**Practical no. 1**

**Objective** – Design Lexical Analyzer and Simulate the code in C/C++ language

**Introduction -** A lexical analyzer, also known as a lexer or tokenizer, is the first phase of a compiler. It takes a sequence of characters (source code) as input and produces a sequence of tokens.

A token is a logical component of the source code, such as a keyword, identifier, constant, string literal, or symbol (like ‘+’, ‘-’, ‘\*’, ‘/’, etc.).

The lexical analyzer reads the source code character by character, grouping them into tokens, and categorizing each token based on its type. This process is known as tokenization.

For example, consider the following line of C++ code:

int sum = a + b;

A lexical analyzer would break this down into the following tokens:

* int (keyword)
* sum (identifier)
* = (operator)
* a (identifier)
* + (operator)
* b (identifier)
* ; (symbol)

The lexical analyzer also removes white spaces and comments from the source code, as they are not needed in later stages of the compiler.

The output of the lexical analyzer (the tokens) is used by the syntax analyzer (or parser), which is the next phase of the compiler. The parser uses these tokens to create a parse tree, which represents the syntactic structure of the program according to the language’s grammar rules.

In summary, the lexical analyzer simplifies the job of the parser by breaking the source code down into manageable tokens. This is a crucial step in the process of compiling source code into machine code.Example of tokens:

* Type token (id, number, real, . . . )
* Punctuation tokens (IF, void, return, . . . )
* Alphabetic tokens (keywords)

Keywords; Examples-for, while, if etc.

Identifier; Examples-Variable name, function name, etc.

Operators; Examples '+', '++', '-' etc.

Separators; Examples ',' ';' etc

**Resource** – C++

**Program logic -** The provided code is a simple lexical analyzer in C++. Here’s the logic of the program:

**isKeyword Function:** This function checks if a given string is a keyword in C++. It has a list of all 32 keywords in C++ and it compares the input string with each keyword. If a match is found, it returns 1, otherwise it returns 0.

**Main Function:** The main function reads a file named “lexicalinput.txt” and analyzes its content. It categorizes the characters in the file into different groups: keywords, identifiers, mathematical operators, logical operators, numerical values, and others.

1. **Keywords**: Words that match any of the 32 C++ keywords.
2. **Identifiers:** Alphanumeric characters that are not keywords.
3. **Mathematical Operators:** Characters that are any of ‘+’, ‘-’, ‘\*’, ‘/’, ‘=’.
4. **Logical Operators:** Characters that are either ‘>’ or ‘<’.
5. **Numerical Values:** Sequences of digits possibly containing a decimal point.
6. **Others:** Any other characters like ‘,’, ‘;’, ‘(’, ‘)’, ‘{’, ‘}’, ‘[’, ‘]’, ‘'’, ‘:’, etc.

The program uses vectors to store the characters of each category. It also uses an array named mark to ensure that each character is only stored once in its respective category.

Finally, the program prints out the characters in each category.

Please note that this program assumes that the input file “lexicalinput.txt” exists in the same directory and can be opened. If the file cannot be opened, the program will print an error message and exit. Also, the program does not handle errors in the input file’s content. For example, it does not check if an identifier starts with a letter or if a number is well-formed. It simply categorizes the characters as they appear in the file.

**Program –**

**lexicalinput.txt –**

int a, b, c;

float d, e;

a = b = 5;

c = 6;

if (a > b) {

c = a - b;

e = d - 2.0;

} else {

d = e + 6.0;

b = a + c;

}

**lexicalanalyzer.cpp –**

#include<bits/stdc++.h>

#include<stdlib.h>

#include<string.h>

#include<ctype.h>

using namespace std;

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],b[30], logical\_op[] = "><",math\_op[]="+-\*/=",numer[]=".0123456789",other[]=",;(){}[]'':";

ifstream fin("lexicalinput.txt");

int mark[1000]={0};

int i,j=0,kc=0,ic=0,lc=0,mc=0,nc=0,oc=0,aaa=0;

vector < string > k;

vector<char >id;

vector<char>lo;

vector<char>ma;

vector<string>nu;

vector<char>ot;

if(!fin.is\_open()){

cout<<"error while opening the file\n";

exit(0);

}

while(!fin.eof()){

ch = fin.get();

for(i = 0; i < 12; ++i){

if(ch == other[i]){

int aa=ch;

if(mark[aa]!=1){

ot.push\_back(ch);

mark[aa]=1;

++oc;

}

}

}

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

if(ch == math\_op[i]){

int aa=ch;

if(mark[aa]!=1){

ma.push\_back(ch);

mark[aa]=1;

++mc;

}

}

}

for(i = 0; i < 2; ++i){

if(ch == logical\_op[i]){

int aa=ch;

if(mark[aa]!=1){

lo.push\_back(ch);

mark[aa]=1;

++lc;

}

}

}

if(ch=='0' || ch=='1' || ch=='2' || ch=='3' || ch=='4' ||

ch=='5' || ch=='6' || ch=='7' || ch=='8' || ch=='9' || ch=='.' ||ch ==

' ' || ch == '\n' || ch == ';'){

if(ch=='0' || ch=='1' || ch=='2' || ch=='3' || ch=='4' ||

ch=='5' || ch=='6' || ch=='7' || ch=='8' || ch=='9' ||

ch=='.')b[aaa++]=ch;

if((ch == ' ' || ch == '\n' || ch == ';') && (aaa != 0)){

b[aaa] = '\0';

aaa = 0;

char arr[30];

strcpy(arr,b);

nu.push\_back(arr);

++nc;

}

}

if(isalnum(ch)){

buffer[j++] = ch;

}

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

buffer[j] = '\0';

j = 0;

if(isKeyword(buffer) == 1){

k.push\_back(buffer);

++kc;

}

else{

if(buffer[0]>=97 && buffer[0]<=122) {

if(mark[buffer[0]-'a']!=1){

id.push\_back(buffer[0]);

++ic;

mark[buffer[0]-'a']=1;

}

}

}

}

}

fin.close();

printf("Keywords: ");

for(int f=0;f<kc;++f){

if(f==kc-1){

cout<<k[f]<<"\n";

}

else {

cout<<k[f]<<", ";

}

}

printf("Identifiers: ");

for(int f=0;f<ic;++f){

if(f==ic-1){

cout<<id[f]<<"\n";

}

else {

cout<<id[f]<<", ";

}

}

printf("Math Operators: ");

for(int f=0;f<mc;++f){

if(f==mc-1){

cout<<ma[f]<<"\n";

}

else {

cout<<ma[f]<<", ";

}

}

printf("Logical Operators: ");

for(int f=0;f<lc;++f){

if(f==lc-1){

cout<<lo[f]<<"\n";

}

else {

cout<<lo[f]<<", ";

}

}

printf("Numerical Values: ");

for(int f=0;f<nc;++f){

if(f==nc-1){

cout<<nu[f]<<"\n";

}

else {

cout<<nu[f]<<", ";

}

}

printf("Others: ");

for(int f=0;f<oc;++f){

if(f==oc-1){

cout<<ot[f]<<"\n";

}

else {

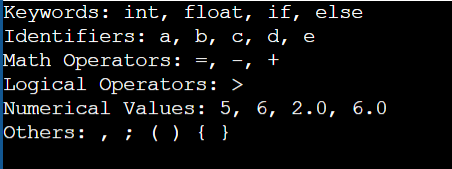
cout<<ot[f]<<" ";

}

}

return 0;

}

**Output –**

**Practical no. 2**

**Objective –** Write a Program to find First of the Grammar

**Introduction –** FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X.

FIRST set is a concept used in syntax analysis, specifically in the context of LL and LR parsing algorithms. It is a set of terminals that can appear immediately after a given non-terminal in a grammar.

The FIRST set of a non-terminal A is defined as the set of terminals that can appear as the first symbol in any string derived from A. If a non-terminal A can derive the empty string, then the empty string is also included in the FIRST set of A.

**Theory -** Rules to compute FIRST set:

1. If x is a terminal, then FIRST(x) = { ‘x’ }
2. If x-> ?, is a production rule, then add ? to FIRST(x).
3. If X->Y1 Y2 Y3….Yn is a production,
4. FIRST(X) = FIRST(Y1)
5. If FIRST(Y1) contains ? then FIRST(X) = { FIRST(Y1) – ? } U { FIRST(Y2) }

If FIRST (Yi) contains ? for all i = 1 to n, then add ? to FIRST(X).

**Resource –** C

**Program logic –** Here’s the breakdown of the logic:

1. **Data Structures:**
   * prodn[10][10]: A 2D array to store the productions of the grammar.
   * firstTerms[10]: An array to store the FIRST set elements.
   * count: To keep track of the number of productions.
   * n: A counter for the number of elements in the FIRST set.
2. **Input:**
   * The program first asks for the number of productions and then for the productions themselves, where epsilon is represented by $.
3. **Calculation of FIRST Set:**
   * The FIRSTfunc function is called with a symbol (non-terminal or terminal) as an argument.
   * If the symbol is a non-terminal (uppercase letter), the function looks through the productions to find those that start with this non-terminal.
   * If the production is like A -> $ (where $ represents epsilon), epsilon is added to the FIRST set.
   * If the production is like A -> a (where a is a terminal), a is added to the FIRST set.
   * If the production is like A -> B (where B is a non-terminal), the function is called recursively with B.
4. **Output:**
   * After calculating the FIRST set for a symbol, it prints out the elements of the FIRST set.
5. **Looping Mechanism:**
   * The program allows for repeated calculation of FIRST sets for different symbols until the user decides to stop by not pressing 1.

**Program –**

#include <stdio.h>

#include <ctype.h>

void FIRSTfunc(char);

int count, n = 0;

char prodn[10][10], firstTerms[10];

main()

{

int i, choice;

char c, ch;

printf("How many productions ? :");

scanf("%d", &count);

printf("Enter %d productions epsilon= $ :\n\n", count);

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

scanf("%s%c", prodn[i], &ch);

do

{

n = 0;

printf("Element :");

scanf("%c", &c);

FIRSTfunc(c);

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

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

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

printf("}\n");

printf("press 1 to continue : ");

scanf("%d%c", &choice, &ch);

} while (choice == 1);

}

void FIRSTfunc(char c)

{

int j;

if (!(isupper(c)))

firstTerms[n++] = c;

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

{

if (prodn[j][0] == c)

{

if (prodn[j][2] == '$')

firstTerms[n++] = '$';

else if (islower(prodn[j][2])) // A--> a or A-->B

firstTerms[n++] = prodn[j][2];

else

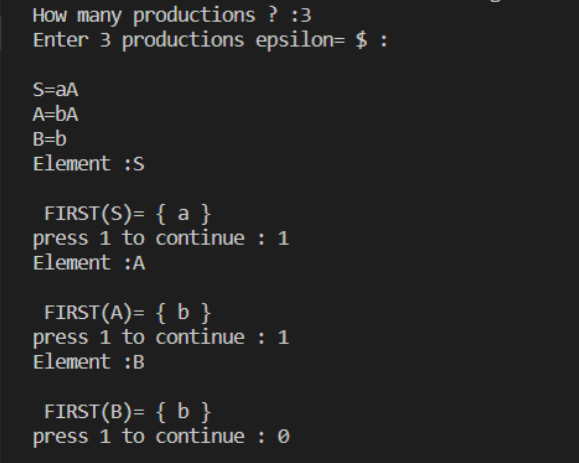
FIRSTfunc(prodn[j][2]);

}

}

}

**Output -**



**Practical no. 3**

**Objective -** Write a Program to compute Follow of the Grammar

**Introduction –** FOLLOW set is used to identify the terminal symbol immediately after a non-terminal in a given language. FOLLOW set is also used to avoid backtracking same as the FIRST set. The only difference is FOLLOW set works on vanishing non-terminal on the right-hand side so that decision-making gets easier for the compiler while parsing.

Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

**Theory –** Store the grammar on a 2D character array production. findfirst function is for calculating the first of any non terminal. Calculation of first falls under two broad cases :

* If the first symbol in the R.H.S of the production is a Terminal then it can directly be included in the first set.
* If the first symbol in the R.H.S of the production is a Non-Terminal then call the findfirst function again on that Non-Terminal. To handle these cases like Recursion is the best possible solution. Here again, if the First of the new Non-Terminal contains an epsilon then we have to move to the next symbol of the original production which can again be a Terminal or a Non-Terminal.

Rules to compute **-**

* FOLLOW(S) = { $ } // where S is the starting Non-Terminal
* If A -> pBq is a production, where p, B and q are any grammar symbols,

then everything in FIRST(q) except Є is in FOLLOW(B).

* If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
* If A->pBq is a production and FIRST(q) contains Є,

then FOLLOW(B) contains { FIRST(q) – Є } U FOLLOW(A)

**Resource –** C

**Program logic –**

1. The main function initializes variables and arrays required for the algorithm.
2. The findfirst function computes the FIRST set for each non-terminal symbol in the given grammar.
3. The follow function computes the FOLLOW set for each non-terminal symbol in the given grammar.
4. The followfirst function calculates the FOLLOW set for a given non-terminal symbol by considering its FIRST set.
5. The program then prints out the FIRST and FOLLOW sets for each non-terminal symbol in the grammar.

Here's a high-level overview of the program logic:

* The program starts by initializing variables and arrays.
* It then computes the FIRST set for each non-terminal symbol using the findfirst function.
* After that, it computes the FOLLOW set for each non-terminal symbol using the follow function.
* The followfirst function is called within the follow function to handle cases where the production rule ends with epsilon (#).
* Finally, the program prints out the computed FIRST and FOLLOW sets for each non-terminal symbol.

**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)

{

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;

// 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;

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;

// 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;

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') {

// 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;

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;

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') {

// 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);

}

}

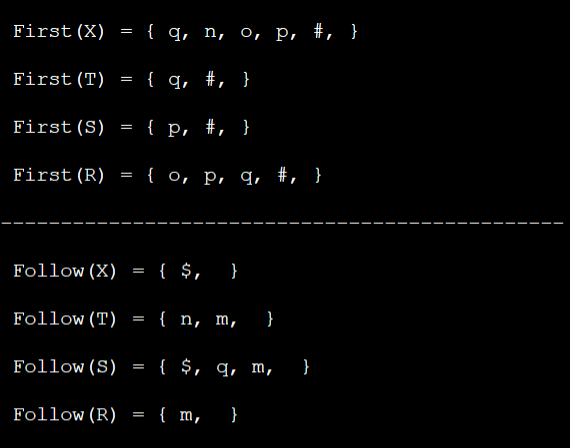
j++;

}

}

}

Output –



**Practical no. 4**

**Objective –** Write a Program to Implement LL(1) Parsing Table

**Introduction -** An LL(1) parser is a type of parser used in compiler design for constructing the parse tree of a given input string. Here’s a brief introduction to LL(1) parsers

**What is an LL(1) Parser ?**

* LL(1) stands for Left-to-right scanning of the input, Leftmost derivation, and 1 token of lookahead.
* It is a top-down parser that uses a single lookahead token to make parsing decisions.
* The parser builds a parse tree from the top (starting with the start symbol) and proceeds down

**Theory –** Predictive parsers rely on two sets associated with each non-terminal symbol in a grammar: First and Follow. The First set contains all the terminals that can appear at the beginning of some string derived from the non-terminal. The Follow set contains all the terminals that can appear immediately to the right of the non-terminal in some “sentential” form.

The predictive parsing table is a tool used to decide which production rule to use at each step in the parsing process. The table is indexed by non-terminals (rows) and terminals (columns). Each entry in the table contains a production rule that should be used if the non-terminal is the current symbol to be processed and the terminal is the current input symbol.

**Rules** –

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

**Algorithm to construct LL(1) Parsing Table:**

**Step 1:**First check all the essential conditions mentioned above and go to step 2.

**Step 2:**Calculate First() and Follow() for all non-terminals.

[**First**](https://www.geeksforgeeks.org/first-set-in-syntax-analysis/)**():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

[Follow](https://www.geeksforgeeks.org/follow-set-in-syntax-analysis/)(): What is the Terminal Symbol which follows a variable in the process of derivation.

**Step 3:**For each production A –> α. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry A –> α in the table.

If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –>  ε in the table.

If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –>  ε in the table for the $.  
To construct the parsing table, we have two functions:

**Resource - C**

**Program Logic** –

1. **Initialization:**
   * The prod array holds the production rules of the grammar.
   * The first array holds the First sets for the non-terminals.
   * The follow array holds the Follow sets for the non-terminals.
   * The table array will hold the predictive parsing table.
2. **Building the Table**:
   * The numr function converts a grammar symbol into an index for the table.
   * The table is initially filled with the string “EMPTY”.
   * The main function then populates the table based on the First and Follow sets:
     + If the First set of a production does not contain the empty string ('@'), the production is placed in the table at the intersection of the row for the non-terminal on the left-hand side of the production and the column for the terminal in the First set.
     + If the First set contains the empty string, the production is placed in the table at the intersection of the row for the non-terminal and the column for the terminal in the Follow set.
3. **Printing the Table:**
   * The table is printed in a formatted manner, with separators for readability.

This code assumes that the grammar is already in a form suitable for predictive parsing (no left recursion, factored, etc.). The predictive parsing table is used by the parser to decide which production to use when parsing an input string according to the grammar.

**Program -**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char prod[3][10] = {"A->aBa", "B->bB", "B->@"};

char first[3][10] = {"a", "b", "@"};

char follow[3][10] = {"$", "a", "a"};

char table[3][4][10];

int numr(char c) {

switch (c) {

case 'A':

return 1;

case 'B':

return 2;

case 'a':

return 1;

case 'b':

return 2;

case '@':

return 3;

}

return 1;

}

int main() {

int i, j;

for (i = 0; i < 3; i++) {

for (j = 0; j < 4; j++) {

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

}

}

printf("\n Grammar\n");

for (i = 0; i < 3; i++)

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

printf("\nfirst ={%s,%s,%s}", first[0], first[1], first[2]);

printf("\nfollow ={%s,%s}", follow[0], follow[1]);

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

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

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

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

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

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

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

for (i = 0; i < 3; i++) {

if (first[i][0] != '@') // A->a

strcpy(table[numr(prod[i][0])][numr(first[i][0])], prod[i]);

else

strcpy(table[numr(prod[i][0])][numr(follow[i][0])], prod[i]);

}

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

for (i = 0; i < 3; i++) {

for (j = 0; j < 4; j++) {

printf("%-30s", table[i][j]);

if (j == 3)

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

}

}

}

**Output -**

