *head)
{
*head = (*head)->next;
}
else
{
prev->next = temp->next;
}
free(temp);
printf("\nThe element is deleted\n");
}
```

OUTPUT:

Program to perform various operations on heap using dynamic memory management
1. Create
2. Display
3. Insert an element in a list
4. Delete an element from list
5. Quit
Enter your choice (1-5): 2

The list is empty

Program to perform various operations on heap using dynamic memory management
1. Create
2. Display
3. Insert an element in a list
4. Delete an element from list
5. Quit
Enter your choice (1-5): 5

RESULT:

Thus a C program is written successfully for the Implementation of heap storage allocation strategies.