**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<ctype.h>

int isKeyword(char buffer[]) {

char keywords[32][10] = {

"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"

};

int i, flag = 0;

for(i = 0; i < 32; ++i) {

if(strcmp(keywords[i], buffer) == 0) {

flag = 1;

break;

}

}

return flag;

}

int main() {

char ch, buffer[15], operators[] = "+-\*/%=";

FILE \*fp;

int i,j=0;

fp = fopen("program.txt","r");

if(fp == NULL) {

printf("error while opening the file\n");

exit(0);

}

while((ch = fgetc(fp)) != EOF) {

for(i = 0; i < 6; ++i) {

if(ch == operators[i]) {

printf("%c is operator\n", ch);

}

}

if(isalnum(ch)) {

buffer[j++] = ch;

}

else if((ch == ' ' || ch == '\n') && (j != 0)) {

buffer[j] = '\0';

j = 0;

if(isKeyword(buffer) == 1) {

printf("%s is keyword\n", buffer);

}

else {

printf("%s is identifier\n", buffer);

}

}

}

fclose(fp);

return 0;

}

**PROGRAM:**

#include <stdio.h>

#include <conio.h>

void main() {

char m[20], t[10][10];

int n, i, j, r = 0, c = 0;

clrscr();

printf("\n\t\t\t\tSIMULATION OF NFA");

printf("\n\t\t\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

// Initialize the transition table

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

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

t[i][j] = ' ';

}

}

// Get the regular expression from the user

printf("\n\nEnter a regular expression: ");

scanf("%s", m);

n = strlen(m);

// Build the transition table

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

switch (m[i]) {

case '|': {

t[r][r + 1] = 'E';

t[r + 1][r + 2] = m[i - 1];

t[r + 2][r + 5] = 'E';

t[r][r + 3] = 'E';

t[r + 4][r + 5] = 'E';

t[r + 3][r + 4] = m[i + 1];

r = r + 5;

break;

}

case '\*': {

t[r - 1][r] = 'E';

t[r][r + 1] = 'E';

t[r][r + 3] = 'E';

t[r + 1][r + 2] = m[i - 1];

t[r + 2][r + 1] = 'E';

t[r + 2][r + 3] = 'E';

r = r + 3;

break;

}

case '+': {

t[r][r + 1] = m[i - 1];

t[r + 1][r] = 'E';

r = r + 1;

break;

}

default: {

if (c == 0) {

if ((isalpha(m[i])) && (isalpha(m[i + 1]))) {

t[r][r + 1] = m[i];

t[r + 1][r + 2] = m[i + 1];

r = r + 2;

c = 1;

}

c = 1;

} else if (c == 1) {

if (isalpha(m[i + 1])) {

t[r][r + 1] = m[i + 1];

r = r + 1;

c = 2;

}

} else {

if (isalpha(m[i + 1])) {

t[r][r + 1] = m[i + 1];

r = r + 1;

c = 3;

}

}

break;

}

}

}

// Print the transition table

printf("\n");

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

printf(" %d", j);

}

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("\n");

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

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

printf(" %c", t[i][j]);

}

printf(" | %d",i);

printf("\n");

}

printf("\nStart state: 0\nFinal state: %d",i-1);

getch();

}

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LEN 100

char NFA\_FILE[MAX\_LEN];

char buffer[MAX\_LEN];

int zz = 0;

// Structure to store DFA states and their

// status ( i.e new entry or already present)

struct DFA {

char \*states;

int count;

} dfa;

int last\_index = 0;

FILE \*fp;

int symbols;

/\* reset the hash map\*/

void reset(int ar[], int size) {

int i;

// reset all the values of

// the mapping array to zero

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

ar[i] = 0;

}

}

// Check which States are present in the e-closure

/\* map the states of NFA to a hash set\*/

void check(int ar[], char S[]) {

int i, j;

// To parse the individual states of NFA

int len = strlen(S);

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

// Set hash map for the position

// of the states which is found

j = ((int)(S[i]) - 65);

ar[j]++;

}

}

// To find new Closure States

void state(int ar[], int size, char S[]) {

int j, k = 0;

// Combine multiple states of NFA

// to create new states of DFA

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

if (ar[j] != 0)

S[k++] = (char)(65 + j);

}

// mark the end of the state

S[k] = '\0';

}

// To pick the next closure from closure set

int closure(int ar[], int size) {

int i;

// check new closure is present or not

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

if (ar[i] == 1)

return i;

}

return (100);

}

// Check new DFA states can be

// entered in DFA table or not

int indexing(struct DFA \*dfa) {

int i;

for (i = 0; i < last\_index; i++) {

if (dfa[i].count == 0)

return 1;

}

return -1;

}

/\* To Display epsilon closure\*/

void Display\_closure(int states, int closure\_ar[],

char \*closure\_table[],

char \*NFA\_TABLE[][symbols + 1],

char \*DFA\_TABLE[][symbols]) {

int i;

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

reset(closure\_ar, states);

closure\_ar[i] = 2;

// to neglect blank entry

if (strcmp(&NFA\_TABLE[i][symbols], "-") != 0) {

// copy the NFA transition state to buffer

strcpy(buffer, &NFA\_TABLE[i][symbols]);

check(closure\_ar, buffer);

int z = closure(closure\_ar, states);

// till closure get completely saturated

while (z != 100) {

if (strcmp(&NFA\_TABLE[z][symbols], "-") != 0) {

strcpy(buffer, &NFA\_TABLE[z][symbols]);

// call the check function

check(closure\_ar, buffer);

}

closure\_ar[z]++;

z = closure(closure\_ar, states);

}

}

// print the e closure for every states of NFA

printf("\n e-Closure (%c) :\t", (char)(65 + i));

bzero((void \*)buffer, MAX\_LEN);

state(closure\_ar, states, buffer);

strcpy(&closure\_table[i], buffer);

printf("%s\n", &closure\_table[i]);

}

}

/\* To check New States in DFA \*/

int new\_states(struct DFA \*dfa, char S[]) {

int i;

// To check the current state is already

// being used as a DFA state or not in

// DFA transition table

for (i = 0; i < last\_index; i++) {

if (strcmp(&dfa[i].states, S) == 0)

return 0;

}

// push the new

strcpy(&dfa[last\_index++].states, S);

// set the count for new states entered

// to zero

dfa[last\_index - 1].count = 0;

return 1;

}

// Transition function from NFA to DFA

// (generally union of closure operation )

void trans(char S[], int M, char \*clsr\_t[], int st,char \*NFT[][symbols + 1], char TB[]) {

int len = strlen(S);

int i, j, k, g;

int arr[st];

int sz;

reset(arr, st);

char temp[MAX\_LEN], temp2[MAX\_LEN];

char \*buff;

// Transition function from NFA to DFA

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

j = ((int)(S[i] - 65));

strcpy(temp, &NFT[j][M]);

if (strcmp(temp, "-") != 0) {

sz = strlen(temp);

g = 0;

while (g < sz) {

k = ((int)(temp[g] - 65));

strcpy(temp2, &clsr\_t[k]);

check(arr, temp2);

g++;

}

}

}

bzero((void \*)temp, MAX\_LEN);

state(arr, st, temp);

if (temp[0] != '\0') {

strcpy(TB, temp);

} else

strcpy(TB, "-");

}

/\* Display DFA transition state table\*/

void Display\_DFA(int last\_index, struct DFA \*dfa\_states,

char \*DFA\_TABLE[][symbols]) {

int i, j;

printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n");

printf("\n STATES OF DFA :\t\t");

for (i = 1; i < last\_index; i++) {

printf("%s, ", &dfa\_states[i].states);

printf("\n");

printf("\n GIVEN SYMBOLS FOR DFA: \t");

}

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

printf("%d, ", i);

printf("\n\n");

printf("STATES\t");

}

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

printf("|%d\t", i);

printf("\n");

}

}

// display the DFA transition state table

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

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

printf("%s\t", &dfa\_states[i + 1].states);

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

printf("|%s \t", &DFA\_TABLE[i][j]);

}

printf("\n");

}

}

// Driver Code

int main() {

int i, j, states;

char T\_buf[MAX\_LEN];

// creating an array dfa structures

struct DFA \*dfa\_states = malloc(MAX\_LEN \* (sizeof(dfa)));

states = 6, symbols = 2;

printf("\n STATES OF NFA :\t\t");

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

printf("%c, ", (char)(65 + i));

printf("\n");

printf("\n GIVEN SYMBOLS FOR NFA: \t");

}

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

printf("%d, ", i);

printf("eps");

printf("\n\n");

char \*NFA\_TABLE[states][symbols + 1];

// Hard coded input for NFA table

char \*DFA\_TABLE[MAX\_LEN][symbols];

strcpy(&NFA\_TABLE[0][0], "FC");

strcpy(&NFA\_TABLE[0][1], "-");

strcpy(&NFA\_TABLE[0][2], "BF");

strcpy(&NFA\_TABLE[1][0], "-");

strcpy(&NFA\_TABLE[1][1], "C");

strcpy(&NFA\_TABLE[1][2], "-");

strcpy(&NFA\_TABLE[2][0], "-");

strcpy(&NFA\_TABLE[2][1], "-");

strcpy(&NFA\_TABLE[2][2], "D");

strcpy(&NFA\_TABLE[3][0], "E");

strcpy(&NFA\_TABLE[3][1], "A");

strcpy(&NFA\_TABLE[3][2], "-");

strcpy(&NFA\_TABLE[4][0], "A");

strcpy(&NFA\_TABLE[4][1], "-");

strcpy(&NFA\_TABLE[4][2], "BF");

strcpy(&NFA\_TABLE[5][0], "-");

strcpy(&NFA\_TABLE[5][1], "-");

strcpy(&NFA\_TABLE[5][2], "-");

printf("\n NFA STATE TRANSITION TABLE \n\n\n");

printf("STATES\t"); }

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

printf("|%d\t", i);

printf("eps\n");

// Displaying the matrix of NFA transition table

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

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

printf("%c\t", (char)(65 + i));

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

printf("|%s \t", &NFA\_TABLE[i][j]);

}

printf("\n");

}

int closure\_ar[states];

char \*closure\_table[states];

Display\_closure(states, closure\_ar, closure\_table, NFA\_TABLE, DFA\_TABLE);

strcpy(&dfa\_states[last\_index++].states, "-");

dfa\_states[last\_index - 1].count = 1;

bzero((void \*)buffer, MAX\_LEN);

strcpy(buffer, &closure\_table[0]);

strcpy(&dfa\_states[last\_index++].states, buffer);

int Sm = 1, ind = 1;

int start\_index = 1;

// Filling up the DFA table with transition values

// Till new states can be entered in DFA table

while (ind != -1) {

dfa\_states[start\_index].count = 1;

Sm = 0;

}

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

trans(buffer, i, closure\_table, states, NFA\_TABLE, T\_buf);

// storing the new DFA state in buffer

strcpy(&DFA\_TABLE[zz][i], T\_buf);

// parameter to control new states

Sm = Sm + new\_states(dfa\_states, T\_buf);

}

ind = indexing(dfa\_states);

if (ind != -1) {

strcpy(buffer, &dfa\_states[++start\_index].states);

zz++; }

// display the DFA TABLE

Display\_DFA(last\_index, dfa\_states, DFA\_TABLE);

return 0;

}

**PROGRAM:**

**For Left Recursion:**

#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");

}

}

**For Left Factoring:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string ip, op1, op2, temp;

int sizes[10] = {};

char c;

int n, j, l;

cout << "Enter the Parent Non-Terminal: ";

cin >> c;

ip.push\_back(c);

op1 += ip + "\'->";

op2 += ip + "\'\'->";

ip += "->";

cout << "Enter the number of productions: ";

cin >> n;

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

cout << "Enter Production " << i + 1 << ": ";

cin >> temp;

sizes[i] = temp.size();

ip += temp;

if(i != n - 1)

ip += "|";

}

cout << "Production Rule: " << ip << endl;

char x = ip[3];

for(int i = 0, k = 3; i < n; i++) {

if(x == ip[k]) {

if(ip[k+1] == '|') {

op1 += "#";

ip.insert(k+1, 1, ip[0]);

ip.insert(k+2, 1, '\'');

k += 4;

} else {

op1 += "|" + ip.substr(k+1, sizes[i]-1);

ip.erase(k-1, sizes[i]+1);

}

} else {

while(ip[k++] != '|');

}

}

char y = op1[6];

for(int i = 0, k = 6; i < n-1; i++) {

if(y == op1[k]) {

if(op1[k+1] == '|') {

op2 += "#";

op1.insert(k+1, 1, op1[0]);

op1.insert(k+2, 2, '\'');

k += 5;

} else {

temp.clear();

for(int s = k + 1; s < op1.length(); s++)

temp.push\_back(op1[s]);

op2 += "|" + temp;

op1.erase(k-1, temp.length()+2);

}

}

}

op2.erase(op2.size()-1);

cout << "After Left Factoring: " << endl;

cout << ip << endl;

cout << op1 << endl;

cout << op2 << endl;

return 0;

}

**PROGRAM:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

// Functions to calculate Follow

void followfirst(char, int, int);

void follow(char c);

// Function to calculate First

void findfirst(char, int, int);

int count, n = 0;

// Stores the final result of the First Sets

char calc\_first[10][100];

// Stores the final result of the Follow Sets

char calc\_follow[10][100];

int m = 0;

// Stores the production rules

char production[10][10];

char f[10], first[10];

int k;

char ck;

int e;

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

int jm = 0;

int km = 0;

int i, choice;

char c, ch;

count = 8;

// The Input grammar

strcpy(production[0], "E=TR");

strcpy(production[1], "R=+TR");

strcpy(production[2], "R=#");

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

strcpy(production[4], "Y=\*FY");

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

strcpy(production[6], "F=(E)");

strcpy(production[7], "F=i");

int kay;

char done[count];

int ptr = -1;

// Initializing the calc\_first array

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

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

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

}

}

int point1 = 0, point2, xxx;

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

c = production[k][0];

point2 = 0;

xxx = 0;

// Checking if First of c has already been calculated

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

if (c == done[kay])

xxx = 1;

}

if (xxx == 1)

continue;

// Function call

findfirst(c, 0, 0);

ptr += 1;

// Adding c to the calculated list

done[ptr] = c;

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

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

// Printing the First Sets of the grammar

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

int lark = 0, chk = 0;

for (lark = 0; lark < point2; lark++) {

if (first[i] == calc\_first[point1][lark]) {

chk = 1;

break;

}

}

if (chk == 0) {

printf("%c, ", first[i]);

calc\_first[point1][point2++] = first[i];

}

}

printf("}\n");

jm = n;

point1++;

}

printf("\n");

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

char donee[count];

ptr = -1;

// Initializing the calc\_follow array

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

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

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

}

}

point1 = 0;

int land = 0;

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

ck = production[e][0];

point2 = 0;

xxx = 0;

// Checking if Follow of ck

// has alredy been calculated

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

if(ck == donee[kay]) {

xxx = 1; }

if (xxx == 1) {

continue; }

land += 1;

// Function call

follow(ck);

ptr += 1;

// Adding ck to the calculated list

donee[ptr] = ck;

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

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

// Printing the Follow Sets of the grammar

for(i = 0 + km; i < m; i++) {

int lark = 0, chk = 0;

for(lark = 0; lark < point2; lark++) {

if (f[i] == calc\_follow[point1][lark]) {

chk = 1;

break;

}

if(chk == 0) {

printf("%c, ", f[i]);

calc\_follow[point1][point2++] = f[i];

}

printf(" }\n\n");

km = m;

point1++;

}

}

void follow(char c) {

int i, j;

// Adding "$" to the follow

// set of the start symbol

if(production[0][0] == c) {

f[m++] = '$';

}

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

for(j = 2;j < 10; j++) {

if(production[i][j] == c) {

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

// Calculate the first of the next

// Non-Terminal in the production

followfirst(production[i][j+1], i, (j+2));

}

if(production[i][j+1]=='\0' && c!=production[i][0]) {

// Calculate the follow of the Non-Terminal in the L.H.S. of the production

follow(production[i][0]);

}

}

}

}

}

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

{

int j;

// The case where we

// encounter a Terminal

if(!(isupper(c))) {

first[n++] = c;

}

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

if(production[j][0] == c) {

if(production[j][2] == '#') {

if(production[q1][q2] == '\0') {

first[n++] = '#';

else if(production[q1][q2] != '\0'

&& (q1 != 0 || q2 != 0)) {

findfirst(production[q1][q2], q1, (q2+1));

}

else {

first[n++] = '#';

}

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

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

}

else {

// Recursion to calculate First of

// New Non-Terminal we encounter at the beginning

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

}

}

}

}

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

int k;

// The case where we encounter

// a Terminal

if(!(isupper(c))) {

f[m++] = c;

else {

int i = 0, j = 1;

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

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

break;

}

//Including the First set of the Non-Terminal in the Follow of the original query

while(calc\_first[i][j] != '!') {

if(calc\_first[i][j] != '#') {

f[m++] = calc\_first[i][j];

}

else {

if(production[c1][c2] == '\0') {

// Case where we reach the end of a production

follow(production[c1][0]);

}

else {

// Recursion to the next symbol

// in case we encounter a "#"

followfirst(production[c1][c2], c1, c2+1); }

}

j++;

}

}

}