PCD Laboratory

Program 1: Tokenization

```
import java.io.*;
import java.util.Scanner;


public class Tokenization {


  // List of keywords in C
  private static final String[] keywords = {
    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if",
    "inline", "int", "long", "register", "restrict", "return", "short",
    "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
    "unsigned", "void", "volatile", "while"
  };


  // Function to check if a word is a keyword
  public static boolean isKeyword(String word) {
    for (String keyword : keywords) {
      if (word.equals(keyword)) {
        return true; // It's a keyword
      }
    }
    return false; // Not a keyword
  }


  // Function to check if a word is a valid identifier
```

```
public static boolean isIdentifier(String word) {
    if (Character.isLetter(word.charAt(0)) || word.charAt(0) == '_') {
        for (int i = 1; i < word.length(); i++) {
            if (!Character.isLetterOrDigit(word.charAt(i)) && word.charAt(i) != '_') {
                return false; // Not a valid identifier
            }
        }
        return true; // It's a valid identifier
    }
    return false; // Not an identifier
}

// Function to classify and print tokens
public static void classifyToken(String token) {
    if (isKeyword(token)) {
        System.out.println("Keyword: " + token);
    } else if (isIdentifier(token)) {
        System.out.println("Identifier: " + token);
    } else if (Character.isDigit(token.charAt(0))) {
        System.out.println("Number: " + token);
    } else {
        System.out.println("Operator/Symbol: " + token);
    }
}

// Function to tokenize the input C code
public static void tokenize(String input) {
    StringBuilder token = new StringBuilder();
```

```
        for (int i = 0; i < input.length(); i++) {
            char c = input.charAt(i);

            // Collecting alphanumeric characters or underscores
            if (Character.isLetterOrDigit(c) || c == '_') {
                token.append(c);
            } else {
                if (token.length() > 0) {
                    classifyToken(token.toString()); // Classify and print the token
                    token.setLength(0); // Reset the token
                }
                if (c != ' ' && c != '\n' && c != '\t') {
                    System.out.println("Operator/Symbol: " + c); // Non-alphanumeric characters (operators, symbols)
                }
            }
        }

        // To handle the last token if there is one
        if (token.length() > 0) {
            classifyToken(token.toString());
        }
    }

    // Function to read C code from a file
    public static String readFile(String filename) throws IOException {
        StringBuilder content = new StringBuilder();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
```

```
        String line;
        while ((line = reader.readLine()) != null) {
            content.append(line).append("\n");
        }
    }
    return content.toString();
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Ask user for file path
    System.out.print("Enter the path to the C program file: ");
    String filePath = scanner.nextLine();

    try {
        // Read C program from the file
        String inputCode = readFile(filePath);
        System.out.println("\nTokens in the input C program:");

        // Tokenize and classify the input code
        tokenize(inputCode);

    } catch (IOException e) {
        System.err.println("An error occurred while reading the file: " + e.getMessage());
    }

    scanner.close();
```

```
    }
}
```

Output:

```
Enter the path to the C program file: E:\Sandeep\Sandeep Java\sample.c

Tokens in the input C program:
Operator/Symbol: #
Identifier: include
Operator/Symbol: <
Identifier: stdio
Operator/Symbol: .
Identifier: h
Operator/Symbol: >
Keyword: int
Identifier: main
Operator/Symbol: (
Operator/Symbol: )
Operator/Symbol: {
Keyword: int
Identifier: a
Operator/Symbol: ,
Identifier: b
Operator/Symbol: ,
Identifier: counter
Operator/Symbol: ;
Keyword: float
Identifier: x
Operator/Symbol: ;
Keyword: char
Identifier: ch
Operator/Symbol: ;
Keyword: double
Identifier: d
Operator/Symbol: ;
Identifier: a
Operator/Symbol: =
Number: 10
Operator/Symbol: ;
Identifier: x
Operator/Symbol: =
Number: 5
Operator/Symbol: .
Number: 5
Operator/Symbol: ;
Keyword: return
Number: 0
Operator/Symbol: ;
Operator/Symbol: }
```

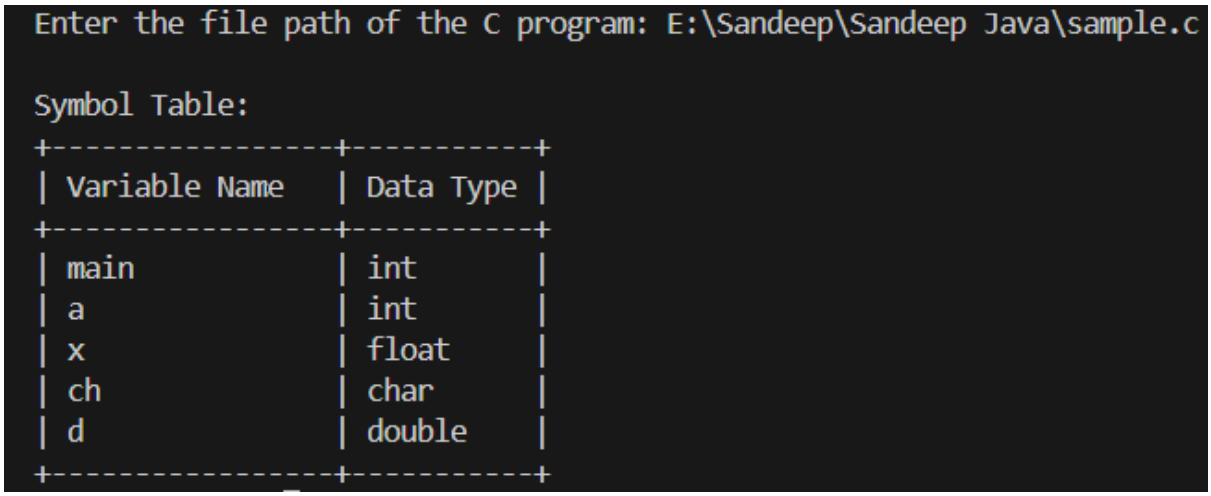
PROGRAM 2: Symbol Table

```
import java.io.*;
import java.util.*;
import java.util.regex.*;

class SymbolTable {
  private Map<String, String> table;

  public SymbolTable() {
    table = new LinkedHashMap<>();
  }

  public void addSymbol(String name, String type) {
    if (!table.containsKey(name)) {
      table.put(name, type);
    }
  }

  public void display() {
    System.out.println("\nSymbol Table:");
    System.out.println("+-----------------+-----------+");
    System.out.printf("| %-15s | %-9s |\n", "Variable Name", "Data Type");
    System.out.println("+-----------------+-----------+");
    for (Map.Entry<String, String> entry : table.entrySet()) {
      System.out.printf("| %-15s | %-9s |\n", entry.getKey(), entry.getValue());
    }
```

```
        System.out.println("+-----------------+-----------+");
    }
}

public class SymbolTableGenerator {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        SymbolTable symbolTable = new SymbolTable();

        System.out.print("Enter the file path of the C program: ");
        String fileName = sc.nextLine();

        String regex = "\\b(int|float|char|double)\\s+([a-zA-Z_][a-zA-Z0-9_]*)";
        Pattern pattern = Pattern.compile(regex);

        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                Matcher matcher = pattern.matcher(line);
                while (matcher.find()) {
                    String type = matcher.group(1);
                    String name = matcher.group(2);
                    symbolTable.addSymbol(name, type);
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
```

```
        }


        symbolTable.display();
        sc.close();
    }
}
```

OUTPUT:



Program 3: NFA Construction

```
import java.util.*;

class State {
   String name;
   Map<Character, List<State>> transitions = new HashMap<>();

   public State(String name) {
      this.name = name;
   }

   public void addTransition(char symbol, State state) {
      transitions.computeIfAbsent(symbol, k -> new ArrayList<>()).add(state);
   }
}

class NFA {
   State startState;
   Set<State> acceptStates;

   public NFA(State startState, Set<State> acceptStates) {
      this.startState = startState;
      this.acceptStates = acceptStates;
   }

   public void display() {
```

```
System.out.println("\nNFA Transitions:");
System.out.println("+------------+--------+------------+");
System.out.printf("| %-10s | %-6s | %-10s |\n", "From State", "Symbol", "To State");
System.out.println("+------------+--------+------------+");

Set<State> visited = new HashSet<>();
Queue<State> queue = new LinkedList<>();
queue.add(startState);
visited.add(startState);

while (!queue.isEmpty()) {
  State state = queue.poll();
  for (Map.Entry<Character, List<State>> entry : state.transitions.entrySet()) {
    for (State nextState : entry.getValue()) {
      System.out.printf("| %-10s | %-6s | %-10s |\n", state.name,
          entry.getKey() == 'ε' ? "ε" : entry.getKey(),
          nextState.name);
      if (!visited.contains(nextState)) {
        queue.add(nextState);
        visited.add(nextState);
      }
    }
  }
}
System.out.println("+------------+--------+------------+");

System.out.println("\nStart State: " + startState.name);
System.out.print("Accept States: ");
```

```
        for (State accept : acceptStates) {
            System.out.print(accept.name + " ");
        }
        System.out.println("\n");
    }
}

public class NFAConstructor {
    static int stateCount = 0;

    public static State newState() {
        return new State("temp" + stateCount++);
    }

    public static String insertConcat(String regex) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < regex.length(); i++) {
            char curr = regex.charAt(i);
            result.append(curr);
            if (i + 1 < regex.length()) {
                char next = regex.charAt(i + 1);
                if ((Character.isLetterOrDigit(curr) || curr == '*' || curr == ')') &&
                    (Character.isLetterOrDigit(next) || next == '(')) {
                    result.append('.');
                }
            }
        }
        return result.toString();
```

```
}

public static int precedence(char op) {
    switch (op) {
        case '*': return 3;
        case '.': return 2;
        case '|': return 1;
        default: return 0;
    }
}

public static NFA buildNFA(String regex) {
    Stack<NFA> stack = new Stack<>();
    Stack<Character> operators = new Stack<>();
    regex = insertConcat(regex);

    for (int i = 0; i < regex.length(); i++) {
        char c = regex.charAt(i);

        if (c == '(') {
            operators.push(c);
        } else if (c == ')') {
            while (operators.peek() != '(') {
                processOperator(stack, operators.pop());
            }
            operators.pop();
        } else if (c == '|' || c == '.' || c == '*') {
```

```
            while (!operators.isEmpty() && precedence(operators.peek()) >= precedence(c))
{
                processOperator(stack, operators.pop());
            }
            operators.push(c);
        } else {
            State start = newState();
            State end = newState();
            start.addTransition(c, end);
            stack.push(new NFA(start, new HashSet<>(Collections.singleton(end))));
        }
    }

    while (!operators.isEmpty()) {
        processOperator(stack, operators.pop());
    }

    NFA finalNFA = stack.pop();
    State newStart = new State("temp" + stateCount++);
    newStart.addTransition('ε', finalNFA.startState);

    return renameStates(new NFA(newStart, finalNFA.acceptStates));
  }

  private static void processOperator(Stack<NFA> stack, char op) {
    if (op == '*') {
        NFA nfa = stack.pop();
        State start = newState();
```

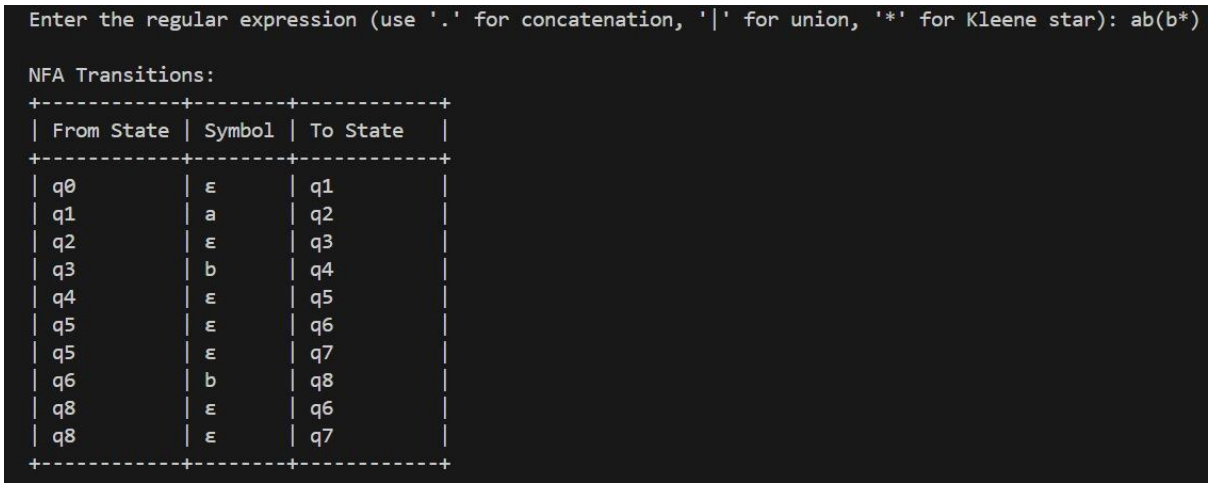
```
        State end = newState();
        start.addTransition('ε', nfa.startState);
        start.addTransition('ε', end);
        for (State accept : nfa.acceptStates) {
            accept.addTransition('ε', nfa.startState);
            accept.addTransition('ε', end);
        }
        stack.push(new NFA(start, new HashSet<>(Collections.singleton(end))));
    } else if (op == '.') {
        NFA right = stack.pop();
        NFA left = stack.pop();
        for (State accept : left.acceptStates) {
            accept.addTransition('ε', right.startState);
        }
        stack.push(new NFA(left.startState, right.acceptStates));
    } else if (op == '|') {
        NFA right = stack.pop();
        NFA left = stack.pop();
        State start = newState();
        State end = newState();
        start.addTransition('ε', left.startState);
        start.addTransition('ε', right.startState);
        for (State accept : left.acceptStates) {
            accept.addTransition('ε', end);
        }
        for (State accept : right.acceptStates) {
            accept.addTransition('ε', end);
        }
```

```
        stack.push(new NFA(start, new HashSet<>(Collections.singleton(end))));
    }
}

// Reassign state names sequentially as q0, q1, q2...
public static NFA renameStates(NFA nfa) {
    Map<State, String> newNames = new HashMap<>();
    Queue<State> queue = new LinkedList<>();
    Set<State> visited = new HashSet<>();

    int counter = 0;
    queue.add(nfa.startState);
    visited.add(nfa.startState);
    newNames.put(nfa.startState, "q" + counter++);

    while (!queue.isEmpty()) {
        State state = queue.poll();
        for (Map.Entry<Character, List<State>> entry : state.transitions.entrySet()) {
            for (State next : entry.getValue()) {
                if (!visited.contains(next)) {
                    newNames.put(next, "q" + counter++);
                    visited.add(next);
                    queue.add(next);
                }
            }
        }
    }
```

```
        // Rename states
        Map<String, State> renamedStates = new HashMap<>();
        for (State oldState : newNames.keySet()) {
            renamedStates.put(newNames.get(oldState), new
State(newNames.get(oldState)));
        }

        // Reconnect transitions with new names
        for (State oldState : newNames.keySet()) {
            State newState = renamedStates.get(newNames.get(oldState));
            for (Map.Entry<Character, List<State>> entry : oldState.transitions.entrySet()) {
                for (State next : entry.getValue()) {
                    newState.addTransition(entry.getKey(),
renamedStates.get(newNames.get(next)));
                }
            }
        }

        Set<State> newAcceptStates = new HashSet<>();
        for (State accept : nfa.acceptStates) {
            newAcceptStates.add(renamedStates.get(newNames.get(accept)));
        }

        return new NFA(renamedStates.get(newNames.get(nfa.startState)),
newAcceptStates);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```
        System.out.print("Enter the regular expression (use '.' for concatenation, '|' for union, '*' for Kleene star): ");
        String regex = scanner.nextLine();
        scanner.close();

        NFA nfa = buildNFA(regex);
        nfa.display();
    }
}
```

Output:



```
Enter the regular expression (use '.' for concatenation, '|' for union, '*' for Kleene star): ab(b*)

NFA Transitions:
+------------+--------+------------+
| From State | Symbol | To State   |
+------------+--------+------------+
| q0         | ε      | q1         |
| q1         | a      | q2         |
| q2         | ε      | q3         |
| q3         | b      | q4         |
| q4         | ε      | q5         |
| q5         | ε      | q6         |
| q5         | ε      | q7         |
| q6         | b      | q8         |
| q8         | ε      | q6         |
| q8         | ε      | q7         |
+------------+--------+------------+
```

Program 4: Minimized DFA

```
import java.util.*;

class DFA {
  static class State {
    String name;
    boolean isAccept;
    Map<Character, State> transitions;

    State(String name, boolean isAccept) {
      this.name = name;
      this.isAccept = isAccept;
      this.transitions = new HashMap<>();
    }

    void addTransition(char symbol, State state) {
      transitions.put(symbol, state);
    }
  }

  private List<State> states;
  private State startState;
  private Set<Character> alphabet;

  public DFA() {
    states = new ArrayList<>();
```

```
    alphabet = new HashSet<>();
}

public void addState(State state, boolean isStart) {
    states.add(state);
    if (isStart) {
        startState = state;
    }
}

public void addAlphabet(Set<Character> symbols) {
    alphabet.addAll(symbols);
}

public void minimize() {
    Set<Set<State>> partitions = new HashSet<>();
    Set<State> acceptStates = new HashSet<>();
    Set<State> nonAcceptStates = new HashSet<>();

    for (State state : states) {
        if (state.isAccept) acceptStates.add(state);
        else nonAcceptStates.add(state);
    }

    if (!acceptStates.isEmpty()) partitions.add(acceptStates);
    if (!nonAcceptStates.isEmpty()) partitions.add(nonAcceptStates);

    boolean changed;
```

```
    do {
      changed = false;
      Set<Set<State>> newPartitions = new HashSet<>();

      for (Set<State> group : partitions) {
        Map<Map<Character, Set<State>>, Set<State>> transitionGroups = new
HashMap<>();

        for (State state : group) {
          Map<Character, Set<State>> key = new HashMap<>();
          for (char symbol : alphabet) {
            State target = state.transitions.get(symbol);
            for (Set<State> partition : partitions) {
              if (partition.contains(target)) {
                key.put(symbol, partition);
                break;
              }
            }
          }
          transitionGroups.computeIfAbsent(key, k -> new HashSet<>()).add(state);
        }

        newPartitions.addAll(transitionGroups.values());
        if (transitionGroups.values().size() > 1) changed = true;
      }

      partitions = newPartitions;
    } while (changed);
```

```
        System.out.println("\nMinimized DFA States:");
        for (Set<State> group : partitions) {
            System.out.print("{ ");
            for (State state : group) {
                System.out.print(state.name + " ");
            }
            System.out.println("}");
        }
    }
}

public class MinimizedDFA {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the regular expression: ");
        String regex = sc.nextLine();

        DFA dfa = new DFA();
        Set<Character> alphabet = new HashSet<>();

        // Example DFA Construction (for simplicity)
        DFA.State q0 = new DFA.State("q0", false);
        DFA.State q1 = new DFA.State("q1", true);

        dfa.addState(q0, true);
        dfa.addState(q1, false);
```

```
        for (char c : regex.toCharArray()) {
            if (Character.isLetterOrDigit(c)) {
                q0.addTransition(c, q1);
                alphabet.add(c);
            }
        }

        dfa.addAlphabet(alphabet);
        dfa.minimize();
        sc.close();
    }
}
```

Output: