**EXPERIMENT - 4**

**AIM:** Write a program for finding first and follow.

**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.

**FOLLOW () -**It isthe set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

**Benefit of FIRST () and FOLLOW ()**

* It can be used to prove the LL (K) characteristic of grammar.
* It can be used to promote in the construction of predictive parsing tables.
* It provides selection information for recursive descent parsers.

**CODE:**

#include<bits/stdc++.h>

using namespace std;

set<char> ss;

bool dfs(char i, char org, char last, map<char,vector<vector<char>>> &mp){

bool rtake = false;

for(auto r : mp[i]){

bool take = true;

for(auto s : r){

if(s == i) break;

if(!take) break;

if(!(s>='A'&&s<='Z')&&s!='e'){

ss.insert(s);

break;

}

else if(s == 'e'){

if(org == i||i == last)

ss.insert(s);

rtake = true;

break;

}

else{

take = dfs(s,org,r[r.size()-1],mp);

rtake |= take;

}

}

}

return rtake;

}

int main(){

int i,j;

ifstream fin("inputfirstfollow.txt");

string num;

vector<int> fs;

vector<vector<int>> a;

map<char,vector<vector<char>>> mp;

char start;

bool flag = 0;

cout<<"Grammar: "<<'\n';

while(getline(fin,num)){

if(flag == 0) start = num[0],flag = 1;

cout<<num<<'\n';

vector<char> temp;

char s = num[0];

for(i=3;i<num.size();i++){

if(num[i] == '|'){

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

temp.clear();

}

else temp.push\_back(num[i]);

}

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

}

map<char,set<char>> fmp;

for(auto q : mp){

ss.clear();

dfs(q.first,q.first,q.first,mp);

for(auto g : ss) fmp[q.first].insert(g);

}

cout<<"================FIRST==============="<<endl;

for(auto cur:fmp)

{

cout<<cur.first<<"->";

for(auto val:cur.second)

{

cout<<val<<" ";

}

cout<<endl;

}

cout<<'\n';

cout<<"FIRST: "<<'\n';

for(auto q : fmp){

string ans = "";

ans += q.first;

ans += " = {";

for(char r : q.second){

ans += r;

ans += ',';

}

ans.pop\_back();

ans+="}";

cout<<ans<<'\n';

}

map<char,set<char>> gmp;

gmp[start].insert('$');

int count = 10;

while(count--){

for(auto q : mp){

for(auto r : q.second){

for(i=0;i<r.size()-1;i++){

if(r[i]>='A'&&r[i]<='Z'){

if(!(r[i+1]>='A'&&r[i+1]<='Z')) gmp[r[i]].insert(r[i+1]);

else {

char temp = r[i+1];

int j = i+1;

while(temp>='A'&&temp<='Z'){

if(\*fmp[temp].begin()=='e'){

for(auto g : fmp[temp]){

if(g=='e') continue;

gmp[r[i]].insert(g);

}

j++;

if(j<r.size()){

temp = r[j];

if(!(temp>='A'&&temp<='Z')){

gmp[r[i]].insert(temp);

break;

}

}

else{

for(auto g : gmp[q.first]) gmp[r[i]].insert(g);

break;

}

}

else{

for(auto g : fmp[temp]){

gmp[r[i]].insert(g);

}

break;

}

}

}

}

}

if(r[r.size()-1]>='A'&&r[r.size()-1]<='Z'){

for(auto g : gmp[q.first]) gmp[r[i]].insert(g);

}

}

}

}

cout<<'\n';

cout<<"FOLLOW: "<<'\n';

for(auto q : gmp){

string ans = "";

ans += q.first;

ans += " = {";

for(char r : q.second){

ans += r;

ans += ',';

}

ans.pop\_back();

ans+="}";

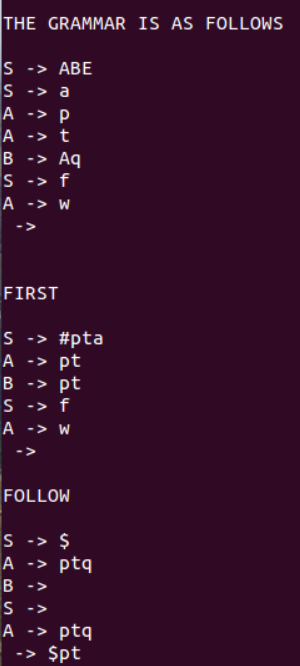
cout<<ans<<'\n';

}

return 0;

}

**OUTPUT:**



**EXPERIMENT - 5**

**AIM:** Write a program to implement shift reducer parser.

**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.

**Basic Operations –**

* **Shift:** This involves moving symbols from the input buffer onto the stack.
* **Reduce:** If the handle appears on top of the stack, then its reduction by using appropriate production rule is done i.e., RHS of a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.
* **Accept:** If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it is means successful parsing is done.
* **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

**CODE:**

#include <bits/stdc++.h>

using namespace std;

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

printf("Reject\n");

}

**OUTPUT:**

