**EXPERIMENT - 7**

**AIM:** Write a program to find out LEADINGS of the non-terminals in a grammar

# THEORY:

Leading is defined for every non-terminal. Terminals that can be the first terminal in a string derived from that non-terminal.

LEADING(A) = { a | A =>+ γaδ }, where

γ is ε or any non-terminal,

=>+ indicates derivation in one or more steps,

A is a non-terminal.

Algorithm for LEADING(A) {

1. ‘a’ is in LEADING(A) is A→ γaδ where γ is ε or any non-terminal.

2. If ‘a’ is in LEADING(B) and A→B, then ‘a’ is in LEADING(A).

}

# SOURCE CODE:

#include <iostream>

#include <unordered\_set>

#include <vector>

#include <map>

using namespace std;

// Function to find leading set of non-terminals

void findLeadingSets(const map<char, vector<string>> &grammar,

map<char, unordered\_set<char>> &leadingSets)

{

bool changed = true;

while (changed)

{

changed = false;

for (const auto &rule : grammar)

{

char nonTerminal = rule.first;

const vector<string> &productions = rule.second;

for (const string &production : productions)

{

for (char symbol : production)

{

// If the symbol is a terminal, add it to the leading set

if (!isupper(symbol))

{

if (leadingSets[nonTerminal].insert(symbol).second)

{

changed = true;

}

break;

}

// If the symbol is a non-terminal, add its leading set to the current non-

terminal's leading set

else

{

for (char leadingSymbol : leadingSets[symbol])

{

if (leadingSets[nonTerminal].insert(leadingSymbol).second)

{

changed = true;

}

}

// If the non-terminal can derive epsilon, continue to the next symbol in

the production

if (!leadingSets[symbol].count('e'))

{

break;

}

}

}

}

}

}

}

int main()

{

// Define the grammar

map<char, vector<string>> grammar = {

{'S', {"aBD", "bAc", "A"}},

{'A', {"d", "e"}},

{'B', {"f"}},

{'C', {"g", "h"}},

{'D', {"i", "e"}}};

// Initialize leading sets

map<char, unordered\_set<char>> leadingSets;

for (const auto &rule : grammar)

{

leadingSets[rule.first] = {};

}

// Find leading sets

findLeadingSets(grammar, leadingSets);

// Print leading sets

for (const auto &leadingSet : leadingSets)

{

cout << "Leading(" << leadingSet.first << ") = { ";

for (char symbol : leadingSet.second)

{

cout << symbol << " ";

}

cout << "}" << endl;

}

return 0;

}

# OUTPUT:

Leading(A) = { e d }

Leading(B) = { f }

Leading(C) = { h g }

Leading(D) = { e i }

Leading(S) = { d e b a }

**EXPERIMENT - 8**

**AIM:** Write a program to implement shift reducing parsing on a string.

# THEORY:

Shift-reduce parsing is a type of top-down parsing algorithm that utilizes a stack to process input symbols and match them against the rules of a grammar. It consists of two main operations:

Shift: The parser removes the current input symbol from the input stream and pushes it onto the stack.

Reduce: The parser pops a sequence of symbols from the stack, which corresponds to the right- hand side of a grammar rule, and replaces it with the left-hand side of the rule. This effectively reduces the grammar "depth".

The parsing process continues until the stack contains only the start symbol of the grammar, indicating that the input string has been successfully parsed.

# SOURCE CODE:

#include <iostream>

#include <vector>

#include <map>

#include <sstream>

using namespace std;

bool endsWithSubstring(const std::string &str, const std::string &subStr)

{

size\_t strLen = str.length();

size\_t subStrLen = subStr.length();

return strLen >= subStrLen && str.compare

(strLen - subStrLen, subStrLen, subStr) == 0;

}

bool check(const map<char, vector<string>> &grammar,

string &stack,

string &input)

{

for (const auto &rule : grammar)

{

char nonTerminal = rule.first;

const vector<string> &productions = rule.second;

for (const string &production : productions)

{

if (endsWithSubstring(stack, production))

{

for (int i = 0; i < production.size(); i++)

stack.pop\_back();

stack.push\_back(nonTerminal);

stringstream ss;

ss << "Reduce " << nonTerminal << " -> " << production;

string action = ss.str();

cout << stack << "\t\t" << input << "\t\t" << action << endl;

check(grammar, stack, input);

}

}

}

return false;

}

void SRParser(const map<char, vector<string>> &grammar, string &input)

{

string stack;

cout << "Stack" << "\t\t";

cout << "Input" << "\t\t";

cout << "Action" << endl;

input.insert(input.begin(), '$');

input.push\_back('$');

int iterations = input.size();

char inputChar;

for (size\_t i = 0; i < iterations; i++)

{

inputChar = input[0];

input = input.substr(1);

stack.push\_back(inputChar);

cout << stack << "\t\t" << input << "\t\tShift " << inputChar << endl;

check(grammar, stack, input);

}

if (stack[0] == '$' && stack[2] == '$')

{

cout << "String Accepted" << endl;

}

else

{

cout << "String Rejected" << endl;

}

}

int main()

{

// Define the grammar

// E -> E+E

// E -> E-E

// E -> (E)

// E -> a

map<char, vector<string>> grammar = {

{'E', {"E+E", "E-E", "(E)", "a"}},

};

string input = "a-(a+a)";

SRParser(grammar, input);

return 0;

}

# OUTPUT:

Stack Input Action

$ a-(a+a)$ Shift $

$a -(a+a)$ Shift a

$E -(a+a)$ Reduce E -> a

$E- (a+a)$ Shift -

$E-( a+a)$ Shift (

$E-(a +a)$ Shift a

$E-(E +a)$ Reduce E -> a

$E-(E+ a)$ Shift +

$E-(E+a )$ Shift a

$E-(E+E )$ Reduce E -> a

$E-(E )$ Reduce E -> E+E

$E-(E) $ Shift )

$E-E $ Reduce E -> (E)

$E $ Reduce E -> E-E

$E$ Shift $

String Accepted

**EXPERIMENT – 9**

**AIM:** Write a program to find FIRST of non-terminals of the given grammar

# THEORY:

The First set of a non-terminal symbol (A) in a context-free grammar (CFG) is the set of all terminal symbols and empty strings (ε) that can be generated by starting from A and applying the rules of the grammar. In simpler terms, it represents the possible first symbols that can appear in a parse tree rooted at A.

# SOURCE CODE:

#include <iostream>

#include <map>

#include <set>

#include <vector>

using namespace std;

// Function to compute the FIRST set for a given non-terminal

set<char> computeFirst(const char nonTerminal, const map<char, vector<string>> &grammar, map<char, set<char>> &firstSet)

{

if (firstSet.find(nonTerminal) != firstSet.end())

{

// If FIRST set for the non-terminal is already computed, return it

return firstSet[nonTerminal];

}

set<char> result;

for (const string &production : grammar.at(nonTerminal))

{

char firstSymbol = production[0];

if (isupper(firstSymbol))

{

// If the first symbol is a non-terminal, recursively compute FIRST set

set<char> firstSetOfFirstSymbol =

computeFirst(firstSymbol, grammar, firstSet);

// Add all symbols from the FIRST set of the first symbol to the result

result.insert(firstSetOfFirstSymbol.begin(),

firstSetOfFirstSymbol.end());

// Check if epsilon is in the FIRST set of the first symbol

bool epsilonFound = firstSetOfFirstSymbol.find('e') !=

firstSetOfFirstSymbol.end();

// If epsilon is in the FIRST set of the first symbol, consider the next symbol in the production

if (epsilonFound && production.length() > 1)

{

firstSymbol = production[1];

result.insert(firstSymbol);

}

}

else

{

// If the first symbol is a terminal, add it to the result

result.insert(firstSymbol);

}

}

// Cache the computed FIRST set for the non-terminal

firstSet[nonTerminal] = result;

return result;

}

int main()

{

// Define the grammar

map<char, vector<string>> grammar = {

{'S', {"aBC", "bCd", "B"}},

{'B', {"g", "e"}},

{'C', {"h", "e"}},

{'D', {"i", "e"}},

};

// Map to store the FIRST sets for each non-terminal

map<char, set<char>> firstSet;

// Compute FIRST sets for each non-terminal in the grammar

for (const auto &rule : grammar)

{

char nonTerminal = rule.first;

computeFirst(nonTerminal, grammar, firstSet);

}

// Print the computed FIRST sets

for (const auto &entry : firstSet)

{

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

for (char symbol : entry.second)

{

cout << symbol << " ";

}

cout << "}" << endl;

}

return 0;

}

# OUTPUT:

FIRST(B) = { e g }

FIRST(C) = { e h }

FIRST(D) = { e i }

FIRST(S) = { a b e g }

**EXPERIMENT – 10**

**AIM:** Write a program to check whether a grammar is operator precedent

# THEORY:

Operator precedence is a mechanism used in programming languages to specify the order of operations when evaluating expressions. It defines the order in which operators should be applied when they appear in the same expression. This is particularly important when multiple operators are present and their order is not explicitly specified by parentheses.

In operator precedence grammars, the precedence relationships between operators are defined explicitly within the grammar itself. This allows for efficient parsing and evaluation of expressions without relying on additional parsing rules or context-dependent information.

# SOURCE CODE:

#include<iostream>

#include<string>

#include<deque>

using namespace std;

int n,n1,n2;

int getPosition(string arr[], string q, int size)

{

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

{

if(q == arr[i])

return i;

}

return -1;

}

int main()

{

string prods[10],leads[10],trails[10],nonterms[10],terms[10];

char op\_table[20][20] = {};

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

cin>>n;

cin.ignore();

cout<<"Enter the productions"<<endl;

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

{

getline(cin,prods[i]);

}

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

cin>>n2;

cin.ignore();

cout<<"Enter the Terminals"<<endl;

for(int i=0;i<n2;i++)

{

cin>>terms[i];

}

terms[n2] = "$";

n2++;

cout<<"Enter the number of Non-Terminals : ";

cin>>n1;

cin.ignore();

for(int i=0;i<n1;i++)

{

cout<<"Enter Non-Terminal : ";

getline(cin,nonterms[i]);

cout<<"Enter Leads of "<<nonterms[i]<<" : ";

getline(cin,leads[i]);

cout<<"Enter Trails of "<<nonterms[i]<<" : ";

getline(cin,trails[i]);

}

cout<<"Enter the Rules (exit to stop)"<<endl;

string rule = "";

while(rule != "exit")

{

getline(cin,rule);

if(rule[0] == '1')

{

int row = getPosition(terms,rule.substr(2,1),n2);

int column = getPosition(terms,rule.substr(4,1),n2);

op\_table[row][column] = '=';

}

if(rule[0] == '2')

{

int ntp = getPosition(nonterms,rule.substr(4,1),n1);

int row = getPosition(terms,rule.substr(2,1),n2);

for(int j=0;j<leads[ntp].size();j++)

{

int col = getPosition(terms,leads[ntp].substr(j,1),n2);

op\_table[row][col] = '<';

}

}

if(rule[0] == '3')

{

int col = getPosition(terms,rule.substr(4,1),n2);

int ntp = getPosition(nonterms,rule.substr(2,1),n1);

for(int j=0;j<trails[ntp].size();j++)

{

int row = getPosition(terms,trails[ntp].substr(j,1),n2);

op\_table[row][col] = '>';

}

}

}

for(int j=0;j<leads[0].size();j++)

{

int col = getPosition(terms,leads[0].substr(j,1),n2);

op\_table[n2-1][col] = '<';

}

for(int j=0;j<trails[0].size();j++)

{

int row = getPosition(terms,trails[0].substr(j,1),n2);

op\_table[row][n2-1] = '>';

}

cout<<endl;

cout<<"Grammar"<<endl;

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

{

cout<<prods[i]<<endl;

}

//Display Table

for(int j=0;j<n2;j++)

cout<<"\t"<<terms[j];

cout<<endl;

for(int i=0;i<n2;i++)

{

cout<<terms[i]<<"\t";

for(int j=0;j<n2;j++)

{

cout<<op\_table[i][j]<<"\t";

}

cout<<endl;

}

//Parsing String

char c;

do {

string ip;

deque<string> op\_stack;

op\_stack.push\_back("$");

cout<<"Enter the string to be parsed : ";

getline(cin,ip);

ip.push\_back('$');

cout<<"Stack\ti/p Buffer\tRelation\tAction"<<endl;

while(true)

{

for(int i=0;i<op\_stack.size();i++)

cout<<op\_stack[i];

cout<<"\t";

cout<<ip<<"\t";

int row = getPosition(terms,op\_stack.back(),n2);

int column = getPosition(terms,ip.substr(0,1),n2);

if(op\_table[row][column] == '<')

{

op\_stack.push\_back("<");

op\_stack.push\_back(ip.substr(0,1));

ip = ip.substr(1);

cout<<"\t"<<"<\t\tPush";

}

else if(op\_table[row][column] == '=')

{

op\_stack.push\_back("=");

op\_stack.push\_back(ip.substr(0,1));

ip = ip.substr(1);

cout<<"\t"<<"=\t\tPush";

}

else if(op\_table[row][column] == '>')

{

string last;

do

{

op\_stack.pop\_back();

last = op\_stack.back();

op\_stack.pop\_back();

}while(last != "<");

cout<<"\t"<<">\t\tPop";

}

else

{

if(ip[0] == '$' && op\_stack.back() == "$")

{

cout<<"\t\t\tAccept\nString Parsed."<<endl;

break;

}

else

{

cout<<endl<<"String cannot be Parsed."<<endl;

break;

}

}

cout<<endl;

}

cout<<"Continue?(Y/N) ";

cin>>c;

cin.ignore();

}

while(c=='y' || c=='Y');

return 0;

}

# OUTPUT:

Enter the number of productions : 4

Enter the productions

S->aAcBe

A->Ab

A->b

B->d

Enter the number of Terminals : 5

Enter the Terminals

a

b

c

d

e

Enter the number of Non-Terminals : 3

Enter Non-Terminal : S

Enter Leads of S : a

Enter Trails of S : e

Enter Non-Terminal : A

Enter Leads of A : bd

Enter Trails of A : bd

Enter Non-Terminal : B

Enter Leads of B : d

Enter Trails of B : d

Enter the Rules (exit to stop)

1 a c

1 c e

2 a A

2 c B

3 A c

3 B e

3 A b

exit

Grammar

S->aAcBe

A->Ab

A->b

B->d

a b c d e $

a < = <

b > >

c < =

d > > >

e >

$ <

Enter the string to be parsed : abcde

Stack i/p Buffer Relation Action

$ abcde$ < Push

$<a bcde$ < Push

$<a<b cde$ > Pop

$<a cde$ = Push

$<a=c de$ < Push

$<a=c<d e$ > Pop

$<a=c e$ = Push

$<a=c=e $ > Pop

$ $ Accept

String Parsed.

Continue?(Y/N) Y

Enter the string to be parsed : abde

Stack i/p Buffer Relation Action

$ abde$ < Push

$<a bde$ < Push

$<a<b de$

String cannot be parsed.

Continue?(Y/N) N