DESIGN DOCUMENT

Project 5 – Infix Expression Calculator
Author: Dinesh Seveti
Course: CSCI 301 – Data Structures
GA for: Dr. Jie Meichsner
Date: October 2025

Program Objective
The goal of this assignment is to implement an **Infix Expression Calculator** that evaluates arithmetic expressions using **stacks** and **C++ exceptions**. The calculator verifies input validity, checks balanced parentheses, converts infix to postfix form, and evaluates the postfix expression to obtain the final numeric result.

Design Overview
The program follows an **object-oriented** structure written in C++.
It consists of three main components:

1. **Stack ADT (Stack.h)**
    - Implemented with a **singly linked list**.
    - Provides push, pop, top, isEmpty, and clear.
    - Uses exceptions to handle illegal operations.

2. **InfixCalculator Class (InfixCalculator.h / InfixCalculator.cpp)**
    - **Private members:**
        - std::string infixExp – stores the user expression.
        - isWellFormed() – validates syntax.
        - isBalanced() – checks matching parentheses.
        - infixToPostfix() – converts infix → postfix.
        - precedence() – returns operator precedence.
        - evalPostfix() – evaluates postfix form.

    - **Public members:**
        - setExpression() – validates and sets expression.
        - evaluate() – performs conversion + evaluation and returns result.

3. **Driver Program (main.cpp)**
    - Provides console interaction.
    - Prompts repeatedly for input until exit.
    - Displays result or error messages.

Algorithmic Design

Step 1 – Validation: Check every character:

- Only digits, operators (+ − * /), and parentheses are allowed.
- Ensures no two operators appear together.
- Uses a stack to verify balanced parentheses.

Step 2 – Infix → Postfix Conversion: Follows the **Shunting Yard Algorithm**:

1. Read tokens left to right.
2. Append operands directly to output.
3. Push operators onto stack according to precedence.

Step 3 – Postfix Evaluation

1. Read tokens left to right.
2. Push operands to stack.
3. On operator: pop two operands, apply operation, push result.
4. Final stack top = answer.

Step 4 – Error Handling

- **Division by zero** → throws runtime_error("Division by zero").
- **Unbalanced parentheses** → detected by isBalanced().
- **Malformed input** → rejected by isWellFormed().

UML Diagram

```
+---------------------+
|   InfixCalculator   |
+---------------------+
| - infixExp : string |
|---------------------|
| + isWellFormed()    |
| + isBalanced()      |
| + infixToPostfix()  |
| + precedence()      |
| + evalPostfix()     |
| + setExpression()   |
| + evaluate()        |
+---------------------+

+---------+
|  Stack  |
+---------+
| - top   |
| + push()|
| + pop() |
| + top() |
| + clear()|
+---------+
```

Test Data and Explanation

Category	**Input Expression**	**Expected Output**	**Explanation**
Basic Valid	2+3	5	Adds two operands.
Operator Precedence	2+3*4	14	Multiplication before addition.
Nested Parentheses	(2+3)*4	20	Parentheses evaluated first.
Complex Case	(9-3)/(2+1)	2	Combines multiple operators and brackets.
Invalid Expression	3*/4	Invalid or unbalanced expression	Consecutive operators detected.
Division by Zero	9/(3-3)	Error: Division by zero	Denominator evaluates to 0.

Testing Method
Each expression was entered through the interactive console.
A recorded terminal session (script.txt) shows input and output results for all cases.