Compiler Assignment – 3

23BCE1344 | Aditya

Experiment-7(a)

Aim:

To Construct Predictive parse table using C language.

Hint: Consider the input grammar without left recursion, find FIRST and FOLLOW for each non-terminal and then construct the parse table.

Algorithm:

1. Input the grammar
   a. Read the terminals
   b. Read the non terminals
   c. Set the first non terminal as the start symbol
   d. Read the productions
2. Compute the First sets
   a. For each production left hand side and right hand side
   b. If the first symbol of right side is a terminal or epsilon add it to the First set
   c. If it is a non terminal add its First set excluding epsilon
   d. If epsilon appears in all symbols add epsilon to the First set
   e. Repeat until no changes
3. Compute the Follow sets
   a. Add dollar to the Follow set of the start symbol
   b. For each non terminal in the productions
   c. Look at symbols after it
   d. If next is a terminal add it to the Follow set
   e. If next is a non terminal add its First set excluding epsilon
   f. If all next symbols derive epsilon or if it is the last symbol add Follow of the left hand side
   g. Repeat until no changes
4. Construct the Parse Table
   a. For each production compute the First of right side

b. For every terminal in this First set add the production to the table
c. If epsilon is in First then for each symbol in Follow of the left side add the production

5. Display the results
   a. Show the grammar
   b. Show the First sets
   c. Show the Follow sets
   d. Show the Parse Table

Code:

```
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <map>
#include <set>
#include <string>
#include <vector>

using namespace std;

class PredictiveParser {
private:
  set<char> terminals;
  set<char> non_terminals;
  vector<pair<char, string>> productions;
  map<char, set<char>> first_sets;
  map<char, set<char>> follow_sets;
  map<pair<char, char>, string> parse_table;

  char start_symbol;

public:
  void inputGrammar() {
    int n_terminals, n_non_terminals, n_productions;

    // Input terminals
    cout << "Enter number of terminals: ";
    cin >> n_terminals;
    cout << "Enter terminals (single characters, space separated): ";
    for (int i = 0; i < n_terminals; i++) {
      char t;
      cin >> t;
      terminals.insert(t);
    }

    // Input non-terminals
```

```
  cout << "Enter number of non-terminals: ";
  cin >> n_non_terminals;
  cout << "Enter non-terminals (single characters, space separated): ";
  for (int i = 0; i < n_non_terminals; i++) {
    char nt;
    cin >> nt;
    non_terminals.insert(nt);
  }

  // Set start symbol as first non-terminal
  start_symbol = *non_terminals.begin();

  // Input productions
  cout << "Enter number of productions: ";
  cin >> n_productions;
  cout << "Enter productions in format 'A->alpha' (use 'e' for epsilon):\n";
  cin.ignore(); // Clear buffer

  for (int i = 0; i < n_productions; i++) {
    string production;
    cout << "Production " << (i + 1) << ": ";
    getline(cin, production);

    // Parse production A->alpha
    char lhs = production[0];
    string rhs = production.substr(3); // Skip 'A->'

    productions.push_back({lhs, rhs});
  }

  cout << "\nGrammar input complete!\n";
  displayGrammar();
}

void displayGrammar() {
  cout << "Terminals: { ";
  for (char t : terminals)
    cout << t << " ";
  cout << "}\n";

  cout << "Non-terminals: { ";
  for (char nt : non_terminals)
    cout << nt << " ";
  cout << "}\n";

  cout << "Start Symbol: " << start_symbol << "\n";
```

```
  cout << "Productions:\n";
  for (auto &prod : productions) {
    cout << "  " << prod.first << " -> " << prod.second << "\n";
  }
}

bool isTerminal(char c) { return terminals.find(c) != terminals.end(); }

bool isNonTerminal(char c) {
  return non_terminals.find(c) != non_terminals.end();
}

bool isEpsilon(char c) { return c == 'e'; }

void computeFirst() {

  bool changed = true;
  while (changed) {
    changed = false;

    for (auto &prod : productions) {
      char lhs = prod.first;
      string rhs = prod.second;

      set<char> old_first = first_sets[lhs];

      // Compute FIRST for this production
      for (int i = 0; i < rhs.length(); i++) {
        char symbol = rhs[i];

        if (isTerminal(symbol) || isEpsilon(symbol)) {
          first_sets[lhs].insert(symbol);
          break;
        } else if (isNonTerminal(symbol)) {
          // Add FIRST(symbol) - {epsilon} to FIRST(lhs)
          for (char c : first_sets[symbol]) {
            if (!isEpsilon(c)) {
              first_sets[lhs].insert(c);
            }
          }

          // If epsilon not in FIRST(symbol), break
          if (first_sets[symbol].find('e') == first_sets[symbol].end()) {
            break;
          }
```

```
        // If we've processed all symbols and all had epsilon
        if (i == rhs.length() - 1) {
          first_sets[lhs].insert('e');
        }
       }
      }

      if (first_sets[lhs] != old_first) {
       changed = true;
      }
     }
    }

    displayFirstSets();
}

void computeFollow() {

    // Add $ to FOLLOW of start symbol
    follow_sets[start_symbol].insert('$');

    bool changed = true;
    while (changed) {
     changed = false;

     for (auto &prod : productions) {
      char lhs = prod.first;
      string rhs = prod.second;

      for (int i = 0; i < rhs.length(); i++) {
       char symbol = rhs[i];

       if (isNonTerminal(symbol)) {
        set<char> old_follow = follow_sets[symbol];

        // Look at symbols after current symbol
        bool all_epsilon = true;
        for (int j = i + 1; j < rhs.length(); j++) {
         char next_symbol = rhs[j];

         if (isTerminal(next_symbol)) {
          follow_sets[symbol].insert(next_symbol);
          all_epsilon = false;
          break;
         } else if (isNonTerminal(next_symbol)) {
```

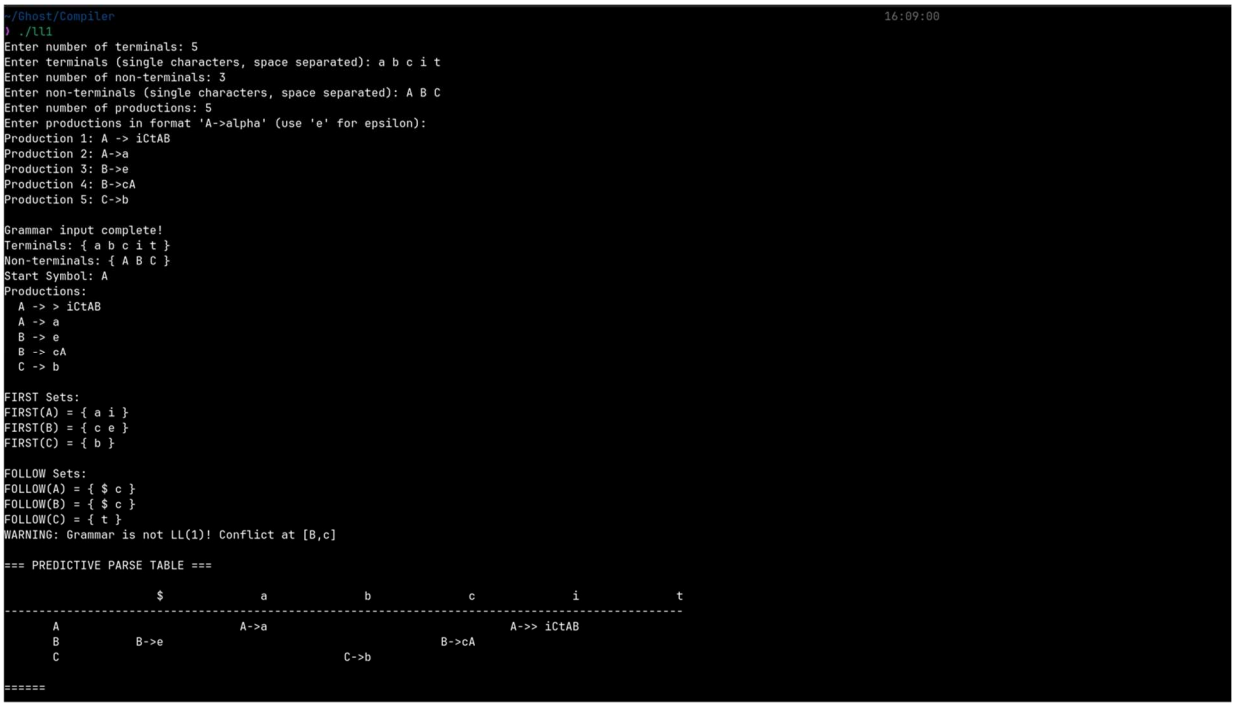
```
            // Add FIRST(next_symbol) - {epsilon} to FOLLOW(symbol)
            for (char c : first_sets[next_symbol]) {
              if (!isEpsilon(c)) {
                follow_sets[symbol].insert(c);
              }
            }

            // If epsilon not in FIRST(next_symbol), break
            if (first_sets[next_symbol].find('e') ==
                first_sets[next_symbol].end()) {
              all_epsilon = false;
              break;
            }
          }
        }

        // If all symbols after have epsilon or no symbols after
        if (all_epsilon) {
          // Add FOLLOW(lhs) to FOLLOW(symbol)
          for (char c : follow_sets[lhs]) {
            follow_sets[symbol].insert(c);
          }
        }

        if (follow_sets[symbol] != old_follow) {
          changed = true;
        }
      }
    }
  }

  displayFollowSets();
}

void displayFirstSets() {
  cout << "\nFIRST Sets:\n";
  for (char nt : non_terminals) {
    cout << "FIRST(" << nt << ") = { ";
    for (char c : first_sets[nt]) {
      cout << c << " ";
    }
    cout << "}\n";
  }
}
```

```
void displayFollowSets() {
  cout << "\nFOLLOW Sets:\n";
  for (char nt : non_terminals) {
    cout << "FOLLOW(" << nt << ") = { ";
    for (char c : follow_sets[nt]) {
      cout << c << " ";
    }
    cout << "}\n";
  }
}

void constructParseTable() {

  for (int i = 0; i < productions.size(); i++) {
    char lhs = productions[i].first;
    string rhs = productions[i].second;

    // Compute FIRST of RHS
    set<char> first_rhs;
    bool epsilon_in_first = true;

    for (int j = 0; j < rhs.length(); j++) {
      char symbol = rhs[j];

      if (isTerminal(symbol) || isEpsilon(symbol)) {
        first_rhs.insert(symbol);
        if (!isEpsilon(symbol)) {
          epsilon_in_first = false;
        }
        break;
      } else if (isNonTerminal(symbol)) {
        for (char c : first_sets[symbol]) {
          first_rhs.insert(c);
        }

        if (first_sets[symbol].find('e') == first_sets[symbol].end()) {
          epsilon_in_first = false;
          break;
        }
      }
    }

    // Add entries to parse table for each terminal in FIRST(RHS)
    for (char terminal : first_rhs) {
      if (!isEpsilon(terminal)) {
        if (parse_table.find({lhs, terminal}) != parse_table.end()) {
```

```
          cout << "WARNING: Grammar is not LL(1)! Conflict at [" << lhs << ","
               << terminal << "]\n";
        }
        parse_table[{lhs, terminal}] = rhs;
      }
    }

    // If epsilon in FIRST(RHS), add entries for FOLLOW(LHS)
    if (first_rhs.find('e') != first_rhs.end()) {
      for (char terminal : follow_sets[lhs]) {
        if (parse_table.find({lhs, terminal}) != parse_table.end()) {
          cout << "WARNING: Grammar is not LL(1)! Conflict at [" << lhs << ","
               << terminal << "]\n";
        }
        parse_table[{lhs, terminal}] = rhs;
      }
    }
  }

  displayParseTable();
}

void displayParseTable() {
  cout << "\n=== PREDICTIVE PARSE TABLE ===\n\n";

  // Create sorted vectors for consistent display
  vector<char> sorted_terminals(terminals.begin(), terminals.end());
  vector<char> sorted_non_terminals(non_terminals.begin(),
                                    non_terminals.end());
  sorted_terminals.push_back('$'); // Add $ to terminals for display
  sort(sorted_terminals.begin(), sorted_terminals.end());
  sort(sorted_non_terminals.begin(), sorted_non_terminals.end());

  // Print header
  cout << setw(8) << " ";
  for (char t : sorted_terminals) {
    cout << setw(15) << t;
  }
  cout << "\n";

  cout << setfill('-') << setw(8 + 15 * sorted_terminals.size()) << ""
       << setfill(' ') << "\n";

  // Print table rows
  for (char nt : sorted_non_terminals) {
    cout << setw(8) << nt;
```

```
      for (char t : sorted_terminals) {
        auto key = make_pair(nt, t);
        if (parse_table.find(key) != parse_table.end()) {
          string entry = string(1, nt) + "->" + parse_table[key];
          cout << setw(15) << entry;
        } else {
          cout << setw(15) << " ";
        }
      }
      cout << "\n";
    }
  }

  void run() {
    inputGrammar();
    computeFirst();
    computeFollow();
    constructParseTable();

    cout << "\n======\n";
  }
};

int main() {
  PredictiveParser parser;
  parser.run();
  return 0;
}
```

Output:


~/Ghost/Compiler 16:09:00
) ./ll1
Enter number of terminals: 5
Enter terminals (single characters, space separated): a b c i t
Enter number of non-terminals: 3
Enter non-terminals (single characters, space separated): A B C
Enter number of productions: 5
Enter productions in format 'A->alpha' (use 'e' for epsilon):
Production 1: A -> iCtAB
Production 2: A->a
Production 3: B->e
Production 4: B->cA
Production 5: C->b
Grammar input complete!
Terminals: { a b c i t }
Non-terminals: { A B C }
Start Symbol: A
Productions:
A -> > iCtAB
A -> a
B -> e
B -> cA
C -> b
FIRST Sets:
FIRST(A) = { a i }
FIRST(B) = { c e }
FIRST(C) = { b }
FOLLOW Sets:
FOLLOW(A) = { $ c }
FOLLOW(B) = { $ c }
FOLLOW(C) = { t }
WARNING: Grammar is not LL(1)! Conflict at [B,c]
=== PREDICTIVE PARSE TABLE ===
$ a b c i t
A A->a A->> iCtAB
B B->e B->cA
C C->b
======

Experiment-7(b)

Aim:

To Implement the Predictive parsing algorithm, get parse table and input string are inputs. Use C language for implementation.

Algorithm:

1. Input the grammar information
   a. Read the list of terminals including dollar
   b. Read the list of non terminals
   c. Set the first non terminal as the start symbol
2. Input the parse table
   a. For each non empty entry provide row non terminal column terminal and production
   b. Store the right side of production in the parse table
3. Display the parse table
4. Parsing process for a given input string
   a. Initialize a stack with dollar at the bottom and start symbol on top
   b. Append dollar at the end of the input string
   c. Repeat until stack is empty or input ends
   i. If the top of stack is a terminal and it matches current input symbol then pop the stack and advance the input
   ii. If the top of stack is a terminal and it does not match then report error and stop
   iii. If the top of stack is a non terminal then check the parse table entry for this non terminal and current input symbol
   - If entry exists replace the non terminal on stack with the right side of production in reverse order
   - If entry does not exist report error and stop
   iv. If top of stack is epsilon remove it and continue
   d. If both stack and input are fully consumed accept the string
   e. Otherwise reject the string
5. Repeat the parsing process for multiple input strings until the user exits
6. End

Code:

```
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <map>
#include <sstream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

class PredictiveParsingAlgorithm {
private:
  vector<char> terminals;
  vector<char> non_terminals;
  map<pair<char, char>, string> parse_table;
  char start_symbol;

public:
  void inputParseTable() {
    int n_terminals, n_non_terminals;

    cout << "\n=== PREDICTIVE PARSING ALGORITHM ===\n\n";

    // Input terminals
    cout << "Enter number of terminals (including $): ";
    cin >> n_terminals;
    cout << "Enter terminals (single characters, space separated): ";
    for (int i = 0; i < n_terminals; i++) {
      char t;
      cin >> t;
      terminals.push_back(t);
    }

    // Input non-terminals
    cout << "Enter number of non-terminals: ";
    cin >> n_non_terminals;
    cout << "Enter non-terminals (single characters, space separated): ";
    for (int i = 0; i < n_non_terminals; i++) {
      char nt;
      cin >> nt;
      non_terminals.push_back(nt);
    }

    // Set start symbol
```

```
  start_symbol = non_terminals[0];
  cout << "Start symbol set to: " << start_symbol << "\n\n";

  // Input parse table entries
  cout << "=== PARSE TABLE INPUT ===\n";
  cout << "Enter only NON-EMPTY parse table entries. Format: 'row col "
          "production'\n";
  cout << "Use 'e' for epsilon productions, 'END' when done.\n";
  cout << "All unspecified entries will be treated as EMPTY/ERROR entries.\n";
  cout << "Example: 'E a E->TF' or 'T * T->*FT'\n\n";

  cin.ignore(); // Clear buffer
  string input_line;

  while (true) {
    cout << "Enter table entry (or 'END' to finish): ";
    getline(cin, input_line);

    if (input_line == "END" || input_line == "end") {
      break;
    }

    // Parse input: row col production
    stringstream ss(input_line);
    char row, col;
    string production;

    ss >> row >> col >> production;

    // Extract RHS from production (skip LHS->)
    size_t arrow_pos = production.find("->");
    if (arrow_pos != string::npos) {
      string rhs = production.substr(arrow_pos + 2);
      parse_table[{row, col}] = rhs;
    } else {
      cout << "Invalid format! Use format: 'A a A->alpha'\n";
    }
  }

  displayParseTable();
}

void displayParseTable() {
  cout << "\n=== PARSE TABLE ===\n\n";

  // Print header
```

```
  cout << setw(8) << " ";
  for (char t : terminals) {
    cout << setw(15) << t;
  }
  cout << "\n";

  cout << setfill('-') << setw(8 + 15 * terminals.size()) << ""
       << setfill(' ') << "\n";

  // Print table rows
  for (char nt : non_terminals) {
    cout << setw(8) << nt;

    for (char t : terminals) {
      auto key = make_pair(nt, t);
      if (parse_table.find(key) != parse_table.end()) {
        string entry = string(1, nt) + "->" + parse_table[key];
        cout << setw(15) << entry;
      } else {
        cout << setw(15) << "ERROR";
      }
    }
    cout << "\n";
  }
  cout << "\n";
}

bool isTerminal(char c) {
  return find(terminals.begin(), terminals.end(), c) != terminals.end();
}

bool isNonTerminal(char c) {
  return find(non_terminals.begin(), non_terminals.end(), c) !=
         non_terminals.end();
}

string stackToString(stack<char> st) {
  string result = "";
  vector<char> temp;

  while (!st.empty()) {
    temp.push_back(st.top());
    st.pop();
  }

  for (int i = temp.size() - 1; i >= 0; i--) {
```

```
    result += temp[i];
  }

  return result;
}

void parseString(string input) {
  cout << "\n=== PARSING STRING: \"" << input << "\" ===\n\n";

  // Initialize stack with $ and start symbol
  stack<char> parsing_stack;
  parsing_stack.push('$');
  parsing_stack.push(start_symbol);

  // Add $ to end of input
  input += '$';
  int input_pointer = 0;

  int step = 1;
  bool accepted = false;

  cout << setw(6) << "Step" << setw(15) << "Stack" << setw(15) << "Input"
       << setw(20) << "Action" << setw(25) << "Production Used" << "\n";
  cout << setfill('-') << setw(85) << "" << setfill(' ') << "\n";

  while (!parsing_stack.empty() && input_pointer < input.length()) {
    char stack_top = parsing_stack.top();
    char current_input = input[input_pointer];

    string stack_content = stackToString(parsing_stack);
    string remaining_input = input.substr(input_pointer);

    cout << setw(6) << step++;
    cout << setw(15) << stack_content;
    cout << setw(15) << remaining_input;

    // Case 1: Stack top is terminal
    if (isTerminal(stack_top)) {
      if (stack_top == current_input) {
        // Match found
        parsing_stack.pop();
        input_pointer++;

        cout << setw(20) << "MATCH & ADVANCE";
        cout << setw(25) << "Terminal matched: " << stack_top;
```

```
    if (stack_top == '$') {
      accepted = true;
      cout << "\n";
      break;
    }
  } else {
    // Error: terminals don't match
    cout << setw(20) << "ERROR";
    cout << setw(25) << "Terminal mismatch";
    cout << "\n\nPARSING FAILED: Terminal mismatch!\n";
    cout << "Expected: '" << stack_top << "', Found: '" << current_input
         << "'\n";
    return;
  }
}
// Case 2: Stack top is non-terminal
else if (isNonTerminal(stack_top)) {
  auto key = make_pair(stack_top, current_input);

  if (parse_table.find(key) != parse_table.end()) {
    // Valid production found
    string production = parse_table[key];
    parsing_stack.pop(); // Remove non-terminal

    cout << setw(20) << "EXPAND";
    cout << setw(25) << stack_top << "->" << production;

    // Push production in reverse order (except epsilon)
    if (production != "e") {
      for (int i = production.length() - 1; i >= 0; i--) {
        parsing_stack.push(production[i]);
      }
    }
  } else {
    // Error: no production in parse table
    cout << setw(20) << "ERROR";
    cout << setw(25) << "No production";
    cout << "\n\nPARSING FAILED: No production in parse table!\n";
    cout << "Non-terminal: '" << stack_top << "', Terminal: '"
         << current_input << "'\n";
    return;
  }
} else {
  cout << setw(20) << "ERROR";
  cout << setw(25) << "Invalid symbol";
  cout << "\n\nPARSING FAILED: Invalid symbol on stack!\n";
```

```
      return;
    }

    cout << "\n";
  }

  // Check final result
  if (accepted && parsing_stack.empty()) {
    cout << "\n*** PARSING SUCCESSFUL! STRING ACCEPTED ***\n";
  } else if (!parsing_stack.empty()) {
    cout << "\nPARSING FAILED: Stack not empty at end!\n";
    cout << "Remaining stack: " << stackToString(parsing_stack) << "\n";
  } else if (input_pointer < input.length()) {
    cout << "\nPARSING FAILED: Input not fully consumed!\n";
    cout << "Remaining input: " << input.substr(input_pointer) << "\n";
  } else {
    cout << "\nPARSING FAILED: Unexpected termination!\n";
  }
}

void parseMultipleStrings() {
  cout << "\n=== STRING PARSING ===\n";

  string input_string;
  while (true) {
    cout << "\nEnter string to parse (or 'EXIT' to quit): ";
    cin >> input_string;

    if (input_string == "EXIT" || input_string == "exit") {
      break;
    }

    parseString(input_string);

    cout << "\n" << string(60, '=') << "\n";
  }
}

void run() {
  inputParseTable();
  parseMultipleStrings();

  cout << "\n=== PREDICTIVE PARSING COMPLETE ===\n";
}
};
```

```
int main() {
  PredictiveParsingAlgorithm parser;
  parser.run();
  return 0;
}
```

Output:

```
~/Ghost/Compiler                                                                                                    16:28:05
❯ ./llistack

=== PREDICTIVE PARSING ALGORITHM ===

Enter number of terminals (including $): 6
Enter terminals (single characters, space separated): i $ + * ( )
Enter number of non-terminals: 5
Enter non-terminals (single characters, space separated): E R T Y F
Start symbol set to: E

=== PARSE TABLE INPUT ===
Enter only NON-EMPTY parse table entries. Format: 'row col production'
Use 'e' for epsilon productions, 'END' when done.
All unspecified entries will be treated as EMPTY/ERROR entries.
Example: 'E a E->TF' or 'T * T->*FT'

Enter table entry (or 'END' to finish): E i E->TR
Enter table entry (or 'END' to finish): E ( E->TR
Enter table entry (or 'END' to finish): R $ E->e
Enter table entry (or 'END' to finish): R + R->+TR
Enter table entry (or 'END' to finish): R ) E->e
Enter table entry (or 'END' to finish): T i T->FY
Enter table entry (or 'END' to finish): T ( T->FY
Enter table entry (or 'END' to finish): Y $ Y->e
Enter table entry (or 'END' to finish): Y + Y->e
Enter table entry (or 'END' to finish): Y * Y->*FY
Enter table entry (or 'END' to finish): Y ) Y->e
Enter table entry (or 'END' to finish): F i F->i
Enter table entry (or 'END' to finish): F ( F->(E)
Enter table entry (or 'END' to finish): END

=== PARSE TABLE ===

                    i             $             +             *             (             )
-------------------------------------------------------------------------------------------------
      E          E->TR          ERROR         ERROR         ERROR          E->TR         ERROR
      R          ERROR           R->e        R->+TR         ERROR          ERROR          R->e
      T          T->FY          ERROR         ERROR         ERROR          T->FY         ERROR
      Y          ERROR           Y->e          Y->e        Y->*FY          ERROR          Y->e
      F           F->i          ERROR         ERROR         ERROR         F->(E)         ERROR


=== STRING PARSING ===

Enter string to parse (or 'EXIT' to quit): i+i*i

=== PARSING STRING: "i+i*i" ===

  Step          Stack          Input              Action          Production Used
```

```
Enter table entry (or 'END' to finish): Y + Y->e
Enter table entry (or 'END' to finish): Y * Y->*FY
Enter table entry (or 'END' to finish): Y ) Y->e
Enter table entry (or 'END' to finish): F i F->i
Enter table entry (or 'END' to finish): F ( F->(E)
Enter table entry (or 'END' to finish): END

=== PARSE TABLE ===

                    i             $             +             *             (             )
-------------------------------------------------------------------------------------------------
      E          E->TR          ERROR         ERROR         ERROR          E->TR         ERROR
      R          ERROR           R->e        R->+TR         ERROR          ERROR          R->e
      T          T->FY          ERROR         ERROR         ERROR          T->FY         ERROR
      Y          ERROR           Y->e          Y->e        Y->*FY          ERROR          Y->e
      F           F->i          ERROR         ERROR         ERROR         F->(E)         ERROR


=== STRING PARSING ===

Enter string to parse (or 'EXIT' to quit): i+i*i

=== PARSING STRING: "i+i*i" ===

  Step          Stack          Input              Action          Production Used
-------------------------------------------------------------------------------------------
     1             $E         i+i*i$              EXPAND                        E->TR
     2            $RT         i+i*i$              EXPAND                        T->FY
     3           $RYF         i+i*i$              EXPAND                        F->i
     4           $RYi         i+i*i$     MATCH & ADVANCE      Terminal matched: i
     5            $RY          +i*i$              EXPAND                        Y->e
     6             $R          +i*i$              EXPAND                       R->+TR
     7           $RT+          +i*i$     MATCH & ADVANCE      Terminal matched: +
     8           $RT           i*i$              EXPAND                        T->FY
     9           $RYF          i*i$              EXPAND                        F->i
    10           $RYi          i*i$     MATCH & ADVANCE      Terminal matched: i
    11            $RY           *i$              EXPAND                       Y->*FY
    12          $RYF*           *i$     MATCH & ADVANCE      Terminal matched: *
    13           $RYF            i$              EXPAND                        F->i
    14           $RYi            i$     MATCH & ADVANCE      Terminal matched: i
    15            $RY             $              EXPAND                        Y->e
    16             $R             $              EXPAND                        R->e
    17              $             $     MATCH & ADVANCE      Terminal matched: $

*** PARSING SUCCESSFUL! STRING ACCEPTED ***

============================================================

Enter string to parse (or 'EXIT' to quit):
```

Experiment-8(a)

Aim:
To Construct precedence table for the given operator grammar.

Algorithm:

1.Input the grammar
a. Read operators which are the terminals including dollar
b. Read non terminals
c. Read the productions in the form left side arrow right side

2.Compute the Leading sets
a. For each production look at the first symbol of the right side
b. If it is an operator add it to Leading of the left side
c. If it is a non terminal then add Leading of that non terminal to the Leading of the left side
d. Repeat until no changes

3.Compute the Trailing sets
a. For each production look at the last symbol of the right side
b. If it is an operator add it to Trailing of the left side
c. If it is a non terminal then add Trailing of that non terminal to the Trailing of the left side
d. Repeat until no changes

4.Construct the precedence table
a. For each production check the right side
b. If two operators appear next to each other then put equal between them
c. If an operator is followed by a non terminal then place less than between the operator and all symbols in Leading of the non terminal
d. If a non terminal is followed by an operator then place greater than between all symbols in Trailing of the non terminal and the operator
e. If a pattern operator non terminal operator appears then place equal between the two operators

5.Display the precedence table

6.Analyze for conflicts
a. Check if any cell has multiple relations which means conflict
b. Count empty cells and report that they may cause parsing errors

7.End

Code:

```
#include <bits/stdc++.h>
using namespace std;

using Set = set<char>;

// helper: print set
string set_to_str(const Set &s) {
  string out;
  for (char c : s) {
    out.push_back(c);
    out.push_back(',');
  }
  if (!out.empty())
    out.pop_back();
  return out;
}

int idx_of(const vector<char> &v, char c) {
  for (size_t i = 0; i < v.size(); ++i)
    if (v[i] == c)
      return (int)i;
  return -1;
}

int main() {

  cout << "Enter number of productions: ";
  int n;
  if (!(cin >> n))
    return 0;
  vector<string> prod(n);
  cout << "Enter the productions (format A->aB etc). Terminals and "
          "nonterminals should be single chars.\n";
  for (int i = 0; i < n; ++i) {
    cin >> prod[i];
    // allow both A->... and A->
    // we'll assume '->' at positions 1 and 2 or at index 1 for simplicity
    // normalize to form "A->rhs"
    if (prod[i].size() >= 2 && prod[i][1] != '-') {
      // maybe format like "A=abc"? leave as-is
    }
  }

  // collect nonterminals (left sides) and terminals (symbols appearing on RHS
  // that are not nonterminals)
  vector<char> NT; // nonterminals
  vector<char> T;  // terminals

  for (int i = 0; i < n; ++i) {
    char A = prod[i][0];
```

```
  if (idx_of(NT, A) == -1)
    NT.push_back(A);
}

// find RHS characters (starting after '->' which we assume starts at index 2
// or 3).
for (int i = 0; i < n; ++i) {
  string &p = prod[i];
  // find start of RHS
  size_t rhs_start = p.find("->");
  if (rhs_start == string::npos)
    rhs_start = 2;
  else
    rhs_start += 2;
  for (size_t k = rhs_start; k < p.size(); ++k) {
    char ch = p[k];
    if (ch == ' ' || ch == '\t')
      continue;
    if (idx_of(NT, ch) == -1 && idx_of(T, ch) == -1) {
      // if not a known nonterminal, treat as terminal for now
      if (!isupper(ch)) // heuristic: uppercase = nonterminal
        T.push_back(ch);
      else {
        // uppercase may be a nonterminal if it appears on LHS earlier; check:
        if (idx_of(NT, ch) == -1) {
          // treat uppercase not in NT as terminal (or possibly missing LHS)
          T.push_back(ch);
        }
      }
    }
  }
}

// If "$" not in terminals, add it as end marker
if (idx_of(T, '$') == -1)
  T.push_back('$');

// map nonterminal -> leading / trailing sets (terminals)
unordered_map<char, Set> LEAD, TRAIL;

// initialize empty sets
for (char A : NT) {
  LEAD[A] = Set();
  TRAIL[A] = Set();
}

// Helper lambdas to extract RHS and iterate
auto rhs_of = [&](int i) -> string {
  size_t rhs_start = prod[i].find("->");
  if (rhs_start == string::npos)
    rhs_start = 2;
```

```
  else
    rhs_start += 2;
  return prod[i].substr(rhs_start);
};

// Compute LEADING
// Rule: If A -> a... then a ∈ LEADING(A) (if a is terminal)
// If A -> B... and B is nonterminal, then LEADING(B) subset of LEADING(A)
// We'll do a fixed-point iterative closure.
bool changed = true;
while (changed) {
  changed = false;
  for (int i = 0; i < n; ++i) {
    char A = prod[i][0];
    string rhs = rhs_of(i);
    if (rhs.empty())
      continue;
    // first symbol
    char x = rhs[0];
    if (idx_of(T, x) != -1) {
      if (LEAD[A].insert(x).second)
        changed = true;
    } else if (idx_of(NT, x) != -1) {
      // add all LEAD[x] to LEAD[A]
      for (char t : LEAD[x])
        if (LEAD[A].insert(t).second)
          changed = true;
      // also if there is terminal after the nonterminal: A -> B a ...,
      // collect a But the iterative approach will propagate via other rules.
    }
    // Also, if first symbol is nonterminal followed by terminal, include that
    // terminal:
    for (size_t k = 0; k + 1 < rhs.size(); ++k) {
      char c1 = rhs[k], c2 = rhs[k + 1];
      if (idx_of(NT, c1) != -1 && idx_of(T, c2) != -1) {
        if (LEAD[A].insert(c2).second)
          changed = true;
      }
    }
  }
}

// Compute TRAILING similar but using last symbol
changed = true;
while (changed) {
  changed = false;
  for (int i = 0; i < n; ++i) {
    char A = prod[i][0];
    string rhs = rhs_of(i);
    if (rhs.empty())
      continue;
```

```
    char x = rhs[rhs.size() - 1];
    if (idx_of(T, x) != -1) {
      if (TRAIL[A].insert(x).second)
        changed = true;
    } else if (idx_of(NT, x) != -1) {
      for (char t : TRAIL[x])
        if (TRAIL[A].insert(t).second)
          changed = true;
    }
    // also if terminal followed by nonterminal anywhere -> add that terminal
    for (int k = (int)rhs.size() - 2; k >= 0; --k) {
      char c1 = rhs[k], c2 = rhs[k + 1];
      if (idx_of(T, c1) != -1 && idx_of(NT, c2) != -1) {
        if (TRAIL[A].insert(c1).second)
          changed = true;
      }
    }
  }
}

// Print LEADING and TRAILING
cout << "\nLEADING sets:\n";
for (char A : NT) {
  cout << "LEADING[" << A << "] = { ";
  bool first = true;
  for (char t : LEAD[A]) {
    if (!first)
      cout << ", ";
    cout << t;
    first = false;
  }
  cout << " }\n";
}
cout << "\nTRAILING sets:\n";
for (char A : NT) {
  cout << "TRAILING[" << A << "] = { ";
  bool first = true;
  for (char t : TRAIL[A]) {
    if (!first)
      cout << ", ";
    cout << t;
    first = false;
  }
  cout << " }\n";
}

// Build operator-precedence table among terminals.
// Initialize table with ' ' (no relation). We'll use chars: '<', '>', '=', '
// '.
int Tn = (int)T.size();
vector<vector<char>> table(
```

```
    Tn, vector<char>(Tn, ' ')); // table[i][j] relation between T[i] and T[j]

// for each production, for adjacent symbols a b:
// if a and b are terminals: set a = b
// if a terminal followed by nonterminal B: for every t in LEADING(B): set a <
// t if nonterminal A followed by terminal b: for every t in TRAILING(A): set
// t > b
for (int i = 0; i < n; ++i) {
  string rhs = rhs_of(i);
  for (size_t k = 0; k + 1 < rhs.size(); ++k) {
    char a = rhs[k], b = rhs[k + 1];
    if (idx_of(T, a) != -1 && idx_of(T, b) != -1) {
      int ia = idx_of(T, a), ib = idx_of(T, b);
      table[ia][ib] = '=';
    }
    if (idx_of(T, a) != -1 && idx_of(NT, b) != -1) {
      int ia = idx_of(T, a);
      for (char t : LEAD[b]) {
        int it = idx_of(T, t);
        if (it != -1)
          table[ia][it] = '<';
      }
    }
    if (idx_of(NT, a) != -1 && idx_of(T, b) != -1) {
      int ib = idx_of(T, b);
      for (char t : TRAIL[a]) {
        int it = idx_of(T, t);
        if (it != -1)
          table[it][ib] = '>';
      }
    }
    // case A B C where a is terminal, b nonterminal, c terminal -> a < c and
    // maybe a = b? already handled
    if (k + 2 < rhs.size()) {
      char c = rhs[k + 2];
      if (idx_of(T, a) != -1 && idx_of(NT, b) != -1 && idx_of(T, c) != -1) {
        int ia = idx_of(T, a), ic = idx_of(T, c);
        table[ia][ic] = '=';
      }
    }
  }
}

// Usually we also set $ relations: $ < leading(S) and trailing(S) > $
// find start symbol as left side of first production
char start = prod[0][0];
int id_dollar = idx_of(T, '$');
if (id_dollar != -1) {
  for (char t : LEAD[start]) {
    int it = idx_of(T, t);
    if (it != -1)
```

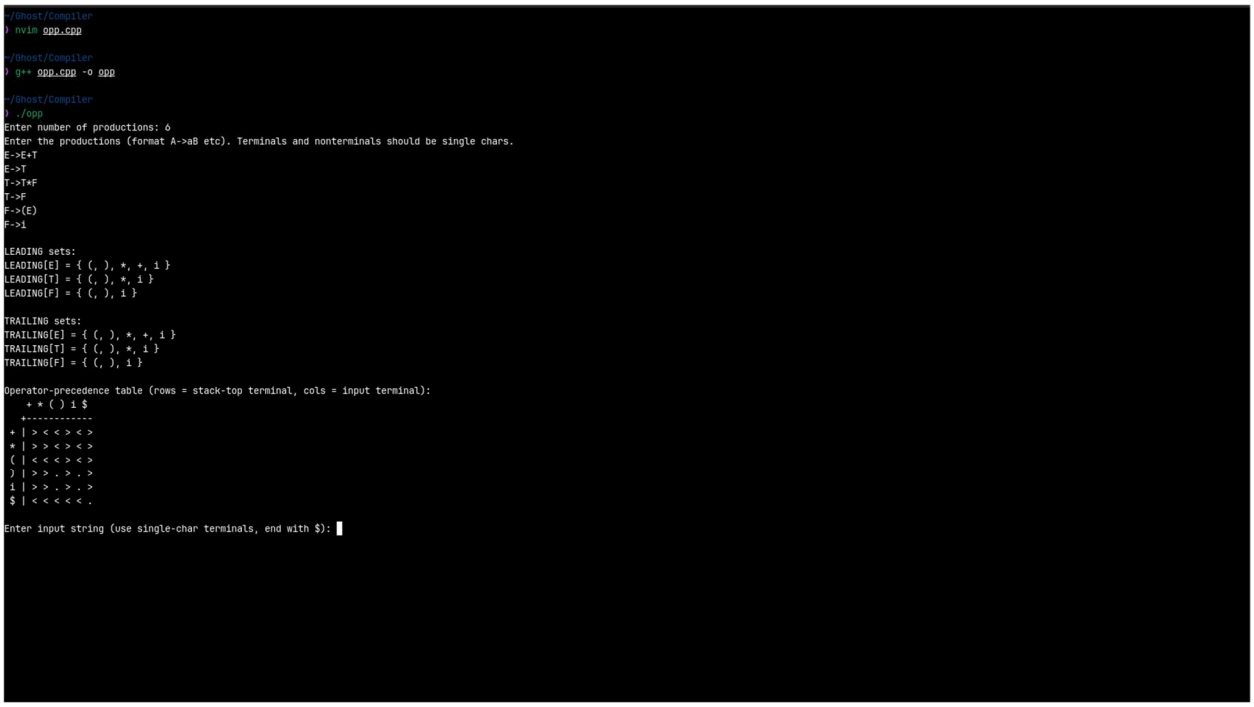
```
      table[id_dollar][it] = '<';
    }
    for (char t : TRAIL[start]) {
      int it = idx_of(T, t);
      if (it != -1)
        table[it][id_dollar] = '>';
    }
    // $ = $ maybe not needed, but set to blank
}

// Print precedence table
cout << "\nOperator-precedence table (rows = stack-top terminal, cols = "
        "input terminal):\n    ";
for (char t : T)
  cout << t << " ";
cout << "\n   +" << string(2 * Tn, '-') << "\n";
for (int i = 0; i < Tn; ++i) {
  cout << " " << T[i] << " | ";
  for (int j = 0; j < Tn; ++j) {
    cout << (table[i][j] == ' ' ? '.' : table[i][j]) << " ";
  }
  cout << "\n";
}

// ---- Parser (operator-precedence) ----
cout << "\nEnter input string (use single-char terminals, end with $): ";
string input;
cin >> input;
// ensure input ends with $
if (input.back() != '$')
  input.push_back('$');

// stack holds symbols (we'll store as chars). For parser we need to find the
// rightmost terminal on stack to compare precedence.
vector<char> stack_sym;
stack_sym.push_back('$'); // bottom marker

// for readability we show actions
auto print_state = [&](const vector<char> &stk, const string &inp,
                       const string &action) {
  // print stack as string
  cout << setw(20);
  string s;
  for (char c : stk)
    s.push_back(c);
  cout << s << "    ";
  cout << setw(20) << inp << "    ";
  cout << action << "\n";
};

cout << "\nParsing steps:\n";
```

```
cout << setw(20) << "Stack" << setw(25) << "Input" << setw(15) << "Action"
     << "\n";
cout << string(60, '-') << "\n";
string action;
size_t ip = 0;
// helper: find index of rightmost terminal in stack
auto rightmost_terminal_pos = [&](const vector<char> &stk) -> int {
  for (int i = (int)stk.size() - 1; i >= 0; --i) {
    if (idx_of(T, stk[i]) != -1)
      return i;
  }
  return -1;
};

bool accepted = false;
int safety = 0;
while (true) {
  // produce readable input remnant
  string rem = input.substr(ip);

  print_state(stack_sym, rem, "");
  if (safety++ > 1000) {
    cout << "Parsing loop limit reached. Aborting.\n";
    break;
  }

  // find topmost terminal on stack
  int pos = rightmost_terminal_pos(stack_sym);
  if (pos == -1) {
    cout << "No terminal on stack (error)\n";
    break;
  }
  char a = stack_sym[pos];
  char b = input[ip];
  int ia = idx_of(T, a), ib = idx_of(T, b);
  char rel = ' ';
  if (ia != -1 && ib != -1)
    rel = table[ia][ib];

  if (a == '$' && b == '$') {
    action = "ACCEPT";
    print_state(stack_sym, rem, action);
    accepted = true;
    break;
  }

  if (rel == '<' || rel == '=') {
    // SHIFT
    action = string("SHIFT ") + b;
    // push the input symbol onto stack
    stack_sym.push_back(b);
```

```
  ip++;
  print_state(stack_sym, input.substr(ip), action);
  continue;
 } else if (rel == '>') {
  // REDUCE: find handle between nearest terminal t (at pos) and the top of
  // stack According to operator-precedence algorithm, find nearest terminal
  // to left having '<' relation with the terminal at or to the right. We'll
  // find the leftmost terminal index lpos such that relation
  // table[lpos][ib_top] is '<' where ib_top is index of the terminal at
  // topmost terminal pos For simplicity: find leftmost terminal index 'l'
  // scanning back from pos-1 until either we find table[ idx(T,
  // stack_sym[l]) ][ idx(T, stack_sym[pos]) ] == '<' OR l==0 Then the
  // handle is everything after that terminal.
  int top_term_pos = rightmost_terminal_pos(stack_sym); // pos
  int lpos = -1;
  for (int k = top_term_pos - 1; k >= 0; --k) {
   if (idx_of(T, stack_sym[k]) != -1) {
    int ik = idx_of(T, stack_sym[k]);
    int itop = idx_of(T, stack_sym[top_term_pos]);
    if (table[ik][itop] == '<') {
     lpos = k;
     break;
    }
   }
  }
  if (lpos == -1) {
   // if none found, set lpos to 0 (bottom)
   lpos = 0;
  }
  // handle is substring stack_sym[lpos+1 ... end]
  int handle_start = lpos + 1;
  int handle_len = (int)stack_sym.size() - handle_start;
  string handle;
  for (int k = handle_start; k < (int)stack_sym.size(); ++k)
   handle.push_back(stack_sym[k]);

  // try to match handle to some production RHS
  bool reduced = false;
  char reduce_to = 'N'; // default nonterminal placeholder
  for (int p = 0; p < n; ++p) {
   string rhs = rhs_of(p);
   if (rhs == handle) {
    // reduce by replacing handle with LHS
    reduce_to = prod[p][0];
    // pop handle symbols
    for (int k = 0; k < handle_len; ++k)
     stack_sym.pop_back();
    stack_sym.push_back(reduce_to);
    action = string("REDUCE by ") + prod[p];
    reduced = true;
    break;
```

```
      }
    }
    if (!reduced) {
      // No exact match -> collapse handle to 'N' nonterminal (to continue)
      for (int k = 0; k < handle_len; ++k)
        stack_sym.pop_back();
      stack_sym.push_back(reduce_to);
      action = string("REDUCE (general) replace '") + handle + "' by N";
    }
    print_state(stack_sym, input.substr(ip), action);
    continue;
  } else {
    // no relation defined -> error
    action = "ERROR: no precedence relation between ";
    action.push_back(a);
    action.push_back('/');
    action.push_back(b);
    print_state(stack_sym, rem, action);
    break;
  }
 } // end parsing loop

 if (accepted)
   cout << "\nInput accepted by operator-precedence parser.\n";
 else
   cout << "\nInput rejected (or error occurred).\n";

 return 0;
}
```

Output:


~/Ghost/Compiler
❯ nvim opp.cpp
~/Ghost/Compiler
❯ g++ opp.cpp -o opp
~/Ghost/Compiler
❯ ./opp
Enter number of productions: 6
Enter the productions (format A->aB etc). Terminals and nonterminals should be single chars.
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
LEADING sets:
LEADING[E] = { (,), *, +, i }
LEADING[T] = { (,), *, i }
LEADING[F] = { (,), i }
TRAILING sets:
TRAILING[E] = { (,), *, +, i }
TRAILING[T] = { (,), *, i }
TRAILING[F] = { (,), i }
Operator-precedence table (rows = stack-top terminal, cols = input terminal):
+ * ( ) i $
+ | > < < > < >
* | > > < > < >
(| < < < > < .
) | > > . > . >
i | > > . > . >
$ | < < < < < .
Enter input string (use single-char terminals, end with $):

Experiment-8(b)

Aim:
To Use the Operator-precedence table in Experiment 8(a), perform the parsing for the given string.

Algorithm:
1.Input grammar
a. Read number of productions
b. Read each production in the form A -> RHS
c. Collect non terminals from the left side
d. Collect terminals from the right side and add dollar

2.Compute Leading sets
a. For each production check first symbol of RHS
b. If terminal add it to Leading of left side
c. If non terminal add its Leading set to left side
d. If non terminal followed by terminal then add that terminal
e. Repeat until no change

3.Compute Trailing sets
a. For each production check last symbol of RHS
b. If terminal add it to Trailing of left side
c. If non terminal add its Trailing set to left side
d. If terminal followed by non terminal then add that terminal
e. Repeat until no change

4.Construct precedence table
a. For each production check symbols in RHS
b. If two terminals are adjacent set equal between them
c. If terminal followed by non terminal then set less than with Leading of non terminal
d. If non terminal followed by terminal then set greater than with Trailing of non terminal
e. If terminal non terminal terminal appears then set equal between the two terminals
f. Add relations for dollar: dollar less than Leading of start symbol and Trailing of start symbol greater than dollar

5.Display Leading sets, Trailing sets, and precedence table

6.Parsing process
a. Initialize stack with dollar
b. Append dollar to end of input string
c. Repeat until input and stack are both consumed
i. Find rightmost terminal in stack

ii. Compare precedence relation with current input symbol
iii. If relation is less or equal then shift input symbol onto stack
iv. If relation is greater then find handle in stack and reduce it to a non terminal
v. If relation is missing then report error and stop
d. If both input and stack reduce to dollar accept the string

7.End

Code:

```
#include <bits/stdc++.h>
using namespace std;

using Set = set<char>;

// helper: print set
string set_to_str(const Set &s) {
  string out;
  for (char c : s) {
    out.push_back(c);
    out.push_back(',');
  }
  if (!out.empty())
    out.pop_back();
  return out;
}

int idx_of(const vector<char> &v, char c) {
  for (size_t i = 0; i < v.size(); ++i)
    if (v[i] == c)
      return (int)i;
  return -1;
}

int main() {

  cout << "Enter number of productions: ";
  int n;
  if (!(cin >> n))
    return 0;
  vector<string> prod(n);
  cout << "Enter the productions (format A->aB etc). Terminals and "
          "nonterminals should be single chars.\n";
  for (int i = 0; i < n; ++i) {
    cin >> prod[i];
    // allow both A->... and A->
    // we'll assume '->' at positions 1 and 2 or at index 1 for simplicity
    // normalize to form "A->rhs"
    if (prod[i].size() >= 2 && prod[i][1] != '-') {
      // maybe format like "A=abc"? leave as-is
```

```
  }
}

// collect nonterminals (left sides) and terminals (symbols appearing on RHS
// that are not nonterminals)
vector<char> NT; // nonterminals
vector<char> T;  // terminals

for (int i = 0; i < n; ++i) {
  char A = prod[i][0];
  if (idx_of(NT, A) == -1)
    NT.push_back(A);
}

// find RHS characters (starting after '->' which we assume starts at index 2
// or 3).
for (int i = 0; i < n; ++i) {
  string &p = prod[i];
  // find start of RHS
  size_t rhs_start = p.find("->");
  if (rhs_start == string::npos)
    rhs_start = 2;
  else
    rhs_start += 2;
  for (size_t k = rhs_start; k < p.size(); ++k) {
    char ch = p[k];
    if (ch == ' ' || ch == '\t')
      continue;
    if (idx_of(NT, ch) == -1 && idx_of(T, ch) == -1) {
      // if not a known nonterminal, treat as terminal for now
      if (!isupper(ch)) // heuristic: uppercase = nonterminal
        T.push_back(ch);
      else {
        // uppercase may be a nonterminal if it appears on LHS earlier; check:
        if (idx_of(NT, ch) == -1) {
          // treat uppercase not in NT as terminal (or possibly missing LHS)
          T.push_back(ch);
        }
      }
    }
  }
}

// If "$" not in terminals, add it as end marker
if (idx_of(T, '$') == -1)
  T.push_back('$');

// map nonterminal -> leading / trailing sets (terminals)
unordered_map<char, Set> LEAD, TRAIL;

// initialize empty sets
```

```
for (char A : NT) {
  LEAD[A] = Set();
  TRAIL[A] = Set();
}

// Helper lambdas to extract RHS and iterate
auto rhs_of = [&](int i) -> string {
  size_t rhs_start = prod[i].find("->");
  if (rhs_start == string::npos)
    rhs_start = 2;
  else
    rhs_start += 2;
  return prod[i].substr(rhs_start);
};

// Compute LEADING
// Rule: If A -> a... then a ∈ LEADING(A) (if a is terminal)
// If A -> B... and B is nonterminal, then LEADING(B) subset of LEADING(A)
// We'll do a fixed-point iterative closure.
bool changed = true;
while (changed) {
  changed = false;
  for (int i = 0; i < n; ++i) {
    char A = prod[i][0];
    string rhs = rhs_of(i);
    if (rhs.empty())
      continue;
    // first symbol
    char x = rhs[0];
    if (idx_of(T, x) != -1) {
      if (LEAD[A].insert(x).second)
        changed = true;
    } else if (idx_of(NT, x) != -1) {
      // add all LEAD[x] to LEAD[A]
      for (char t : LEAD[x])
        if (LEAD[A].insert(t).second)
          changed = true;
      // also if there is terminal after the nonterminal: A -> B a ...,
      // collect a But the iterative approach will propagate via other rules.
    }
    // Also, if first symbol is nonterminal followed by terminal, include that
    // terminal:
    for (size_t k = 0; k + 1 < rhs.size(); ++k) {
      char c1 = rhs[k], c2 = rhs[k + 1];
      if (idx_of(NT, c1) != -1 && idx_of(T, c2) != -1) {
        if (LEAD[A].insert(c2).second)
          changed = true;
      }
    }
  }
}
```

```
// Compute TRAILING similar but using last symbol
changed = true;
while (changed) {
  changed = false;
  for (int i = 0; i < n; ++i) {
    char A = prod[i][0];
    string rhs = rhs_of(i);
    if (rhs.empty())
      continue;
    char x = rhs[rhs.size() - 1];
    if (idx_of(T, x) != -1) {
      if (TRAIL[A].insert(x).second)
        changed = true;
    } else if (idx_of(NT, x) != -1) {
      for (char t : TRAIL[x])
        if (TRAIL[A].insert(t).second)
          changed = true;
    }
    // also if terminal followed by nonterminal anywhere -> add that terminal
    for (int k = (int)rhs.size() - 2; k >= 0; --k) {
      char c1 = rhs[k], c2 = rhs[k + 1];
      if (idx_of(T, c1) != -1 && idx_of(NT, c2) != -1) {
        if (TRAIL[A].insert(c1).second)
          changed = true;
      }
    }
  }
}

// Print LEADING and TRAILING
cout << "\nLEADING sets:\n";
for (char A : NT) {
  cout << "LEADING[" << A << "] = { ";
  bool first = true;
  for (char t : LEAD[A]) {
    if (!first)
      cout << ", ";
    cout << t;
    first = false;
  }
  cout << " }\n";
}
cout << "\nTRAILING sets:\n";
for (char A : NT) {
  cout << "TRAILING[" << A << "] = { ";
  bool first = true;
  for (char t : TRAIL[A]) {
    if (!first)
      cout << ", ";
    cout << t;
```

```
    first = false;
  }
  cout << " }\n";
}

// Build operator-precedence table among terminals.
// Initialize table with ' ' (no relation). We'll use chars: '<', '>', '=', '
// '.
int Tn = (int)T.size();
vector<vector<char>> table(
    Tn, vector<char>(Tn, ' ')); // table[i][j] relation between T[i] and T[j]

// for each production, for adjacent symbols a b:
// if a and b are terminals: set a = b
// if a terminal followed by nonterminal B: for every t in LEADING(B): set a <
// t if nonterminal A followed by terminal b: for every t in TRAILING(A): set
// t > b
for (int i = 0; i < n; ++i) {
  string rhs = rhs_of(i);
  for (size_t k = 0; k + 1 < rhs.size(); ++k) {
    char a = rhs[k], b = rhs[k + 1];
    if (idx_of(T, a) != -1 && idx_of(T, b) != -1) {
      int ia = idx_of(T, a), ib = idx_of(T, b);
      table[ia][ib] = '=';
    }
    if (idx_of(T, a) != -1 && idx_of(NT, b) != -1) {
      int ia = idx_of(T, a);
      for (char t : LEAD[b]) {
        int it = idx_of(T, t);
        if (it != -1)
          table[ia][it] = '<';
      }
    }
    if (idx_of(NT, a) != -1 && idx_of(T, b) != -1) {
      int ib = idx_of(T, b);
      for (char t : TRAIL[a]) {
        int it = idx_of(T, t);
        if (it != -1)
          table[it][ib] = '>';
      }
    }
    // case A B C where a is terminal, b nonterminal, c terminal -> a < c and
    // maybe a = b? already handled
    if (k + 2 < rhs.size()) {
      char c = rhs[k + 2];
      if (idx_of(T, a) != -1 && idx_of(NT, b) != -1 && idx_of(T, c) != -1) {
        int ia = idx_of(T, a), ic = idx_of(T, c);
        table[ia][ic] = '=';
      }
    }
  }
```

```
}

// Usually we also set $ relations: $ < leading(S) and trailing(S) > $
// find start symbol as left side of first production
char start = prod[0][0];
int id_dollar = idx_of(T, '$');
if (id_dollar != -1) {
  for (char t : LEAD[start]) {
    int it = idx_of(T, t);
    if (it != -1)
      table[id_dollar][it] = '<';
  }
  for (char t : TRAIL[start]) {
    int it = idx_of(T, t);
    if (it != -1)
      table[it][id_dollar] = '>';
  }
  // $ = $ maybe not needed, but set to blank
}

// Print precedence table
cout << "\nOperator-precedence table (rows = stack-top terminal, cols = "
        "input terminal):\n    ";
for (char t : T)
  cout << t << " ";
cout << "\n   +" << string(2 * Tn, '-') << "\n";
for (int i = 0; i < Tn; ++i) {
  cout << " " << T[i] << " | ";
  for (int j = 0; j < Tn; ++j) {
    cout << (table[i][j] == ' ' ? '.' : table[i][j]) << " ";
  }
  cout << "\n";
}

// ---- Parser (operator-precedence) ----
cout << "\nEnter input string (use single-char terminals, end with $): ";
string input;
cin >> input;
// ensure input ends with $
if (input.back() != '$')
  input.push_back('$');

// stack holds symbols (we'll store as chars). For parser we need to find the
// rightmost terminal on stack to compare precedence.
vector<char> stack_sym;
stack_sym.push_back('$'); // bottom marker

// for readability we show actions
auto print_state = [&](const vector<char> &stk, const string &inp,
                       const string &action) {
  // print stack as string
```

```
  cout << setw(20);
  string s;
  for (char c : stk)
    s.push_back(c);
  cout << s << "    ";
  cout << setw(20) << inp << "    ";
  cout << action << "\n";
};

cout << "\nParsing steps:\n";
cout << setw(20) << "Stack" << setw(25) << "Input" << setw(15) << "Action"
     << "\n";
cout << string(60, '-') << "\n";
string action;
size_t ip = 0;
// helper: find index of rightmost terminal in stack
auto rightmost_terminal_pos = [&](const vector<char> &stk) -> int {
  for (int i = (int)stk.size() - 1; i >= 0; --i) {
    if (idx_of(T, stk[i]) != -1)
      return i;
  }
  return -1;
};

bool accepted = false;
int safety = 0;
while (true) {
  // produce readable input remnant
  string rem = input.substr(ip);

  print_state(stack_sym, rem, "");
  if (safety++ > 1000) {
    cout << "Parsing loop limit reached. Aborting.\n";
    break;
  }

  // find topmost terminal on stack
  int pos = rightmost_terminal_pos(stack_sym);
  if (pos == -1) {
    cout << "No terminal on stack (error)\n";
    break;
  }
  char a = stack_sym[pos];
  char b = input[ip];
  int ia = idx_of(T, a), ib = idx_of(T, b);
  char rel = ' ';
  if (ia != -1 && ib != -1)
    rel = table[ia][ib];

  if (a == '$' && b == '$') {
    action = "ACCEPT";
```

```
  print_state(stack_sym, rem, action);
  accepted = true;
  break;
 }

 if (rel == '<' || rel == '=') {
  // SHIFT
  action = string("SHIFT ") + b;
  // push the input symbol onto stack
  stack_sym.push_back(b);
  ip++;
  print_state(stack_sym, input.substr(ip), action);
  continue;
 } else if (rel == '>') {
  // REDUCE: find handle between nearest terminal t (at pos) and the top of
  // stack According to operator-precedence algorithm, find nearest terminal
  // to left having '<' relation with the terminal at or to the right. We'll
  // find the leftmost terminal index lpos such that relation
  // table[lpos][ib_top] is '<' where ib_top is index of the terminal at
  // topmost terminal pos For simplicity: find leftmost terminal index 'l'
  // scanning back from pos-1 until either we find table[ idx(T,
  // stack_sym[l]) ][ idx(T, stack_sym[pos]) ] == '<' OR l==0 Then the
  // handle is everything after that terminal.
  int top_term_pos = rightmost_terminal_pos(stack_sym); // pos
  int lpos = -1;
  for (int k = top_term_pos - 1; k >= 0; --k) {
   if (idx_of(T, stack_sym[k]) != -1) {
    int ik = idx_of(T, stack_sym[k]);
    int itop = idx_of(T, stack_sym[top_term_pos]);
    if (table[ik][itop] == '<') {
     lpos = k;
     break;
    }
   }
  }
  if (lpos == -1) {
   // if none found, set lpos to 0 (bottom)
   lpos = 0;
  }
  // handle is substring stack_sym[lpos+1 ... end]
  int handle_start = lpos + 1;
  int handle_len = (int)stack_sym.size() - handle_start;
  string handle;
  for (int k = handle_start; k < (int)stack_sym.size(); ++k)
   handle.push_back(stack_sym[k]);

  // try to match handle to some production RHS
  bool reduced = false;
  char reduce_to = 'N'; // default nonterminal placeholder
  for (int p = 0; p < n; ++p) {
   string rhs = rhs_of(p);
```

```
      if (rhs == handle) {
        // reduce by replacing handle with LHS
        reduce_to = prod[p][0];
        // pop handle symbols
        for (int k = 0; k < handle_len; ++k)
          stack_sym.pop_back();
        stack_sym.push_back(reduce_to);
        action = string("REDUCE by ") + prod[p];
        reduced = true;
        break;
      }
    }
    if (!reduced) {
      // No exact match -> collapse handle to 'N' nonterminal (to continue)
      for (int k = 0; k < handle_len; ++k)
        stack_sym.pop_back();
      stack_sym.push_back(reduce_to);
      action = string("REDUCE (general) replace '") + handle + "' by N";
    }
    print_state(stack_sym, input.substr(ip), action);
    continue;
  } else {
    // no relation defined -> error
    action = "ERROR: no precedence relation between ";
    action.push_back(a);
    action.push_back('/');
    action.push_back(b);
    print_state(stack_sym, rem, action);
    break;
  }
 } // end parsing loop

 if (accepted)
  cout << "\nInput accepted by operator-precedence parser.\n";
 else
  cout << "\nInput rejected (or error occurred).\n";
 return 0;
}
```

Output:

```
TRAILING[T] = { (, ), *, i }
TRAILING[F] = { (, ), i }

Operator-precedence table (rows = stack-top terminal, cols = input terminal):
    + * ( ) i $
   +-----------
 + | > < < > < >
 * | > > < > < >
 ( | < < < > < >
 ) | > > . > . >
 i | > > . > . >
 $ | < < < < < .

Enter input string (use single-char terminals, end with $): i+i*i

Parsing steps:
              Stack                    Input         Action
------------------------------------------------------------
                  $                  i+i*i$
                 $i                   +i*i$    SHIFT i
                 $i                   +i*i$
                 $F                   +i*i$    REDUCE by F->i
                 $F                   +i*i$
                $F+                    i*i$    SHIFT +
                $F+                    i*i$
               $F+i                     *i$    SHIFT i
               $F+i                     *i$
               $F+F                     *i$    REDUCE by F->i
               $F+F                     *i$
              $F+F*                      i$    SHIFT *
              $F+F*                      i$
             $F+F*i                       $    SHIFT i
             $F+F*i                       $
             $F+F*F                       $    REDUCE by F->i
             $F+F*F                       $
               $F+N                       $    REDUCE (general) replace 'F*F' by N
               $F+N                       $
                 $N                       $    REDUCE (general) replace 'F+N' by N
                 $N                       $
                 $N                       $    ACCEPT

Input accepted by operator-precedence parser.
```