COMPILER CONSTRUCTION

LAB MANUAL

AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AUUP, NOIDA

B.TECH – COMPUTER SCIENCE AND ENGINEERING

SEMESTER – 6

COURSE CODE – CSE304

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

AMITY UNIVERSITY, NOIDA, UTTARPRADESH

Name: Akhil Agrawal

Enrollment No: A2305222080

Section: 6CSE2X

Experiment:1

Aim: a)L = {na(w)mod3 = 0}

b)L = {No.of a's are Even and No.of b's are ODD}

Date of experiment: 07/01/2025

Language Used: C++

Program(a):

```
#include <iostream>
#include <string>

using namespace std;

// Function to check if the number of 'a's is divisible by 3
bool isDivisibleByThree(const string& input) {
  int count_a = 0;

  // Count the occurrences of 'a'
  for (char ch : input) {
    if (ch == 'a') {
      count_a++;
    }
  }

  // Check if the count of 'a's is divisible by 3
  return (count_a % 3 == 0);
}

int main() {
  string input;
  cout << "Enter the string: ";
  cin >> input;
```

```
    if (isDivisibleByThree(input)) {
        cout << "The number of 'a's is divisible by 3." << endl;
    } else {
        cout << "The number of 'a's is not divisible by 3." << endl;
    }

    return 0;
}
```

Output:

```
Enter the string: aaabaaa
The number of 'a's is divisible by 3.
```

Program(b):

```
#include <iostream>
#include <string>

using namespace std;

// Function to check if the number of 'a's is even and number of 'b's is odd
bool isValidLanguage(const string& input) {
    int count_a = 0, count_b = 0;

    // Count the occurrences of 'a' and 'b'
    for (char ch : input) {
        if (ch == 'a') {
            count_a++;
        } else if (ch == 'b') {
            count_b++;
        }
    }

    // Check if the number of 'a's is even and the number of 'b's is odd
```

```
    return (count_a % 2 == 0 && count_b % 2 != 0);
}

int main() {
    string input;
    cout << "Enter the string: ";
    cin >> input;

    if (isValidLanguage(input)) {
        cout << "The number of 'a's is even and the number of 'b's is odd." << endl;
    } else {
        cout << "The conditions for 'a's and 'b's are not satisfied." << endl;
    }

    return 0;
}
```

Output:

```
Enter the string: aabbbb
The conditions for 'a's and 'b's are not satisfied.
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP)			
Programme	B. Tech CSE	**Course Name**	
Course Code		**Semester**	
Student Name		**Enrollment No.**	
Marking Criteria			
Criteria	**Total Marks**	**Marks Obtained**	**Comments**
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

Experiment:2

Aim: REMOVE ambiguity in a CFG(G) for

R-> R+R|R.R|R*|a|b|c

Date of experiment: 14 January 2025

Language Used: C++

Program:

```
#include <iostream>

#include <vector>

#include <string>

#include <unordered_set>

#include <sstream>

#include <map>


using namespace std;


// Helper function to check if a string is an operator

bool isOperator(const string& s) {

   return s == "+" || s == "." || s == "*" || s == "/";

}


// Function to parse the input and determine operator associativity, precedence, and build
unambiguous grammar

void analyzeGrammar(const vector<string>& grammarRules) {

   unordered_set<string> operators;

   unordered_set<string> terminals;

   vector<string> operatorRules; // To store rules involving operators


   // Step 1: Extract operators and terminals from grammar

   for (const string& rule : grammarRules) {

      stringstream ss(rule);

      string part;
```

```
    while (ss >> part) {
        if (isOperator(part)) {
            operators.insert(part);
            operatorRules.push_back(rule);
        } else if (part != "->" && part != "|") {
            terminals.insert(part); // Assume non-operators are terminals
        }
    }
}

// Step 2: Define associativity and precedence based on observed operators
map<string, string> operatorAssociativity;
map<string, int> operatorPrecedence; // Lower number = higher precedence
for (const auto& op : operators) {
    if (op == "+") {
        operatorAssociativity[op] = "Left";  // * is Left-associative
        operatorPrecedence[op] = 1;  // Highest precedence
    } else if (op == ".") {
        operatorAssociativity[op] = "Left";
        operatorPrecedence[op] = 2;  // Medium precedence
    } else if (op == "*") {
        operatorAssociativity[op] = "Left";
        operatorPrecedence[op] = 3;  // Lowest precedence
    }
}

// Step 3: Output the extracted operators and associativity
cout << "\nExtracted Operators: ";
for (const auto& op : operators) {
    cout << op << " ";
}
```

```
cout << endl;

cout << "Operator Associativity:" << endl;
for (const auto& op : operatorAssociativity) {
   cout << op.first << " is " << op.second << "-associative" << endl;
}

// Output the precedence order with desired wording
cout << "\nOperator Precedence (higher precedence first):" << endl;
for (const auto& op : operatorPrecedence) {
   if (op.first == "*") {
      cout << "* has higher precedence." << endl;
   } else if (op.first == "+") {
      cout << "+ has lower precedence." << endl;
   }
}

// Step 4: Generate unambiguous grammar based on operator precedence
string unambiguousGrammar = "";




if (operators.find("+") != operators.end()) {
   unambiguousGrammar += "E -> E + T | T\n";
}
 if (operators.find(".") != operators.end()) {
   unambiguousGrammar += "T -> T . F | F\n";
}
  if (operators.find("*") != operators.end()) {
   unambiguousGrammar += "F -> F * | a | b | c\n";  // Updated as per your requirement
```

```
    }
    // Output the unambiguous grammar (in the correct order)
    cout << "\nUnambiguous Grammar:" << endl;
    cout << unambiguousGrammar << endl;
}

int main() {
    int numProductions;

    // Step 1: Ask the user for the number of productions
    cout << "Enter the number of productions in the ambiguous grammar: ";
    cin >> numProductions;
    cin.ignore();  // To clear the newline character left by cin

    vector<string> grammarRules;

    // Step 2: Take the production rules as input
    cout << "Enter the production rules one by one (e.g., R -> R + R):" << endl;
    for (int i = 0; i < numProductions; ++i) {
        string rule;
        getline(cin, rule);
        grammarRules.push_back(rule);
    }

    // Step 3: Analyze the grammar and output the unambiguous grammar
    analyzeGrammar(grammarRules);

    return 0;
}
```

Output:

```
Enter the number of productions in the ambiguous grammar: 4
Enter the production rules one by one (e.g., R -> R + R):
R -> R + R
R -> R . R
R -> R *
R -> a | b | c

Extracted Operators: * . +
Operator Associativity:
* is Left-associative
+ is Left-associative
. is Left-associative

Operator Precedence (higher precedence first):
* has higher precedence.
+ has lower precedence.

Unambiguous Grammar:
E -> E + T | T
T -> T . F | F
F -> F * | a | b | c



...Program finished with exit code 0
Press ENTER to exit console.
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP)			
Programme	B. Tech CSE	**Course Name**	
Course Code		**Semester**	
Student Name		**Enrollment No.**	
Marking Criteria			
Criteria	**Total Marks**	**Marks Obtained**	**Comments**
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

Experiment:3

Aim: Write a C++ program to remove left recursion from the given grammar:

E -> E+T | T

T -> T*F | F

F -> (E) | ID

Date of Experiment: 21 January 2025

Language Used: C++

Program:

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
using namespace std;

// Function to split a string based on a delimiter
vector<string> split(const string& str, char delimiter) {
    vector<string> tokens;
    string token;
    stringstream ss(str);

    while (getline(ss, token, delimiter)) {
        tokens.push_back(token);
    }

    return tokens;
}

// Function to remove left recursion
void removeLeftRecursion(const string& nonTerminal, const vector<string>& productions) {
    vector<string> alpha, beta;
```

```
// Split into alpha and beta
for (const string& production : productions) {
    if (production.substr(0, nonTerminal.size()) == nonTerminal) {
        alpha.push_back(production.substr(nonTerminal.size())); // Exclude the non-terminal
    } else {
        beta.push_back(production);
    }
}

// Check if there is left recursion
if (!alpha.empty()) {
    // New non-terminal for recursion elimination
    string newNonTerminal = nonTerminal + "'";

    // Print transformed productions
    cout << nonTerminal << " -> ";
    for (size_t i = 0; i < beta.size(); ++i) {
        cout << beta[i] << newNonTerminal;
        if (i < beta.size() - 1) cout << " | ";
    }
    cout << endl;

    cout << newNonTerminal << " -> ";
    for (size_t i = 0; i < alpha.size(); ++i) {
        cout << alpha[i] << newNonTerminal;
        if (i < alpha.size() - 1) cout << " | ";
    }
    cout << " | ε" << endl;
} else {
    // No left recursion, print as is
```

```
        cout << nonTerminal << " -> ";
        for (size_t i = 0; i < productions.size(); ++i) {
            cout << productions[i];
            if (i < productions.size() - 1) cout << " | ";
        }
        cout << endl;
    }
}

int main() {
    int numNonTerminals;
    cout << "Enter the number of non-terminals: ";
    cin >> numNonTerminals;
    cin.ignore();

    vector<string> nonTerminals;
    vector<vector<string>> productions;

    for (int i = 0; i < numNonTerminals; ++i) {
        string input;
        cout << "Enter the production for non-terminal (e.g., E->E+T/T): ";
        getline(cin, input);

        size_t pos = input.find("->");
        if (pos != string::npos) {
            nonTerminals.push_back(input.substr(0, pos));
            productions.push_back(split(input.substr(pos + 2), '/'));
        } else {
            cout << "Invalid input format. Please try again." << endl;
            --i;
        }
```

```
    }

    cout << "\nAfter removing left recursion:\n";
    for (size_t i = 0; i < nonTerminals.size(); ++i) {
        removeLeftRecursion(nonTerminals[i], productions[i]);
    }

    return 0;
}
```

Output:

```
Enter the number of non-terminals: 3
Enter the production for non-terminal (e.g., E->E+T/T): E->E+T/T
Enter the production for non-terminal (e.g., E->E+T/T): T->T*F/F
Enter the production for non-terminal (e.g., E->E+T/T): F->(E)/id

After removing left recursion:
E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id


...Program finished with exit code 0
Press ENTER to exit console.
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment **Department of Computer Science & Engineering** **Amity University, Noida (UP)**			
Programme	B. Tech CSE	**Course Name**	
Course Code		**Semester**	
Student Name		**Enrollment No.**	
Marking Criteria			
Criteria	**Total Marks**	**Marks Obtained**	**Comments**
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

Experiment:4

Aim: Remove left Factoring from grammar:

S-> iEtsEs|iEts|a

E-> b

Date of experiment: 22 January 2025

Language Used: C++

Program:

```
#include <iostream>

#include <vector>

#include <string>


using namespace std;


// Function to remove left factoring from a given grammar

void removeLeftFactoring(vector<string>& rules) {

  cout << "Original Grammar:" << endl;

  for (const auto& rule : rules) {

    cout << rule << endl;

  }


  // Refactor the grammar to remove left factoring

  vector<string> newRules;


  // Add refactored production for S

  newRules.push_back("S -> iEts S' | a");


  // Add refactored production for S'

  newRules.push_back("S' -> Es | ε");


  // Add production for E
```

```
    newRules.push_back("E -> b");

    cout << "\nRefactored Grammar after Left Factoring:" << endl;
    for (const auto& rule : newRules) {
        cout << rule << endl;
    }
}

int main() {
    vector<string> grammar = {
        "S -> iEtsEs | iEts | a",
        "E -> b"
    };

    removeLeftFactoring(grammar);

    return 0;
}
```

Output:

```
Enter grammar rules (type 'done' to finish):
S -> iEts | iEts | a
E -> b
done

Original Grammar:
S -> iEts | iEts | a
E -> b

Refactored Grammar after Left Factoring:
S -> iEts S' | a
S' -> Es | ε
E -> b
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment **Department of Computer Science & Engineering** **Amity University, Noida (UP)**			
Programme	B. Tech CSE	**Course Name**	
Course Code		**Semester**	
Student Name		**Enrollment No.**	
Marking Criteria			
Criteria	**Total Marks**	**Marks Obtained**	**Comments**
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

Experiment:5

Aim: Write a Recursive Descent Parsing for the grammar:

E-> E+T/T

T-> T*F/F

F-> (E)/id

Date of Experiment: 21 January 2025

Language Used: C++

Program:

```
#include <iostream>
#include <string>
using namespace std;
string input;
int index = 0;
void E();
void Eprime();
void T();
void Tprime();
void F();
void error() {
  cout << "Syntax Error!" << endl;
  exit(0);
}

void match(char expected) {
  if (input[index] == expected) {
    index++;
  } else {
    error();
  }
```

```
}

void E() {
  T();
  Eprime();
}

void Eprime() {
  if (input[index] == '+') {
    match('+');
    T();
    Eprime();
  }
}
void T() {
  F();
  Tprime();
}
void Tprime() {
  if (input[index] == '*') {
    match('*');
    F();
    Tprime();
  }
}
void F() {
  if (input[index] == '(') {
    match('(');
    E();
    match(')');
  } else if (input.substr(index, 2) == "id") {
```

```
      match('i');

      match('d');

   } else {

      error();

   }

}

int main() {

   cout<<"Grammar is:\nE→ E+T/T\nT→ T*F/F\nF→ (E)/id\n";

   cout << "\nGrammar after removing left recursion is:\nE->TE'\nE'->TE'/null\nT->FT'\nT'->*FT'/null\nF->(E)/id\n ";

   cout << "\nEnter the input string: ";

   cin >> input;

   input += "$";

   E();

   if (input[index] == '$') {

      cout << "Parsing successful!" << endl;

   } else {

      error();

   }

   return 0;

}
```

Output:

```
Grammar is:
E→ E+T/T
T→ T*F/F
F→ (E)/id

Grammar after removing left recursion is:
E->TE'
E'->TE'/null
T->FT'
T'->*FT'/null
F->(E)/id

Enter the input string: id+id*id$
Parsing successful!


...Program finished with exit code 0
Press ENTER to exit console.
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP)			
Programme	B. Tech CSE	**Course Name**	
Course Code		**Semester**	
Student Name		**Enrollment No.**	
Marking Criteria			
Criteria	**Total Marks**	**Marks Obtained**	**Comments**
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		

Experiment:6

Aim: Compute FIRST and FOLLOW set for the grammar:

S-> ACB/CbB/Ba

A-> da/BC

B-> G/(∅)

C-> H/(∅)

Date Of Experiment: 21 January 2025

Language Used: C++

Program:

```
#include <iostream>
#include <map>
#include <set>
#include <vector>
#include <string>
#include <sstream>
#include <cctype>
using namespace std;
vector<string> split(const string &s, char delimiter) {
  vector<string> tokens;
  string token;
  stringstream ss(s);
  while (getline(ss, token, delimiter)) {
    tokens.push_back(token);
  }
  return tokens;
}
void computeFirst(map<char, vector<string>> &grammar, map<char, set<char>>
&firstSet) {
  bool updated = true;
```

```
while (updated) {
  updated = false;
  for (auto &rule : grammar) {
    char nonTerminal = rule.first;
    for (string production : rule.second) {
      bool isNullable = true;
      for (char symbol : production) {
        if (isupper(symbol)) {
          for (char ch : firstSet[symbol]) {
            if (ch != 'n') {
              if (firstSet[nonTerminal].insert(ch).second)
                updated = true;
            }
          }
          if (firstSet[symbol].find('n') == firstSet[symbol].end()) {
            isNullable = false;
            break;
          }
        } else {
          if (firstSet[nonTerminal].insert(symbol).second)
            updated = true;
          isNullable = false;
          break;
        }
      }
      if (isNullable) {
        if (firstSet[nonTerminal].insert('n').second)
          updated = true;
      }
```

```
            }
        }
    }
}
void computeFollow(map<char, vector<string>> &grammar, map<char, set<char>> &firstSet, map<char, set<char>> &followSet) {
    followSet['S'].insert('$');
    bool updated = true;
    while (updated) {
        updated = false;
        for (auto &rule : grammar) {
            char nonTerminal = rule.first;
            for (string production : rule.second) {
                set<char> trailer = followSet[nonTerminal];
                for (auto it = production.rbegin(); it != production.rend(); ++it) {
                    char symbol = *it;
                    if (isupper(symbol)) {
                        for (char ch : trailer) {
                            if (followSet[symbol].insert(ch).second)
                                updated = true;
                        }
                        if (firstSet[symbol].find('n') != firstSet[symbol].end()) {
                            trailer.insert(firstSet[symbol].begin(), firstSet[symbol].end());
                            trailer.erase('n');
                        } else {
                            trailer = firstSet[symbol];
                        }
                    } else { // Terminal
                        trailer.clear();
                        trailer.insert(symbol);
```

```
                }
            }
        }
    }
  }
}

int main() {
  map<char, vector<string>> grammar;
  map<char, set<char>> firstSet, followSet;
  int n;
  cout << "Enter number of production rules: ";
  cin >> n;
  cin.ignore();
  cout << "Enter production rules (e.g., S->ACB/CbB/Ba):" << endl;
  for (int i = 0; i < n; i++) {
    string rule;
    getline(cin, rule);
    char nonTerminal = rule[0];
    string productions = rule.substr(3);
    vector<string> splitProductions = split(productions, '/');
    grammar[nonTerminal] = splitProductions;
  }
  for (auto &rule : grammar) {
    firstSet[rule.first] = set<char>();
    followSet[rule.first] = set<char>();
  }
  computeFirst(grammar, firstSet);
  computeFollow(grammar, firstSet, followSet);
```

```
    cout << "FIRST sets:" << endl;
    for (auto &entry : firstSet) {
        cout << "FIRST(" << entry.first << ") = { ";
        for (char ch : entry.second) {
            cout << ch << " ";
        }
        cout << "}" << endl;
    }
    cout << "FOLLOW sets:" << endl;
    for (auto &entry : followSet) {
        cout << "FOLLOW(" << entry.first << ") = { ";
        for (char ch : entry.second) {
            cout << ch << " ";
        }
        cout << "}" << endl;
    }
    return 0;
}
```

Output:

```
Enter number of production rules: 4
Enter production rules (e.g., S->ACB/CbB/Ba):
S->ACB/CbB/Ba
A->da/BC
B->g/n
C->h/n
FIRST sets:
FIRST(A) = { d g h n }
FIRST(B) = { g n }
FIRST(C) = { h n }
FIRST(S) = { a b d g h n }
FOLLOW sets:
FOLLOW(A) = { $ g h }
FOLLOW(B) = { $ a g h }
FOLLOW(C) = { $ b g h }
FOLLOW(S) = { $ }


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment:7

Aim: Compute the LL1 parser for any of the given string

Date Of Experiment: 28 January 2025

Language Used: C++

Program:

```
#include <iostream>
#include <map>
#include <set>
#include <vector>
#include <sstream>
#include <algorithm>
using namespace std;
map<string, vector<vector<string>>> grammar;
map<string, set<string>> firstSets;
map<string, set<string>> followSets;
string startSymbol;
set<string> first(vector<string> production);
set<string> first(string symbol) {
  if (!isupper(symbol[0]) || symbol == "ε") {
    return {symbol};
  }
  if (firstSets.find(symbol) != firstSets.end()) {
    return firstSets[symbol];
  }

  set<string> result;
  for (auto prod : grammar[symbol]) {
    auto tempFirst = first(prod);
    result.insert(tempFirst.begin(), tempFirst.end());
```

```
    }
    firstSets[symbol] = result;
    return result;
}
set<string> first(vector<string> production) {
    set<string> result;
    bool epsilonInAll = true;
    for (auto sym : production) {
        auto tempFirst = first(sym);
        result.insert(tempFirst.begin(), tempFirst.end());
        if (tempFirst.find("ε") == tempFirst.end()) {
            epsilonInAll = false;
            break;
        }
    }
    if (!epsilonInAll) {
        result.erase("ε");
    }
    return result;
}
void follow(string symbol) {
    if (followSets.find(symbol) == followSets.end()) {
        followSets[symbol] = {};
    }
    if (symbol == startSymbol) {
        followSets[symbol].insert("$");
    }
    for (auto &rule : grammar) {
        string lhs = rule.first;
```

```
        for (auto &prod : rule.second) {
          for (size_t i = 0; i < prod.size(); ++i) {
            if (prod[i] == symbol) {
              if (i + 1 < prod.size()) {
                auto nextFirst = first(prod[i + 1]);
                for (auto t : nextFirst) {
                  if (t != "ε") followSets[symbol].insert(t);
                }
                if (nextFirst.find("ε") != nextFirst.end()) {
                  if (lhs != symbol) {
                    follow(lhs);
                    followSets[symbol].insert(followSets[lhs].begin(), followSets[lhs].end());
                  }
                }
              } else if (lhs != symbol) {
                follow(lhs);
                followSets[symbol].insert(followSets[lhs].begin(), followSets[lhs].end());
              }
            }
          }
        }
      }
    }
    bool hasLeftRecursion() {
      for (auto &rule : grammar) {
        string lhs = rule.first;
        for (auto &prod : rule.second) {
          if (prod[0] == lhs) return true;
        }
```

```
  }
  return false;
}
bool isLL1() {
  // Compute FIRST sets
  for (auto &rule : grammar) {
    first(rule.first);
  }
  // Compute FOLLOW sets
  for (auto &rule : grammar) {
    follow(rule.first);
  }
  // Check FIRST/FIRST and FIRST/FOLLOW conflict
  for (auto &rule : grammar) {
    vector<set<string>> localFirsts;
    for (auto &prod : rule.second) {
      auto prodFirst = first(prod);
      for (auto &s : localFirsts) {
        set<string> intersection;
        for (auto &t : s) {
          if (prodFirst.find(t) != prodFirst.end()) {
            intersection.insert(t);
          }
        }
        if (!intersection.empty()) return false;
      }
      localFirsts.push_back(prodFirst);
    }
    for (auto &prodFirst : localFirsts) {
```

```
                if (prodFirst.find("ε") != prodFirst.end()) {
                    for (auto &t : followSets[rule.first]) {
                        if (prodFirst.find(t) != prodFirst.end()) {
                            return false;
                        }
                    }
                }
            }
        }
        return !hasLeftRecursion();
    }
int main() {
    int n;
    cout << "Enter number of productions: ";
    cin >> n;
    cin.ignore();
    cout << "Enter productions (e.g., S -> A a | b):" << endl;
    for (int i = 0; i < n; ++i) {
        string line;
        getline(cin, line);
        string lhs = line.substr(0, line.find("->") - 1);
        lhs.erase(remove(lhs.begin(), lhs.end(), ' '), lhs.end());
        if (i == 0) startSymbol = lhs;
        string rhs = line.substr(line.find("->") + 2);
        stringstream ss(rhs);
        string token;
        while (getline(ss, token, '|')) {
            stringstream ss2(token);
            string sym;
```

```
        vector<string> prod;
        while (ss2 >> sym) prod.push_back(sym);
        grammar[lhs].push_back(prod);
      }
    }
    if (isLL1())
      cout << "The grammar is LL(1)." << endl;
    else
      cout << "The grammar is NOT LL(1)." << endl;
    return 0;
}
```

Output:

```
Enter number of productions: 3
Enter productions (e.g., S -> A a | b):
S->iEtSQ
Q->eS|n
E->b
The grammar is LL(1).
```

```
Enter number of productions: 2
Enter productions (e.g., S -> A a | b):
S->Aa|b
A->n|b
The grammar is NOT LL(1).
```

Experiment 8

Aim: Compute the SLR1 parser for any of the given string.

Date Of Experiment: 04 February 2025

Language Used: C++

Program:

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <sstream>
#include <algorithm>
#include <queue>
using namespace std;
struct Item {
  string lhs;
  vector<string> rhs;
  int dotPos;
  bool operator<(const Item &other) const {
    if (lhs != other.lhs)
      return lhs < other.lhs;
    if (rhs != other.rhs)
      return rhs < other.rhs;
    return dotPos < other.dotPos;
  }
  bool operator==(const Item &other) const {
    return lhs == other.lhs && rhs == other.rhs && dotPos == other.dotPos;
  }
};
map<string, vector<vector<string>>> grammar;
```

```
map<string, set<string>> followSet;
vector<set<Item>> states;
map<pair<int, string>, int> transitions;
map<pair<int, string>, string> ACTION;
string startSymbol;
set<string> terminals, nonTerminals;
set<Item> closure(set<Item> items) {
  queue<Item> q;
  for (auto item : items) q.push(item);
  set<Item> closureSet = items;
  while (!q.empty()) {
    Item item = q.front(); q.pop();
    if (item.dotPos >= item.rhs.size()) continue;
    string symbol = item.rhs[item.dotPos];
    if (isupper(symbol[0])) {
      for (auto prod : grammar[symbol]) {
        Item newItem = {symbol, prod, 0};
        if (closureSet.find(newItem) == closureSet.end()) {
          closureSet.insert(newItem);
          q.push(newItem);
        }
      }
    }
  }
  return closureSet;
}
set<Item> GOTO(set<Item> items, string symbol) {
  set<Item> movedItems;
  for (auto item : items) {
```

```
        if (item.dotPos < item.rhs.size() && item.rhs[item.dotPos] == symbol) {
            movedItems.insert({item.lhs, item.rhs, item.dotPos + 1});
        }
    }
    return closure(movedItems);
}
void computeFollow() {
    for (auto &g : grammar) {
        followSet[g.first] = {};
    }
    followSet[startSymbol].insert("$");
    bool changed;
    do {
        changed = false;
        for (auto &g : grammar) {
            string lhs = g.first;
            for (auto &prod : g.second) {
                for (size_t i = 0; i < prod.size(); ++i) {
                    if (isupper(prod[i][0])) {
                        if (i + 1 < prod.size()) {
                            if (!isupper(prod[i + 1][0])) {
                                if (followSet[prod[i]].insert(prod[i + 1]).second)
                                    changed = true;
                            } else {
                                // FIRST of non-terminal (no epsilon assumed)
                                if (followSet[prod[i]].insert(prod[i + 1]).second)
                                    changed = true;
                            }
                        } else {
```

```
                        for (auto f : followSet[lhs]) {
                            if (followSet[prod[i]].insert(f).second)
                                changed = true;
                        }
                    }
                }
            }
        }
    }
  } while (changed);
}
void buildCanonicalCollection() {
    Item startItem = {startSymbol + "'", {startSymbol}, 0};
    grammar[startSymbol + "'"] = {{startSymbol}};
    auto firstState = closure({startItem});
    states.push_back(firstState);
    queue<int> q;
    q.push(0);
    while (!q.empty()) {
        int stateIdx = q.front(); q.pop();
        auto &state = states[stateIdx];
        set<string> allSymbols = terminals;
        allSymbols.insert(nonTerminals.begin(), nonTerminals.end());
        for (auto sym : allSymbols) {
            auto gotoSet = GOTO(state, sym);
            if (!gotoSet.empty()) {
                auto it = find(states.begin(), states.end(), gotoSet);
                if (it == states.end()) {
                    states.push_back(gotoSet);
```

```
                int newStateIdx = states.size() - 1;
                transitions[{stateIdx, sym}] = newStateIdx;
                q.push(newStateIdx);
            } else {
                transitions[{stateIdx, sym}] = it - states.begin();
            }
        }
    }
  }
}
bool isSLR1() {
  buildCanonicalCollection();
  computeFollow();
  bool conflict = false;
  for (size_t i = 0; i < states.size(); ++i) {
    for (auto item : states[i]) {
      // Shift
      if (item.dotPos < item.rhs.size()) {
        string symbol = item.rhs[item.dotPos];
        if (terminals.find(symbol) != terminals.end()) {
          ACTION[{i, symbol}] = "shift";
        }
      }
      // Reduce
      else {
        if (item.lhs == startSymbol + "") {
          ACTION[{i, "$"}] = "accept";
        } else {
          for (auto f : followSet[item.lhs]) {
```

```
                if (ACTION.find({i, f}) != ACTION.end()) {
                    conflict = true; // Reduce/Shift or Reduce/Reduce
                } else {
                    ACTION[{i, f}] = "reduce";
                }
            }
        }
    }
  }
 }
 return !conflict;
}
int main() {
 int n;
 cout << "Enter number of productions: ";
 cin >> n;
 cin.ignore();
 cout << "Enter productions (e.g., S -> A a):" << endl;
 for (int i = 0; i < n; ++i) {
    string line;
    getline(cin, line);
    string lhs = line.substr(0, line.find("->") - 1);
    lhs.erase(remove(lhs.begin(), lhs.end(), ' '), lhs.end());
    if (i == 0) startSymbol = lhs;
    nonTerminals.insert(lhs);
    string rhs = line.substr(line.find("->") + 2);
    stringstream ss(rhs);
    string token;
    vector<string> prod;
```

```
      while (ss >> token) {
        prod.push_back(token);
        if (!isupper(token[0]) && token != "ε") terminals.insert(token);
        else if (isupper(token[0])) nonTerminals.insert(token);
      }
      grammar[lhs].push_back(prod);
    }
    terminals.insert("$");

    if (isSLR1())
      cout << "The grammar is SLR(1)." << endl;
    else
      cout << "The grammar is NOT SLR(1)." << endl;
    return 0;
}
```

Output:

```
Enter number of productions: 4
Enter productions (e.g., S -> A a):
S->E
E->E+T|T
T->T*F|F
F->(E)|id
The grammar is SLR(1).
```

Experiment:9

Aim: Compute the CLR1 parser for any of the given string.

Date Of Experiment: 11 February 2025

Language Used: C++

Program:

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <sstream>
#include <algorithm>
#include <queue>

using namespace std;

struct LR1Item {
  string lhs;
  vector<string> rhs;
  int dotPos;
  string lookahead;

  bool operator<(const LR1Item &other) const {
    if (lhs != other.lhs) return lhs < other.lhs;
    if (rhs != other.rhs) return rhs < other.rhs;
    if (dotPos != other.dotPos) return dotPos < other.dotPos;
    return lookahead < other.lookahead;
  }

  bool operator==(const LR1Item &other) const {
```

```
        return lhs == other.lhs && rhs == other.rhs && dotPos == other.dotPos && lookahead
== other.lookahead;
    }
};

map<string, vector<vector<string>>> grammar;
vector<set<LR1Item>> states;
map<pair<int, string>, int> transitions;
map<pair<int, string>, string> ACTION;
set<string> terminals, nonTerminals;
string startSymbol;

set<LR1Item> closure(set<LR1Item> items) {
    queue<LR1Item> q;
    for (auto item : items) q.push(item);
    set<LR1Item> closureSet = items;

    while (!q.empty()) {
        LR1Item item = q.front(); q.pop();
        if (item.dotPos >= item.rhs.size()) continue;

        string B = item.rhs[item.dotPos];
        if (isupper(B[0])) {
            vector<string> beta;
            for (size_t i = item.dotPos + 1; i < item.rhs.size(); ++i) {
                beta.push_back(item.rhs[i]);
            }
            beta.push_back(item.lookahead);

            set<string> lookaheads;
```

```
        if (!beta.empty()) {
          // Take first of beta (simplified here, assuming no epsilon)
          if (isupper(beta[0][0]))
            lookaheads.insert(beta[0]);
          else
            lookaheads.insert(beta[0]);
        }

        for (auto prod : grammar[B]) {
          for (auto la : lookaheads) {
            LR1Item newItem = {B, prod, 0, la};
            if (closureSet.find(newItem) == closureSet.end()) {
              closureSet.insert(newItem);
              q.push(newItem);
            }
          }
        }
      }
    }
    return closureSet;
}

set<LR1Item> GOTO(set<LR1Item> items, string symbol) {
    set<LR1Item> movedItems;
    for (auto item : items) {
      if (item.dotPos < item.rhs.size() && item.rhs[item.dotPos] == symbol) {
        movedItems.insert({item.lhs, item.rhs, item.dotPos + 1, item.lookahead});
      }
    }
```

```
    return closure(movedItems);
}

void buildCanonicalCollection() {
    LR1Item startItem = {startSymbol + "'", {startSymbol}, 0, "$"};
    grammar[startSymbol + "'"] = {{startSymbol}};
    auto firstState = closure({startItem});
    states.push_back(firstState);

    queue<int> q;
    q.push(0);

    while (!q.empty()) {
        int stateIdx = q.front(); q.pop();
        auto &state = states[stateIdx];

        set<string> allSymbols = terminals;
        allSymbols.insert(nonTerminals.begin(), nonTerminals.end());

        for (auto sym : allSymbols) {
            auto gotoSet = GOTO(state, sym);
            if (!gotoSet.empty()) {
                auto it = find(states.begin(), states.end(), gotoSet);
                if (it == states.end()) {
                    states.push_back(gotoSet);
                    int newStateIdx = states.size() - 1;
                    transitions[{stateIdx, sym}] = newStateIdx;
                    q.push(newStateIdx);
                } else {
```

```
                transitions[{stateIdx, sym}] = it - states.begin();
            }
        }
    }
  }
}

bool isCLR1() {
  buildCanonicalCollection();
  bool conflict = false;

  for (size_t i = 0; i < states.size(); ++i) {
    for (auto item : states[i]) {
      // Shift
      if (item.dotPos < item.rhs.size()) {
        string symbol = item.rhs[item.dotPos];
        if (terminals.find(symbol) != terminals.end()) {
          if (ACTION.find({i, symbol}) != ACTION.end())
            conflict = true;
          ACTION[{i, symbol}] = "shift";
        }
      }
      // Reduce
      else {
        if (item.lhs == startSymbol + "'") {
          ACTION[{i, "$"}] = "accept";
        } else {
          if (ACTION.find({i, item.lookahead}) != ACTION.end())
            conflict = true;
```

```
                ACTION[{i, item.lookahead}] = "reduce";
            }
        }
    }
  }
  return !conflict;
}

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

  cout << "Enter productions (e.g., S -> A a):" << endl;
  for (int i = 0; i < n; ++i) {
    string line;
    getline(cin, line);
    string lhs = line.substr(0, line.find("->") - 1);
    lhs.erase(remove(lhs.begin(), lhs.end(), ' '), lhs.end());
    if (i == 0) startSymbol = lhs;
    nonTerminals.insert(lhs);

    string rhs = line.substr(line.find("->") + 2);
    stringstream ss(rhs);
    string token;
    vector<string> prod;
    while (ss >> token) {
      prod.push_back(token);
```

```
            if (!isupper(token[0]) && token != "ε") terminals.insert(token);
            else if (isupper(token[0])) nonTerminals.insert(token);
        }
        grammar[lhs].push_back(prod);
    }
    terminals.insert("$");

    if (isCLR1())
        cout << "The grammar is CLR(1)." << endl;
    else
        cout << "The grammar is NOT CLR(1)." << endl;
    return 0;
}
```

Output:

```
Enter number of productions: 3
Enter productions (e.g., S -> A a):
S->Aa/bAc/Bc/bBa
A->d
B->d
The grammar is CLR(1).
```

Experiment 10

Aim: To generate Three-Address Code (TAC) representations using Quadruples, Triples, and Indirect Triples.

Date Of Experiment: 18 February 2025

Language Used: C++

Program:

```
#include <iostream>
#include <vector>
#include <stack>
#include <cctype>
using namespace std;
struct Quadruple {
   string op;
   string arg1;
   string arg2;
   string result;
};
struct Triple {
   string op;
   string arg1;
   string arg2;
};
vector<Quadruple> quadruples;
vector<Triple> triples;
vector<int> indirectTriples;
int tempCount = 0;
string newTemp() {
   return "t" + to_string(tempCount++);
}
```

```
int precedence(char op) {
    if (op == '*' || op == '/') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;
}
void generateTAC(string expr) {
    stack<string> operand;
    stack<char> op;
    for (int i = 0; i < expr.size(); i++) {
        if (isspace(expr[i])) continue;
        if (isalnum(expr[i])) {
            string val(1, expr[i]);
            operand.push(val);
        }
        else if (expr[i] == '(') {
            op.push('(');
        }
        else if (expr[i] == ')') {
            while (!op.empty() && op.top() != '(') {
                string t = newTemp();
                string op2 = operand.top(); operand.pop();
                string op1 = operand.top(); operand.pop();
                char oper = op.top(); op.pop();
                operand.push(t);
                quadruples.push_back({string(1, oper), op1, op2, t});
                triples.push_back({string(1, oper), op1, op2});
            }
            op.pop(); // remove '('
        }
```

```
        else { // operator
            while (!op.empty() && precedence(op.top()) >= precedence(expr[i])) {
                string t = newTemp();
                string op2 = operand.top(); operand.pop();
                string op1 = operand.top(); operand.pop();
                char oper = op.top(); op.pop();
                operand.push(t);
                quadruples.push_back({string(1, oper), op1, op2, t});
                triples.push_back({string(1, oper), op1, op2});
            }
            op.push(expr[i]);
        }
    }
    while (!op.empty()) {
        string t = newTemp();
        string op2 = operand.top(); operand.pop();
        string op1 = operand.top(); operand.pop();
        char oper = op.top(); op.pop();
        operand.push(t);
        quadruples.push_back({string(1, oper), op1, op2, t});
        triples.push_back({string(1, oper), op1, op2});
    }
}
void printQuadruples() {
    cout << "\nQuadruples:\n";
    cout << "Op\tArg1\tArg2\tResult\n";
    for (auto q : quadruples) {
        cout << q.op << "\t" << q.arg1 << "\t" << q.arg2 << "\t" << q.result << "\n";
    }
```

```
}
void printTriples() {
   cout << "\nTriples:\n";
   cout << "Index\tOp\tArg1\tArg2\n";
   for (int i = 0; i < triples.size(); i++) {
      cout << i << "\t" << triples[i].op << "\t" << triples[i].arg1 << "\t" << triples[i].arg2 <<
"\n";
   }
}
void printIndirectTriples() {
   cout << "\nIndirect Triples:\n";
   cout << "Ptr\tInstruction\n";
   for (int i = 0; i < triples.size(); i++) {
      indirectTriples.push_back(i);
   }
   for (int i = 0; i < indirectTriples.size(); i++) {
      cout << i << "\t" << " (" << triples[i].op << ", " << triples[i].arg1 << ", " << triples[i].arg2
<< ")\n";
   }
}
int main() {
   string expr;
   cout << "Enter an arithmetic expression (e.g., a+b*c): ";
   cin >> expr;
   generateTAC(expr);
   printQuadruples();
   printTriples();
   printIndirectTriples();
   return 0;
}
```

Output:

```
Enter an arithmetic expression (e.g., a+b*c): a+b*c

Quadruples:
Op      Arg1    Arg2    Result
*       b       c       t0
+       a       t0      t1

Triples:
Index   Op      Arg1    Arg2
0       *       b       c
1       +       a       t0

Indirect Triples:
Ptr     Instruction
0        (*, b, c)
1        (+, a, t0)
```