**To demonstrate whether input identifier is number or character:**

#include <iostream>

#include <cctype> // For isdigit and isalpha functions

using namespace std;

int main() {

string input;

cout << "Enter an identifier: ";

cin >> input;

bool isNumber = true;

bool isCharacter = true;

// Check if the input is a number

for (char ch : input) {

if (!isdigit(ch)) {

isNumber = false;

break;

}

}

// Check if the input is a character

for (char ch : input) {

if (!isalpha(ch)) {

isCharacter = false;

break;

}

}

// Output results

if (isNumber) {

cout << "This is a number: " << input << endl;

} else if (isCharacter) {

cout << "This is a character: " << input << endl;

} else {

cout << "Unknown input: " << input << endl;

}

return 0;

}

**Output:**

Enter an identifier: 10

This is a number: 10

**To implement a program to find first() for any given production:**

#include <iostream>

#include <map>

#include <set>

#include <vector>

#include <sstream>

using namespace std;

map<char, vector<string>> grammar;

map<char, set<char>> firstSet;

bool isTerminal(char symbol) { return !isupper(symbol) && symbol != '#'; }

set<char> calculateFirst(char symbol) {

if (!firstSet[symbol].empty()) return firstSet[symbol];

for (const string &prod : grammar[symbol]) {

for (char sym : prod) {

if (isTerminal(sym)) { firstSet[symbol].insert(sym); break; }

set<char> firstSym = calculateFirst(sym);

firstSet[symbol].insert(firstSym.begin(), firstSym.end());

if (!firstSym.count('#')) break;

}

}

return firstSet[symbol];

}

int main() {

int n; cout << "Enter number of productions: "; cin >> n; cin.ignore();

while (n--) {

string input, prod; cout << "Enter production (e.g., S -> AB|a): ";

getline(cin, input);

char nonTerminal = input[0];

stringstream ss(input.substr(5));

while (getline(ss, prod, '|')) grammar[nonTerminal].push\_back(prod);

}

for (auto &rule : grammar) calculateFirst(rule.first);

for (auto &rule : firstSet) {

cout << "FIRST(" << rule.first << ") = { ";

for (char term : rule.second) cout << term << " ";

cout << "}\n";

}

}

**Output:**

Enter number of productions: 3

Enter production (e.g., S -> AB|a): S -> AB|a

Enter production (e.g., S -> AB|a): A -> a|b

Enter production (e.g., S -> AB|a): B -> c|#

FIRST(#) = { }

FIRST(A) = { a b }

FIRST(B) = { c }

FIRST(S) = { a b }

**To implement a program to find follow() for any given production:**

#include <iostream>

#include <map>

#include <set>

#include <vector>

#include <sstream>

using namespace std;

map<char, vector<string>> grammar;

map<char, set<char>> firstSet, followSet;

bool isTerminal(char c) { return !isupper(c); } // Terminal is not a non-terminal

// Function to calculate FIRST set

void calculateFirst(char symbol) {

if (!firstSet[symbol].empty()) return; // Already calculated

for (const string &prod : grammar[symbol]) {

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

char ch = prod[i];

if (isTerminal(ch)) {

firstSet[symbol].insert(ch); // Terminal, add to FIRST

break;

} else {

calculateFirst(ch); // Non-terminal, calculate FIRST of ch

bool hasEpsilon = false;

for (char f : firstSet[ch]) {

if (f == '#') hasEpsilon = true;

else firstSet[symbol].insert(f);

}

if (!hasEpsilon) break; // Stop if epsilon not found

}

}

}

}

// Function to calculate FOLLOW set

void calculateFollow(char symbol) {

for (auto &rule : grammar) {

char lhs = rule.first;

for (const string &prod : rule.second) {

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

if (prod[i] == symbol) {

if (i == prod.size() - 1 && lhs != symbol)

followSet[symbol].insert(followSet[lhs].begin(), followSet[lhs].end());

else if (i < prod.size() - 1) {

char next = prod[i + 1];

if (isTerminal(next)) followSet[symbol].insert(next);

else {

for (char f : firstSet[next]) if (f != '#') followSet[symbol].insert(f);

if (firstSet[next].count('#')) followSet[symbol].insert(followSet[lhs].begin(), followSet[lhs].end());

}

}

}

}

}

}

}

int main() {

int n;

cout << "Enter number of productions: ";

cin >> n; cin.ignore();

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

string input;

cout << "Enter production (e.g., S -> AB|a): ";

getline(cin, input);

char nonTerminal = input[0];

string rhs = input.substr(5);

stringstream ss(rhs);

string production;

while (getline(ss, production, '|')) {

grammar[nonTerminal].push\_back(production);

}

}

// Calculate FIRST sets for all non-terminals

for (auto &rule : grammar) {

calculateFirst(rule.first);

}

// Start FOLLOW calculation with start symbol

followSet[grammar.begin()->first].insert('$');

bool changed;

do {

changed = false;

for (const auto &rule : grammar) {

size\_t before = followSet[rule.first].size();

calculateFollow(rule.first);

if (followSet[rule.first].size() > before) changed = true;

}

} while (changed);

// Output FIRST sets

cout << "\nFIRST sets:\n";

for (const auto &rule : firstSet) {

cout << "FIRST(" << rule.first << ") = { ";

for (char c : rule.second) {

if (c != '#') cout << c << " "; // Don't print epsilon (#)

}

cout << "}\n";

}

// Output FOLLOW sets

cout << "\nFOLLOW sets:\n";

for (const auto &rule : followSet) {

cout << "FOLLOW(" << rule.first << ") = { ";

for (char c : rule.second) {

cout << c << " ";

}

cout << "}\n";

}

return 0;

}

**Output :**

Enter number of productions:

3

Enter production (e.g., S -> AB|a): S -> AB|a

Enter production (e.g., S -> AB|a): A -> a|#

Enter production (e.g., S -> AB|a):

B -> bB -> b

FIRST sets:

FIRST(A) = { a }

FIRST(B) = { b }

FIRST(S) = { a b }

FOLLOW sets:

FOLLOW(A) = { $ b }

FOLLOW(B) = { }

FOLLOW(S) = { }

**to implement bottom up parser:**

#include <iostream>

#include <stack>

#include <string>

using namespace std;

bool isTerminal(char symbol) {

return symbol == 'i' || symbol == '+' || symbol == '\*' || symbol == '$';

}

// Function to reduce stack by grammar rules

bool reduce(stack<string> &parseStack) {

if (parseStack.size() < 3) return false;

string top2 = parseStack.top(); parseStack.pop();

string top1 = parseStack.top(); parseStack.pop();

string top0 = parseStack.top();

if (top0 == "E" && top1 == "+" && top2 == "E") {

parseStack.pop();

parseStack.push("E"); // E -> E + E

cout << "Reduce by E -> E + E\n";

return true;

} else if (top0 == "E" && top1 == "\*" && top2 == "E") {

parseStack.pop();

parseStack.push("E"); // E -> E \* E

cout << "Reduce by E -> E \* E\n";

return true;

} else {

parseStack.push(top0);

parseStack.push(top1);

parseStack.push(top2);

}

return false;

}

// Function to perform reduction for a single `id` to `E`

bool reduceSingle(stack<string> &parseStack) {

if (parseStack.top() == "id") {

parseStack.pop();

parseStack.push("E"); // E -> id

cout << "Reduce by E -> id\n";

return true;

}

return false;

}

// Parsing function

bool parse(const string &input) {

stack<string> parseStack;

int i = 0;

while (i < input.size()) {

char curr = input[i];

string action;

if (curr == 'i' && input.substr(i, 2) == "id") {

action = "Shift id";

parseStack.push("id");

i += 2;

} else if (curr == '\*' || curr == '+') {

action = "Shift " + string(1, curr);

parseStack.push(string(1, curr));

i++;

} else if (curr == '$') {

parseStack.push("$");

action = "Shift $";

break;

} else {

cout << "Invalid symbol encountered.\n";

return false;

}

cout << action << endl;

reduceSingle(parseStack);

while (reduce(parseStack));

}

// Final reductions to see if we have a single E on the stack

reduceSingle(parseStack);

while (reduce(parseStack));

// Ensure only E and $ remain on the stack for a valid expression

if (parseStack.size() == 2 && parseStack.top() == "$") {

parseStack.pop(); // Pop the $

return parseStack.top() == "E"; // Only E should remain

}

return false;

}

int main() {

string input;

cout << "Enter an expression to parse (e.g., id+id\*id): ";

cin >> input;

input += "$"; // End marker

if (parse(input)) {

cout << "Input accepted!" << endl;

} else {

cout << "Input rejected!" << endl;

}

return 0;

}

**Output :**

Enter an expression to parse (e.g., id+id\*id): id\*id+id

Shift id

Reduce by E -> id

Shift \*

Shift id

Reduce by E -> id

Reduce by E -> E \* E

Shift +

Shift id

Reduce by E -> id

Reduce by E -> E + E

Input accepted!

=== Code Execution Successful ===