****

**NCR CAMPUS, MODINAGAR**

# (A Constituent of SRM University, Chennai T.N.)

**Delhi-Meerut Road, Sikri Kalan, Modinagar – 201204, GHAZIABAD (U.P.)**

**Compiler Design Lab**

**(18CSC304J)**

**Lab Record**

**(Jan-May 2023)**

**Name of Student : Aditya Saxena**

**Reg. No: : RA2011003030223**

**Degree/ Branch : B-Tech / CSE**

**Section : 6thSem, Sec-K**

**Course Code : 18CSC304J**

**Course Title : Compiler Design**

**Faculty Incharge : Dr. Mayank Gupta**

**SRM IST, DELHI-NCR CAMPUS, MODINAGAR**

**Department Of Computer Science and Engineering**



# REGISTRATION NO

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **R** | **A** | **2** | **0** | **1** | **1** | **0** | **0** | **3** | **0** | **3** | **0** | **2** | **2** | **3** |

**BONAFIDE CERTIFICATE**

It is to be certified that the bonafide practical record submitted by *"****Aditya Saxena"*** of 6th semester for Bachelor of Technology degree in the Department of Computer Science and Engineering, Delhi-NCR Campus, SRM IST has been done for the course **Compiler Design** Lab (18CSC304J) during the academic semester session Jan 2023 – May 2023.

# Dr. R. P. Mahapatra Dr. Mayank Gupta

Head of the Department Assistant Professor Computer Science & Engg. Computer Science & Engg.

*Submitted for university examination held on \_\_/\_\_\_\_/\_\_\_ at SRM IST, NCR Campus.*

**Examiner 1 Examiner 2**

**INDEX**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Exp. No.** | **Title of Experiment** | **Page No.** | **Date of Experiment** | **Date of Completion**  **of Experiment** | **Teacher’s Signature** |
| **1** | Implementation of Lexical Analyzer | 1-3 |  |  |  |
| **2** | Regular Expression to NFA | 4-5 |  |  |  |
| **3** | Conversion from NFA to DFA | 6-7 |  |  |  |
| **4** | Elimination of Ambiguity, Left Recursion and Left Factoring | 8-11 |  |  |  |
| **5** | Computation of FIRST and FOLLOW | 12-19 |  |  |  |
| **6** | Computation of Predictive Parsing | 20-23 |  |  |  |
| **7** | Computation of Shift Reduce Parsing | 24-27 |  |  |  |
| **8** | Computation of LEADING and TRAILING | 28-34 |  |  |  |
| **9** | Computation of LR(0) item | 35-42 |  |  |  |
| **10** | Intermediate code generation – Postfix, Prefix | 43-46 |  |  |  |
| **11** | Intermediate code generation – Quadruple, Triple, Indirect triple | 47-52 |  |  |  |
| **12** | A simple code Generator |  |  |  |  |

**EXPERIMENT-1**

**Implementation of Lexical Analyzer**

**Aim:** Write a program in C/C++ to implement a lexical analyzer.

**Theory:**

Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences . In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.

Programs that perform Lexical Analysis in compiler design are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

**Program:**

#include<iostream>

#include<cstring>

#include<stdlib.h>

#include<ctype.h>

using namespace std;

string arr[] = { "if","else", "while", "break", "continue", "include", "iostream", "std", "main","cin", "cout", "return", "float", "double", "string","switch" ,"bool","union"}

bool isKeyword(string a) {

for (int i = 0; i < 14; i++) {

if (arr[i] == a) {

return true;

}

}

return false;

}

int main() {

printf(“Aditya Saxena\n”);

string input;

cout << "Enter the program code: ";

getline(cin, input);

string s;

for (int i = 0; i < input.length(); i++) {

char c = input[i];

if (c == ' ' || c == '\t' || c == '\n' || c == '\r') {

if (s != "") {

if (isKeyword(s)) {

cout << s << " is a keyword\n";

} else if (s == "+" || s == "-" || s == "" || s == "/" || s == "^" || s == "&&" || s == "||" || s == "=" || s == "==" || s == "&" || s == "|" || s == "%" || s == "++" || s == "--" || s == "+=" || s == "-=" || s == "/=" || s == "=" || s == "%=") {

cout << s << " is an operator\n";

} else if (s == "(" || s == "{" || s == "[" || s == ")" || s == "}" || s == "]" || s == "<" || s == ">" || s == "()" || s == ";" || s == "<<" || s == ">>" || s == "," || s == "#") {

cout << s << " is a symbol\n";

} else if (isdigit(s[0])) {

int x = 0;

if (!isdigit(s[x++])) {

continue;

} else {

cout << s << " is a constant\n";

}

} else {

cout << s << " is an identifier\n";

}

s = "";

}

} else {

s += c;

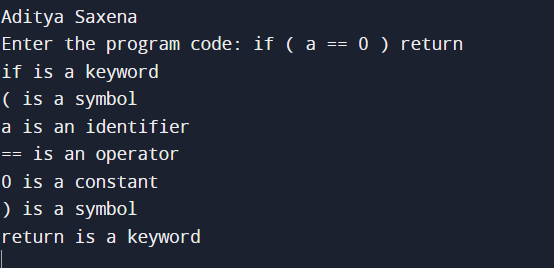
}

}

return 0;

}

**Output:-**

****

**Result:-** The Program Executed successfully.

**EXPERIMENT-2**

**Regular Expression to NFA**

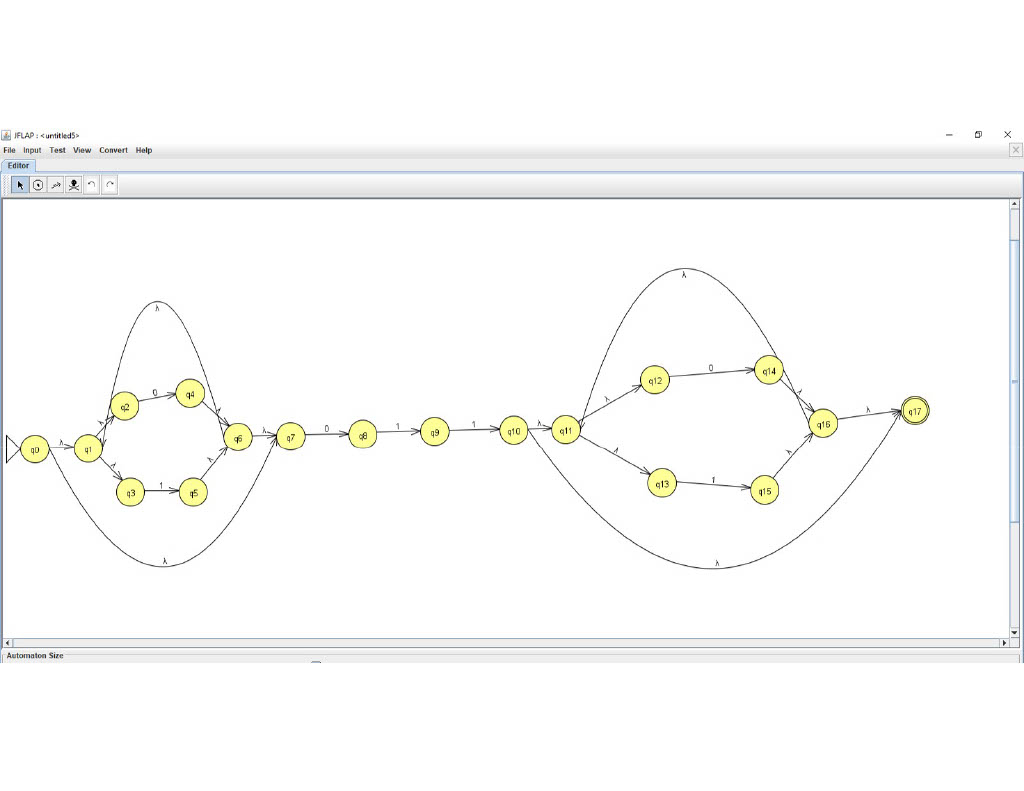
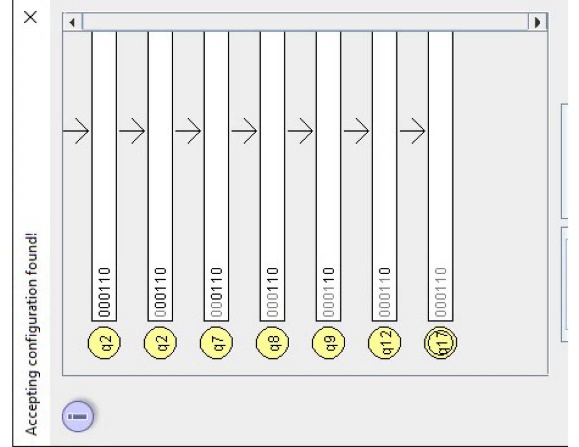
**Aim:** To convert the given Regular expression to NFA by using JFLAP.

**Theory:**

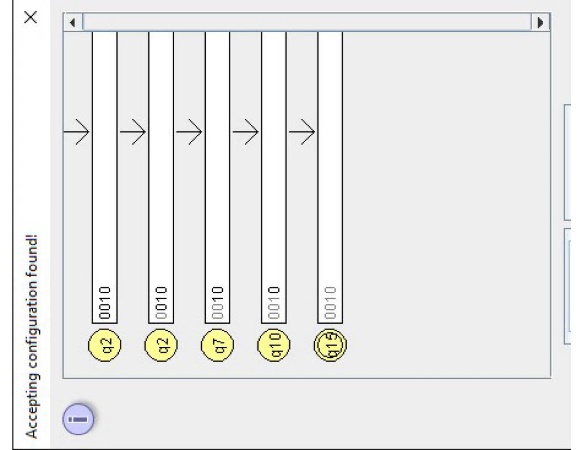
In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. Regular expressions are a concise way to represent a set of strings in formal languages and automata theory. They are a notation for describing regular languages, which can be recognized by finite state automata.

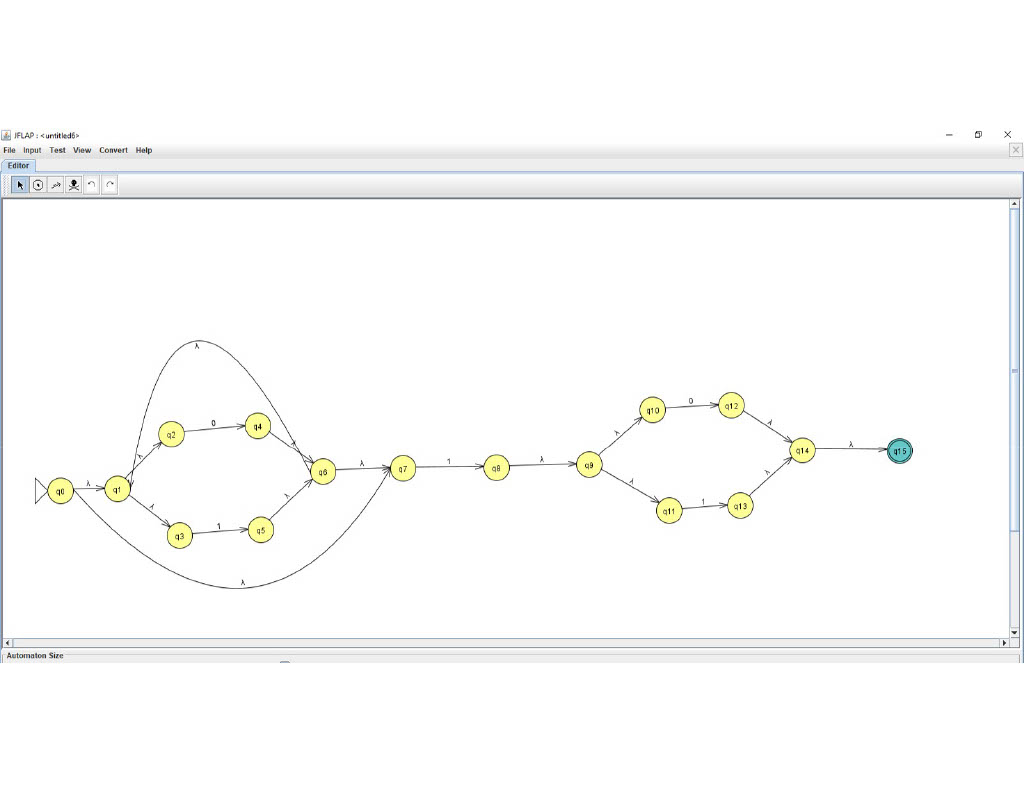
A regular expression is another representation of a regular language, and is defined over an alphabet (defined as Σ). The simplest regular expressions are symbols from λ, ∅, and symbols from Σ. Regular expressions can be built from these simple regular expressions with parenthesis, in addition to union, Kleene star and concatenation operators. In JFLAP, the concatenation symbol is implicit whenever two items are next to each other, and it is not explicitly stated.

1. **(0+1)\*011(0+1)\***

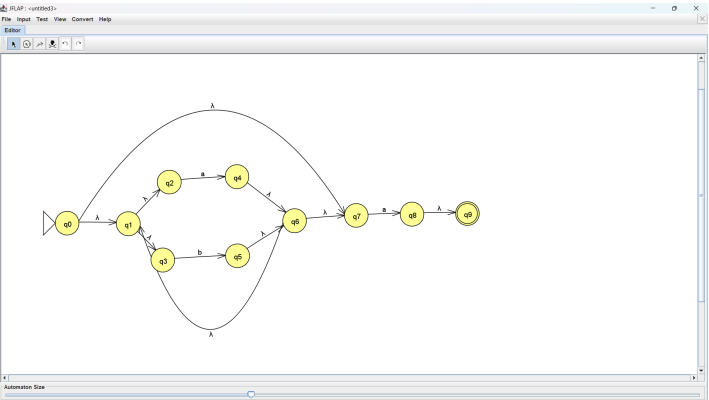
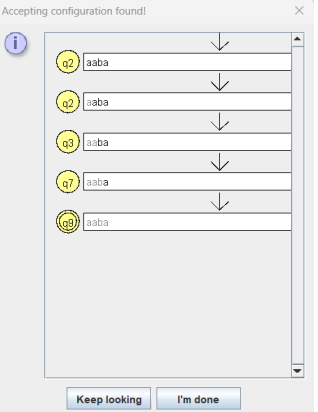


1. **(0+1)\*1(0+1)**





**3.(a+b)\*a**

****

**Result:** We converted the given Regular expression to NFA.

**EXPERIMENT-3**

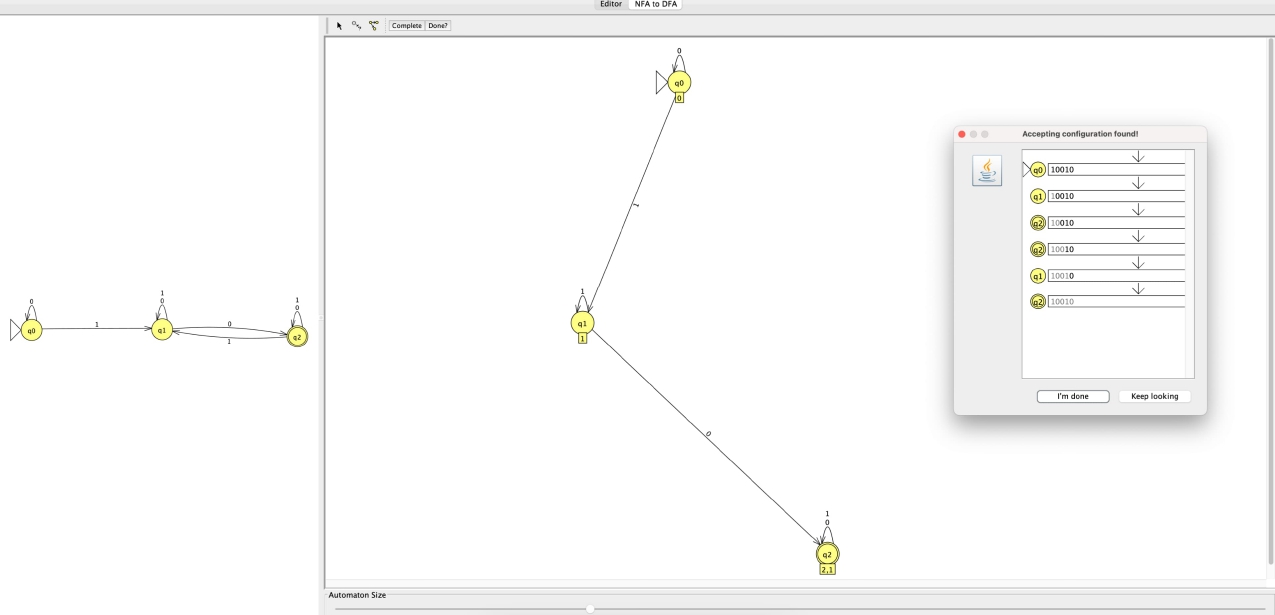
**NFA to DFA**

**Aim:** To convert the given NFA to DFA by using JFLAP.

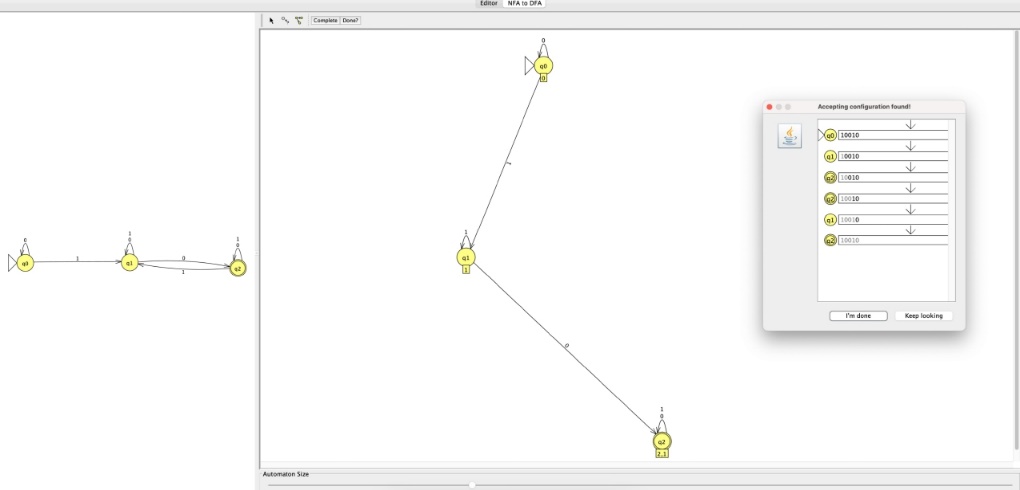
**Theory:**

An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.This operator may be applied to any nondeterministic FA. At the end of the operation, there will be a completed NFA. The conversion practice used is the standard canonical method of creating an equivalent DFA from an NFA, that is: each state in the DFA being built corresponds to a nonempty set of states in the original NFA. Therefore, for an NFA with n states, there are potentially 2n - 1 states in the DFA, though realistically this upper bound is rarely met.

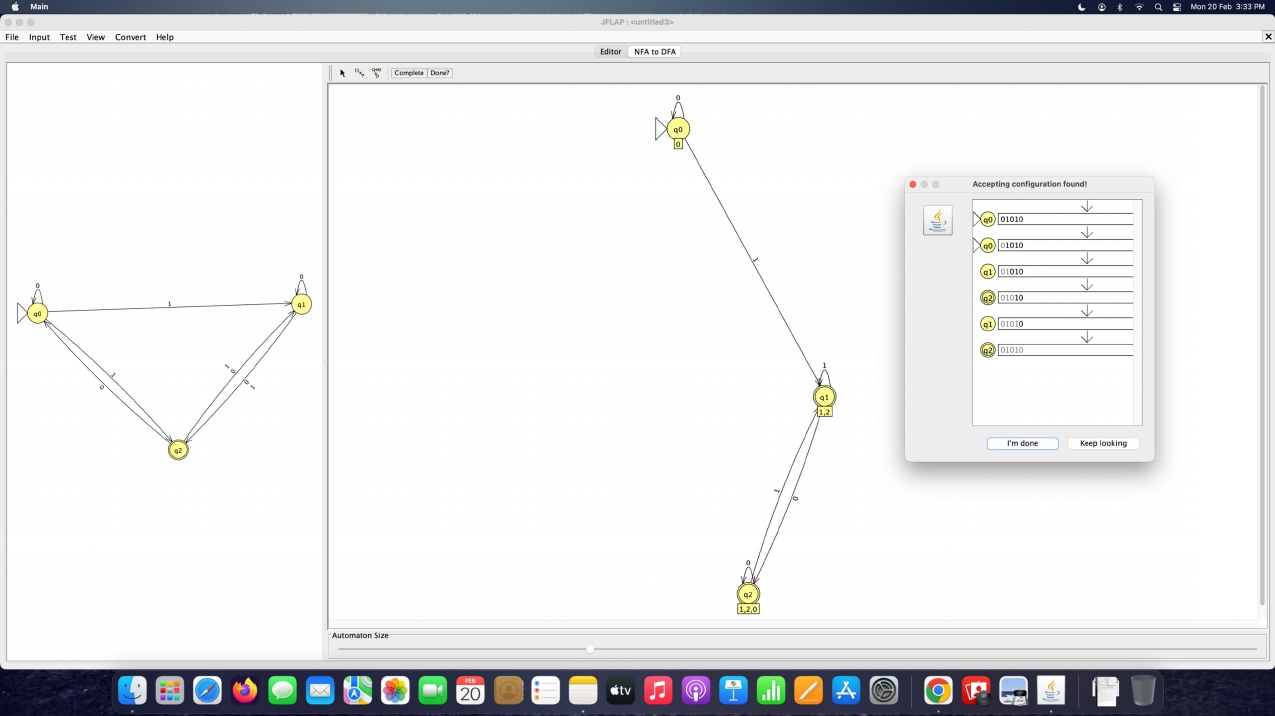
1. **Input NFA**



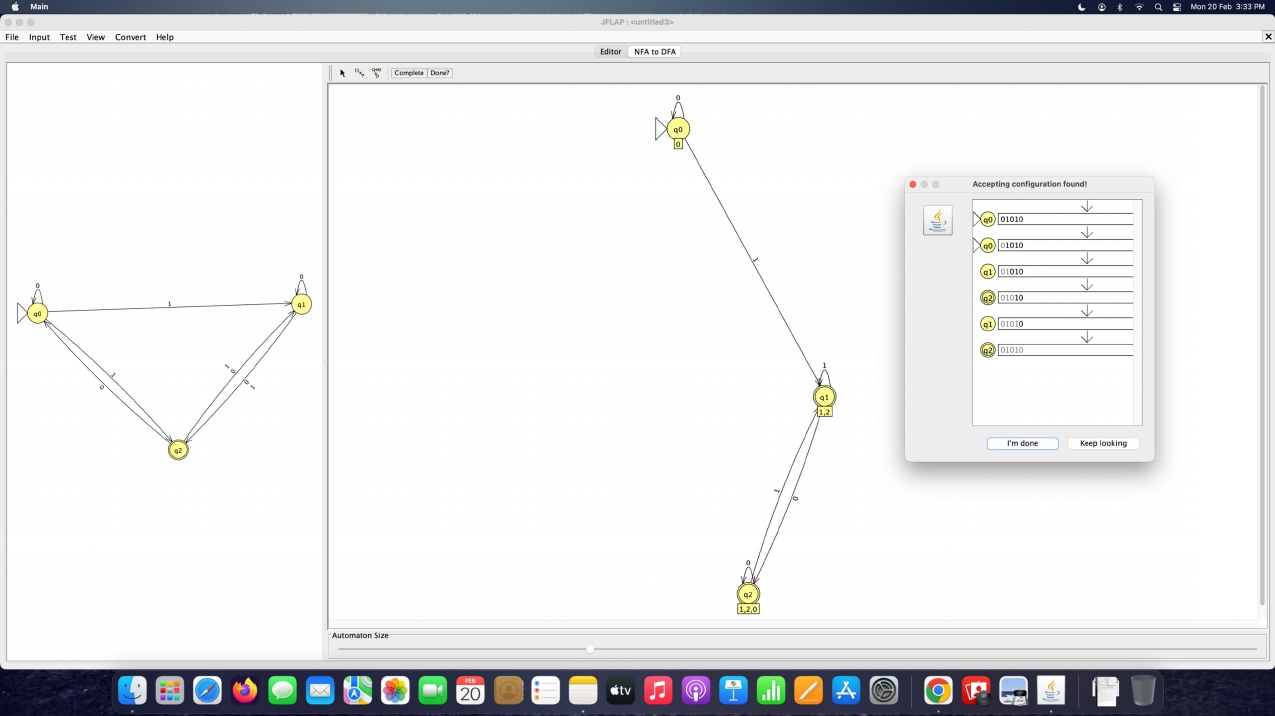
**Output DFA**



1. **Input NFA**



**Output DFA**



**Result:** We converted the given NFA to DFA.

**EXPERIMENT-4**

**Elimination of Ambiguity, Left Recursion and Left Factoring**

**Aim:** Write a program in C/C++ to Elimination of Ambiguity, Left Recursion and Left Factoring for a given set of production rule of a grammar.

**Theory:**

Left Recursion

* Left Recursion. The production is left-recursive if the leftmost symbol on the right side is the same as the non-terminal on the left side.
* For example, expr → expr + term. If one were to code this production in a recursive-descent parser, the parser would go in an infinite loop.

**Left Factoring**

* Left factoring is another useful grammar transformation used in parsing
* **Left Factoring** is a grammar transformation technique. It consists in "factoring out" prefixes which are common to two or more productions.

**Program**

**Left Recursion**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define SIZE 20

int main()

{

    printf("Aditya Saxena\n");

    char pro[SIZE], alpha[SIZE], beta[SIZE];

    int nont\_terminal,i,j, index=3;

    printf("Enter the Production as E->E|A: ");

    scanf("%s", pro);

    nont\_terminal=pro[0];

    if(nont\_terminal==pro[index])

    {

        for(i=++index,j=0;pro[i]!='|';i++,j++){

            alpha[j]=pro[i];

            if(pro[i+1]==0){

                printf("This Grammar CAN'T BE REDUCED.\n");

                exit(0);

            }

        }

        alpha[j]='\0';

        if(pro[++i]!=0)

        {

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

                beta[i]=pro[j];

            }

            beta[i]='\0';

            printf("\nGrammar Without Left Recursion: \n\n");

            printf(" %c->%s%c'\n", nont\_terminal,beta,nont\_terminal);

            printf(" %c'->%s%c'|#\n", nont\_terminal,alpha,nont\_terminal);

        }

        else

            printf("This Grammar CAN'T be REDUCED.\n");

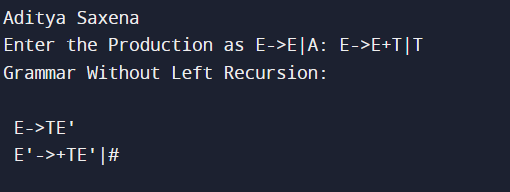
    }

    else

        printf("\n This Grammar is not LEFT RECURSIVE.\n");

}

**Output:-**

****

**Left Factoring**

#include<stdio.h>

#include<string.h>

int main()

{

    printf("Aditya Saxena");

    char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

    int i,j=0,k=0,l=0,pos;

    printf("Enter Production : A->");

    gets(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';

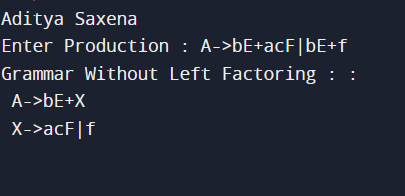
    printf("\nGrammar Without Left Factoring : : \n");

    printf(" A->%s",modifiedGram);

    printf("\n X->%s\n",newGram);

}

**Output:-**

****

**Result:** The Program Executed successfully.

**EXPERIMENT-5**

**Computation of FIRST and FOLLOW in a grammar**

**Aim:** Write a program in C/C++ to find a FIRST and FOLLOW set from a given set of production rule.

**Theory:-**

FIRST and FOLLOW are two functions associated with grammar that help us fill in the entries of an M-table.

**FIRST ()**− It is a function that gives the set of terminals that begin the strings derived from the production rule.

A symbol c is in FIRST (α) if and only if α ⇒ cβ for some sequence β of grammar symbols.

A terminal symbol a is in FOLLOW (N) if and only if there is a derivation from the start symbol S of the grammar such that S ⇒ αNαβ, where α and β are a (possible empty) sequence of grammar symbols. In other words, a terminal c is in FOLLOW (N) if c can follow N at some point in a derivation.

**Program:-**

#include <ctype.h>

#include <stdio.h>

#include <string.h>

void followfirst(char, int, int);

void follow(char c);

void findfirst(char, int, int);

int count, n = 0;

char calc\_first[10][100];

char calc\_follow[10][100];

int m = 0;

char production[10][10];

char f[10], first[10];

int k;

char ck;

int e;

int main(int argc, char\*\* argv)

{

printf("Aditya Saxena\n\n");

int jm = 0;

int km = 0;

int i, choice;

char c, ch;

count = 8;

strcpy(production[0], "X=TnS");

strcpy(production[1], "X=Rm");

strcpy(production[2], "T=q");

strcpy(production[3], "T=#");

strcpy(production[4], "S=p");

strcpy(production[5], "S=#");

strcpy(production[6], "R=om");

strcpy(production[7], "R=ST");

int kay;

char done[count];

int ptr = -1;

for (k = 0; k < count; k++) {

for (kay = 0; kay < 100; kay++) {

calc\_first[k][kay] = '!';

}

}

int point1 = 0, point2, xxx;

for (k = 0; k < count; k++) {

c = production[k][0];

point2 = 0;

xxx = 0;

for (kay = 0; kay <= ptr; kay++)

if (c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

findfirst(c, 0, 0);

ptr += 1;

done[ptr] = c;

printf("\n First(%c) = { ", c);

calc\_first[point1][point2++] = c;

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;

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;

xxx = 0;

for (kay = 0; kay <= ptr; kay++)

if (ck == donee[kay])

xxx = 1;

if (xxx == 1)

continue;

land += 1;

follow(ck);

ptr += 1;

donee[ptr] = ck;

printf(" Follow(%c) = { ", ck);

calc\_follow[point1][point2++] = ck;

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;

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') {

followfirst(production[i][j + 1], i,

(j + 2));

}

if (production[i][j + 1] == '\0'

&& c != production[i][0]) {

follow(production[i][0]);

}

}

}

}

}

void findfirst(char c, int q1, int q2)

{

int j;

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)) {

findfirst(production[q1][q2], q1,

(q2 + 1));

}

else

first[n++] = '#';

}

else if (!isupper(production[j][2])) {

first[n++] = production[j][2];

}

else {

findfirst(production[j][2], j, 3);

}

}

}

}

void followfirst(char c, int c1, int c2)

{

int k;

if (!(isupper(c)))

f[m++] = c;

else {

int i = 0, j = 1;

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

if (calc\_first[i][0] == c)

break;

}

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);

}

}

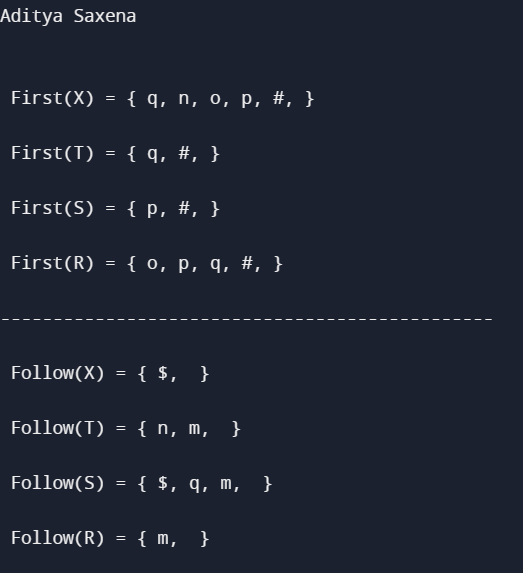
j++;

}

}

}

**Output:-**

****

**Result:** The Program Executed successfully.

**EXPERIMENT-6**

**Computation of Predictive Parsing**

**Aim:** Write a program in C/C++ for construction of predictive parser table.

**Theory:**

A predictive parser is a recursive descent parser with no backtracking or backup. It is a top-down parser that does not require backtracking. At each step, the choice of the rule to be expanded is made upon the next terminal symbol.  
Consider

A -> A1 | A2 | ... | An

If the non-terminal is to be further expanded to ‘A’, the rule is selected based on the current input symbol ‘a’ only.

**Program:-**

#include <stdio.h>

#include <string.h>

char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };

char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };

char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@" };

char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };

char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };

char table[5][6][10];

int numr(char c)

{

switch (c)

{

case 'S':

return 0;

case 'A':

return 1;

case 'B':

return 2;

case 'C':

return 3;

case 'a':

return 0;

case 'b':

return 1;

case 'c':

return 2;

case 'd':

return 3;

case '$':

return 4;

}

return (2);

}

int main(){

int i, j, k;

for (i = 0; i < 5; i++)

for (j = 0; j < 6; j++)

strcpy(table[i][j], " ");

printf("The following grammar is used for Parsing Table:\n");

for (i = 0; i < 7; i++)

printf("%s\n", prod[i]);

printf("\nPredictive parsing table:\n");

fflush(stdin);

for (i = 0; i < 7; i++){

k = strlen(first[i]);

for (j = 0; j < 10; j++)

if (first[i][j] != '@')

strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);

}

for (i = 0; i < 7; i++){

if (strlen(pror[i]) == 1){

if (pror[i][0] == '@'){

k = strlen(follow[i]);

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

strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);

}

}

}

strcpy(table[0][0], " ");

strcpy(table[0][1], "a");

strcpy(table[0][2], "b");

strcpy(table[0][3], "c");

strcpy(table[0][4], "d");

strcpy(table[0][5], "$");

strcpy(table[1][0], "S");

strcpy(table[2][0], "A");

strcpy(table[3][0], "B");

strcpy(table[4][0], "C");

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

for (i = 0; i < 5; i++)

for (j = 0; j < 6; j++)

{

printf("%-10s", table[i][j]);

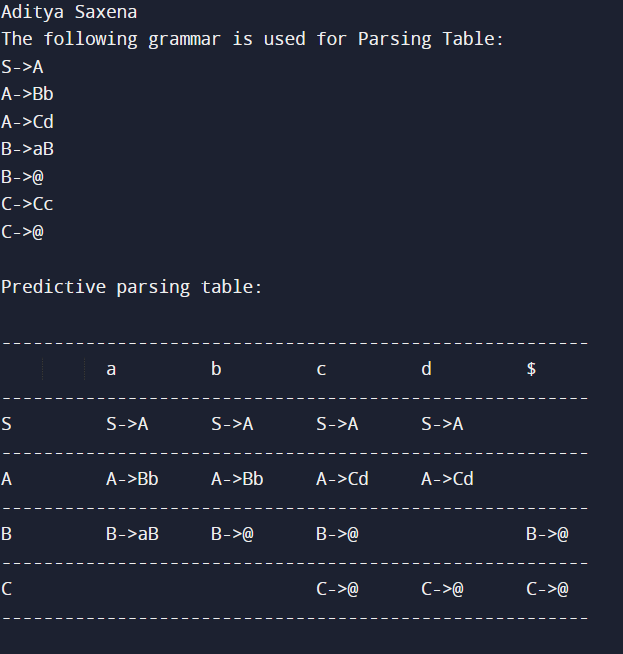
if (j == 5)

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

}

}

**Output:-**

****

**Result:** The Program Executed successfully.

**EXPERIMENT-7**

**Computation of Shift Reduce Parsing**

**Aim:** Write a program in C/C++ to implement the shift reduce parsing.

**Theory:-**

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of the shift-reduce parser is the LR parser.

This parser requires some data structures i.e.

* An input buffer for storing the input string.
* A stack for storing and accessing the production rules.

**Program:-**

#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("Aditya Saxena\n");

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();

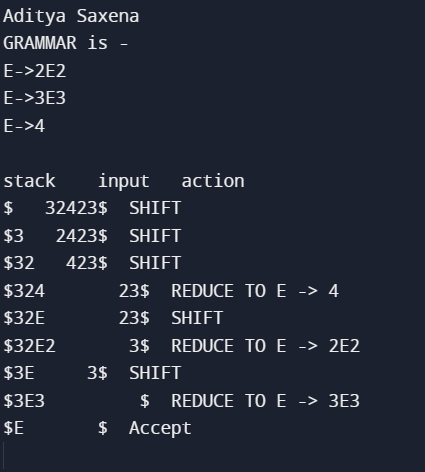
if(stk[0] == 'E' && stk[1] == '\0') printf("Accept\n");

else

printf("Reject\n");

}

**Output:-**

****

**Result:** The Program Executed successfully.

**EXPERIMENT-8**

**Computation of Leading and Trailing**

**Aim:** Write a program in C/C++ for construction of predictive parser table.

**Theory:**

**LEADING**

If production is of form A → aα or A → Ba α where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is

**Leading(A) = {a}**

If production is of form A → Bα, if a is in LEADING (B), then a will also be in LEADING (A).

**TRAILING**

If production is of form  A→ αa or A → αaB where B is Non-terminal, and α can be any string then,

**TRAILING (A) = {a}**

If production is of form  A → αB. If a is in TRAILING (B), then a will be in TRAILING (A).

**Program:-**

#include<bits/stdc++.h>

using namespace std;

#include <cstring>

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(){

cout<<"Aditya Saxena\n";

int i, s, k, j, n;

char pr[30][30], b, c;

cout<< "Enter the no of productions:";

cin>> n;

cout << "Enter the productions one by one\n";

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

cin >> 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 < t; j++)

tr[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++){

cout << "Leading[" << NT[i] << "]"

<< "\t{";

for (j = 0; j < t; j++){

if (l[i][j] == 't')

cout << T[j] << ",";

}

cout << "}\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++){

cout << "Trailing[" << NT[i] << "]"

<< "\t{";

for (j = 0; j < t; j++){

if (tr[i][j] == 't')

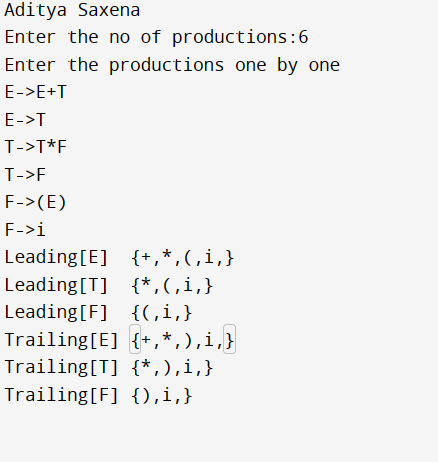
cout << T[j] << ","; }

cout << "}\n";

}

return 0;}

**Output:-**



**Result:** The Program Executed successfully.

**EXPERIMENT-9**

**Computation of LR(0) item**

**Aim:** Write a program in C/C++ for computation of LR(0) item.

**Theory:**

An LR (0) item is a production G with dot at some position on the right side of the production.LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.In the LR (0), we place the reduce node in the entire row.

Example:-

Given grammar:

S → AA

A → aA | b

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •S

S → •AA

A → •aA

A → •b

**Program:-**

#include<iostream>

#include<string.h>

using namespace std;

char prod[20][20],listofvar[26]="ABCDEFGHIJKLMNOPQR";

int novar=1,i=0,j=0,k=0,n=0,m=0,arr[30];

int noitem=0;

struct Grammar

{

    char lhs;

    char rhs[8];

}g[20],item[20],clos[20][10];

int isvariable(char variable)

{

    for(int i=0;i<novar;i++)

        if(g[i].lhs==variable)

            return i+1;

    return 0;

}

void findclosure(int z, char a)

{

    int n=0,i=0,j=0,k=0,l=0;

    for(i=0;i<arr[z];i++)

    {

        for(j=0;j<strlen(clos[z][i].rhs);j++)

        {

            if(clos[z][i].rhs[j]=='.' && clos[z][i].rhs[j+1]==a)

            {

                clos[noitem][n].lhs=clos[z][i].lhs;

                strcpy(clos[noitem][n].rhs,clos[z][i].rhs);

                char temp=clos[noitem][n].rhs[j];

                clos[noitem][n].rhs[j]=clos[noitem][n].rhs[j+1];

                clos[noitem][n].rhs[j+1]=temp;

                n=n+1;

            }

        }

    }

    for(i=0;i<n;i++)

    {

        for(j=0;j<strlen(clos[noitem][i].rhs);j++)

        {

            if(clos[noitem][i].rhs[j]=='.' && isvariable(clos[noitem][i].rhs[j+1])>0)

            {

                for(k=0;k<novar;k++)

                {

                    if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)

                    {

                        for(l=0;l<n;l++)

                            if(clos[noitem][l].lhs==clos[0][k].lhs && strcmp(clos[noitem][l].rhs,clos[0][k].rhs)==0)

                                break;

                        if(l==n)

                        {

                            clos[noitem][n].lhs=clos[0][k].lhs;

                        strcpy(clos[noitem][n].rhs,clos[0][k].rhs);

                            n=n+1;

                        }

                    }

                }

            }

        }

    }

    arr[noitem]=n;

    int flag=0;

    for(i=0;i<noitem;i++)

    {

        if(arr[i]==n)

        {

            for(j=0;j<arr[i];j++)

            {

                int c=0;

                for(k=0;k<arr[i];k++)

                    if(clos[noitem][k].lhs==clos[i][k].lhs && strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)

                        c=c+1;

                if(c==arr[i])

                {

                    flag=1;

                    goto exit;

                }

            }

        }

    }

    exit:;

    if(flag==0)

        arr[noitem++]=n;

}

int main()

{

    cout<<"Aditya Saxena\n";

    cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :\n";

    do

    {

        cin>>prod[i++];

    }while(strcmp(prod[i-1],"0")!=0);

    for(n=0;n<i-1;n++)

    {

        m=0;

        j=novar;

        g[novar++].lhs=prod[n][0];

        for(k=3;k<strlen(prod[n]);k++)

        {

            if(prod[n][k] != '|')

            g[j].rhs[m++]=prod[n][k];

            if(prod[n][k]=='|')

            {

                g[j].rhs[m]='\0';

                m=0;

                j=novar;

                g[novar++].lhs=prod[n][0];

            }

        }

    }

    for(i=0;i<26;i++)

        if(!isvariable(listofvar[i]))

            break;

    g[0].lhs=listofvar[i];

    char temp[2]={g[1].lhs,'\0'};

    strcat(g[0].rhs,temp);

    cout<<"\n\n augumented grammar \n";

    for(i=0;i<novar;i++)

        cout<<endl<<g[i].lhs<<"->"<<g[i].rhs<<" ";

    for(i=0;i<novar;i++)

    {

        clos[noitem][i].lhs=g[i].lhs;

        strcpy(clos[noitem][i].rhs,g[i].rhs);

        if(strcmp(clos[noitem][i].rhs,"ε")==0)

            strcpy(clos[noitem][i].rhs,".");

        else

        {

            for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--)

                clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];

            clos[noitem][i].rhs[0]='.';

        }

    }

    arr[noitem++]=novar;

    for(int z=0;z<noitem;z++)

    {

        char list[10];

        int l=0;

        for(j=0;j<arr[z];j++)

        {

            for(k=0;k<strlen(clos[z][j].rhs)-1;k++)

            {

                if(clos[z][j].rhs[k]=='.')

                {

                    for(m=0;m<l;m++)

                        if(list[m]==clos[z][j].rhs[k+1])

                            break;

                    if(m==l)

                        list[l++]=clos[z][j].rhs[k+1];

                }

            }

        }

        for(int x=0;x<l;x++)

            findclosure(z,list[x]);

    }

    cout<<"\n THE SET OF ITEMS ARE \n\n";

    for(int z=0; z<noitem; z++)

    {

        cout<<"\n I"<<z<<"\n\n";

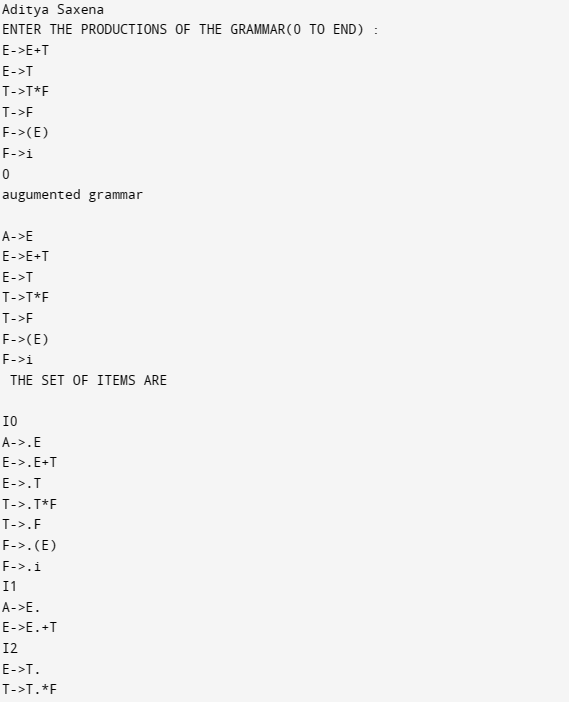
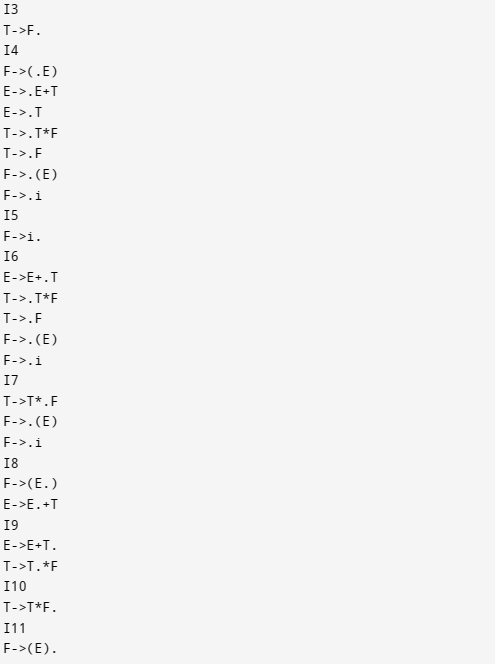
        for(j=0;j<arr[z];j++)

            cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";

    }

}

**Output:-**



**Result:** The Program Executed successfully.

**EXPERIMENT-10**

**Intermediate code generation – Postfix, Prefix**

**Aim:** Write a program in C/C++ for Intermediate code generation – Postfix, Prefix.

**Theory:**

Intermediate code generation is the process of transforming the source code of a programming language into an intermediate representation that is easier to analyse and optimize than the original source code. Intermediate code generation typically occurs after lexical analysis, parsing, and semantic analysis, and before code optimization and code generation.

Two common intermediate representations are postfix notation and prefix notation.

Postfix notation, also known as reverse Polish notation, represents an expression by placing the operators after the operands.

Prefix notation, also known as Polish notation, represents an expression by placing the operators before the operands.

**Program:-**

#include <iostream>

#include <stack>

#include <string>

#include <unordered\_set>

#include <unordered\_map>

using namespace std;

unordered\_set<char> operators = {'+', '-', '\*', '/', '(', ')'};

unordered\_map<char, int> PRI = {{'+', 1}, {'-', 1}, {'\*', 2}, {'/', 2}};

string infix\_to\_postfix(string formula) {

    stack<char> s;

    string output = "";

    for (char ch : formula) {

        if (operators.find(ch) == operators.end()) {

            output += ch;

        } else if (ch == '(') { s.push('('); }

else if (ch == ')') {

            while (!s.empty() && s.top() != '(') {

                output += s.top();

                s.pop();  }

            s.pop();  // pop '('

        } else {

            while (!s.empty() && s.top() != '(' && PRI[ch] <= PRI[s.top()]) {

                output += s.top();

                s.pop(); }

            s.push(ch); }

    }

    while (!s.empty()) {

        output += s.top();

        s.pop();  }

    cout << "POSTFIX: " << output << endl;

    return output;}

string infix\_to\_prefix(string formula) {

    stack<char> op\_stack;

    stack<string> exp\_stack;

    for (char ch : formula) {

        if (operators.find(ch) == operators.end()) {

            exp\_stack.push(string(1, ch));

        } else if (ch == '(') {

            op\_stack.push(ch);

        } else if (ch == ')') {

            while (op\_stack.top() != '(') {

                char op = op\_stack.top();

                op\_stack.pop();

                string a = exp\_stack.top();

                exp\_stack.pop();

                string b = exp\_stack.top();

                exp\_stack.pop();

                exp\_stack.push(op + b + a);

            }

            op\_stack.pop();

        } else {

            while (!op\_stack.empty() && op\_stack.top() != '(' && PRI[ch] <= PRI[op\_stack.top()]) {

                char op = op\_stack.top();

                op\_stack.pop();

                string a = exp\_stack.top();

                exp\_stack.pop();

                string b = exp\_stack.top();

                exp\_stack.pop();

                exp\_stack.push(op + b + a);

            }

            op\_stack.push(ch);

        }

    }

    while (!op\_stack.empty()) {

        char op = op\_stack.top();

        op\_stack.pop();

        string a = exp\_stack.top();

        exp\_stack.pop();

        string b = exp\_stack.top();

        exp\_stack.pop();

        exp\_stack.push(op + b + a);

    }

    cout << "PREFIX: " << exp\_stack.top() << endl;

    return exp\_stack.top();

}

int main() {

    cout<<"Aditya Saxena\n";

    string expression;

    cout << "INPUT THE EXPRESSION: ";

    getline(cin, expression);

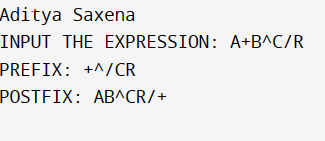
    string prefix = infix\_to\_prefix(expression);

    string postfix = infix\_to\_postfix(expression);

    return 0;

}

**Output:-**

****

**Result: -** The Program Executed successfully.

**EXPERIMENT-11**

**Intermediate code generation – Quadruple, Triple, Indirect triple**

**Aim:** Write a program in C/C++ for Intermediate code generation – Quadruple, Triple, Indirect triple.

**Theory:**

Intermediate code generation is the process of transforming the source code of a programming language into an intermediate representation that is easier to analyse and optimize than the original source code.

Most common forms of intermediate code are: - quadruple, triple, indirect triple.

A quadruple is a data structure that consists of four fields: an operator, and three operands. The operator represents an operation such as addition, subtraction, multiplication, or division, and the operands represent the inputs and output of the operation.

A triple is a data structure that is similar to a quadruple, but it has only three fields: an operator, and two operands. The output of the operation is also represented by a temporary variable.

An indirect triple is a data structure that represents an operation where the output is not explicitly stored in a temporary variable, but is instead stored indirectly through the use of other variables or registers.

**Program:-**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

struct Quadruple{

string op;

string arg1;

string arg2;

string result;

};

struct Triple{

string op;

string arg1;

string arg2;

};

vector<Triple> triples;

vector<Quadruple> quadruples;

int main() {

cout<<”Aditya Saxena”<<endl;

string expr;

cout << "Enter an arithmetic expression:";

getline(cin, expr);

vector<string> tokens;

string token;

for(char c : expr) {

if (c == ' ' || c == '\t')

continue;

if (isdigit(c))

token += c;

else {

if (!token.empty()) {

tokens.push\_back(token);

token.clear();

}

tokens.push\_back(string(1, c));

}

}

if (!token.empty()) {

tokens.push\_back(token);

}

vector<string> t1=tokens;

int temp\_num = 0;

string temp\_prefix = "t";

string temp;

for (size\_t i = 0; i < t1.size(); i++) {

string token = t1[i];

if (token == "\*" || token == "/") {

Quadruple q;

q.op = token;

q.arg1 = t1[i-1];

q.arg2 = t1[i+1];

q.result = temp\_prefix + to\_string(temp\_num);

temp\_num++;

quadruples.push\_back(q);

t1[i-1] = q.result;

t1.erase(t1.begin() + i, t1.begin() + i + 2);

i--;

}

}

for (size\_t i = 0; i < t1.size(); i++) {

string token = t1[i];

if (token == "+" || token == "-") {

Quadruple q;

q.op = token;

q.arg1 = t1[i-1];

q.arg2 = t1[i+1];

q.result = temp\_prefix + to\_string(temp\_num);

temp\_num++;

quadruples.push\_back(q);

t1[i-1] = q.result;

t1.erase(t1.begin() + i, t1.begin() + i + 2);

i--;

}

}

cout << "Quadruples:" << endl;

for (const Quadruple& q : quadruples) {

cout << q.op << " " << q.arg1 << " " << q.arg2 << " " << q.result << endl;

}

int temp\_num2 = 0;

string temp2;

for (size\_t i = 0; i < tokens.size(); i++) {

string token = tokens[i];

if (token == "\*" || token == "/") {

Triple t;

t.op = token;

t.arg1 = tokens[i-1];

t.arg2 = tokens[i+1];

triples.push\_back(t);

tokens[i-1] = to\_string(temp\_num2);

tokens.erase(tokens.begin() + i, tokens.begin() + i + 2);

i--;

temp\_num2++;

}

}

for (size\_t i = 0; i < tokens.size(); i++) {

string token = tokens[i];

if (token == "+" || token == "-") {

Triple t;

t.op = token;

t.arg1 = tokens[i-1];

t.arg2 = tokens[i+1];

triples.push\_back(t);

tokens[i-1] = to\_string(temp\_num2);

tokens.erase(tokens.begin() + i, tokens.begin() + i + 2);

i--;

temp\_num2++;

}

}

Triple t;

t.op = "=";

t.arg1 = tokens[0];

t.arg2 = "";

triples.push\_back(t);

cout << "Triples:\n";

for (Triple t : triples) {

cout << t.op << " " << t.arg1 << " " << t.arg2 << endl;

}

cout << "Indirect Triples:\n";

int id=100;

int ids=0;

for (Triple t : triples) {

cout<<id<<" "<<ids<<endl;

id+=1;

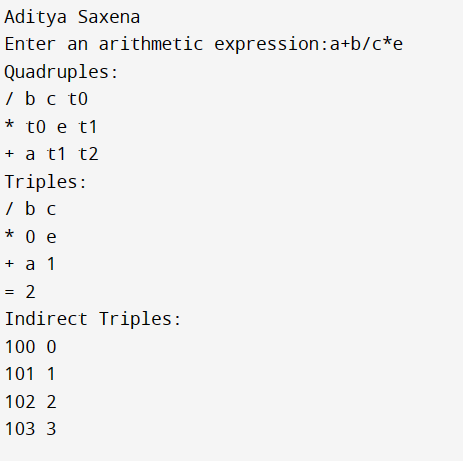
ids+=1;

}

return 0;

}

**Output:-**



**Result: -** The Program Executed successfully.