SymbolBalance.py - Syntax Checking and Infix to Postfix Conversion

1. Introduction

This document provides an overview of the SymbolBalance.py program, which processes mathematical expressions to check for syntax errors and convert infix notation to postfix notation.

2. Program Design

The program is designed to:

1. Parse user input and remove comments.
2. Check for syntax errors using a stack-based approach.
3. Convert infix expressions to postfix notation if syntax is valid.

3. Algorithm and Pseudocode

3.1 parseInput Method

Functionality:

- Accept user input
- Removes comments enclosed in /* */.
- Returns a cleaned version of the input expression.

Pseudocode:

function parseInput(expression):

Accept User input

Remove any comments enclosed in /* */

Return the cleaned expression

3.2 checkSyntax Method

Functionality:

- Uses a stack to verify the balance of {}, [], (), and /* */.
- Detects and reports three types of syntax errors:
    1. Unmatched opening symbol (NonEmptyStack Error)
    2. Unmatched closing symbol (EmptyStack Error)
    3. Mismatched symbols (Mismatch Error)
- Returns a message if the input is balanced.

Pseudocode:

```
function checkSyntax(expression):
    Initialize an empty stack
    Define matching pairs for { }, [ ], ( ), /* */
    For each character in expression:
        If character is an opening symbol, push onto stack
        If character is a closing symbol:
            If stack is empty, return "EmptyStack Error"
            Pop from stack and check if matches opening symbol
            If mismatch, return "Mismatch Error"
    If stack is not empty, return "NonEmptyStack Error"
    Return "Symbol Balanced"
```

3.3 postfixExpress Method

Functionality:

- Converts infix expressions to postfix using the **Shunting-yard algorithm**.
- Uses operator precedence to handle expressions correctly.
- Runs only if checkSyntax confirms the input is balanced.

Pseudocode:

```
function postfixExpress(expression):
    Initialize an empty stack for operators
    Initialize an empty list for output
    Define operator precedence
    For each token in expression:
        If token is operand, add to output
        If token is operator:
            While stack is not empty and precedence of top is greater:
                Pop from stack and add to output
            Push operator onto stack
    Pop remaining stack operators to output
    Return postfix expression
```

4. Data Structures

- **Stack** - Used for syntax checking and operator precedence management.
- **List** - Used for storing the final postfix output.

5. API Specifications

parseInput(expression: str) -> str

- **Input -** Mathematical expression (with or without comments)
- **Output:** Cleaned expression without comments

checkSyntax(expression: str) -> str

- **Input-** Mathematical expression
- **Output-** Syntax validation result or error message

postfixExpress(expression: str) -> str

- **Input-** Balanced infix expression
- **Output-** Equivalent postfix expression

6. Example Screenshots and Behavior

This part is divided into two: Valid and Syntax Errors

6.1 Valid Input

a) Input: { a + [b * (c + d)] / e } , **Output:** a b c d + * e / +

```
Enter the expression: { a + [ b * ( c + d ) ] / e }
Symbol Balanced
Postfix Expression: a b c d + * e / +
```

6.2 Syntax Errors

1) The input ends with one or more symbols missing their corresponding closing symbols;
   INPUT: (a + b * c

```
Enter the expression: (a + b * c
Error1: NonEmptyStack error, unmatched ( left in stack
```

2) There is a closing symbol without its corresponding opening symbol;
   INPUT: `a + b) * c`**,**

```
Enter the expression: a + b) * c
Error2: EmptyStack error, missing opening symbol for ) at position 5
```

3) There is a mismatch between closing and opening symbols, e.g. { (}).
INPUT: `{ ( a + b ] ) }`