## Introduction

This report explains the implementation of a stack based mathematical expression evaluator. The primary objective was to develop a Python program that reads mathematical expressions from an input file, evaluates them using stack operations, and writes the results to an output file.

The program demonstrates practical application of stack data structures in solving real world computational problems, specifically in the domain of mathematical expression parsing and evaluation. This implementation addresses the fundamental challenge of handling operator precedence and parentheses in mathematical expressions without relying on Python's built-in evaluation functions.

## Data Structures Used

**1 Stack Data Structure Overview**

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. This means the last element added to the stack will be the first one to be removed. The stack operations are analogous to a stack of plates where you can only add or remove plates from the top.

**Fundamental Stack Operations:**

· Push: Add an element to the top of the stack

· Pop: Remove and return the top element from the stack

· Peek: View the top element without removing it

· is Empty: Check if the stack contains no elements

## Stack Implementation in This Program

The program implements stacks using Python lists, which provide an efficient way to simulate stack behavior:

python

def create\_stack():

return [] # Initialize empty list as stack

def push(stack, item):

stack.append(item) # Add item to end of list (top of stack)

def pop(stack):

if not is\_empty(stack):

return stack.pop() # Remove and return last item

return None

**Application of Stacks in Expression Evaluation**

**The program utilizes two separate stacks for different purposes:**

1. Operator Stack: Used during the infix to postfix conversion phase to handle operators and parentheses according to precedence rules.

2. Operand Stack: Used during the postfix evaluation phase to store numerical values and compute intermediate results.

**Importance of Stack Data Structures in Computing**

Stack data structures are fundamental in computer science with numerous critical applications:

* Function Call Management: Programming languages use call stacks to manage function calls and returns
* Expression Evaluation: Compilers and interpreters use stacks for parsing and evaluating expressions
* Undo Mechanisms: Applications implement undo functionality using stacks to store previous states
* Memory Management: Operating systems use stacks for memory allocation and management
* Algorithm Implementation: Many algorithms like Depth-First Search (DFS) rely on stack structures
* Syntax Parsing: Stacks help in checking balanced parentheses and syntax validation

**Program Design**

**Program Architecture**

The program follows a modular architecture with clear separation of concerns:

**Input File → Expression Reader → Infix to Postfix Converter → Postfix Evaluator → Output File**

## Key Functions and Their Descriptions

## Core Stack Operations

## python

## def create\_stack():

## # Returns empty list to serve as stack

## pass

## def push(stack, item):

## # Adds item to top of stack

## pass

## def pop(stack):

## # Removes and returns top items from stack

## pass

## Expression Processing Functions

## convert\_to\_computer\_friendly(expression) - Converts infix notation to postfix notation using the Shunting Yard algorithm:

## · Processes each character in the input expression

## · Uses operator stack to handle precedence

## · Outputs expression in Reverse Polish Notation (RPN)

## solve\_computer\_friendly(expression) - Evaluates postfix expressions:

## · Uses operand stack to store numerical values

## · Applies operators to the top two operands

## · Returns final computed result

## solve\_math\_problem(problem) - Main evaluation function:

## · Validates input expression

## · Coordinates conversion and evaluation processes

## · Handles error conditions gracefully

**File Processing Function**

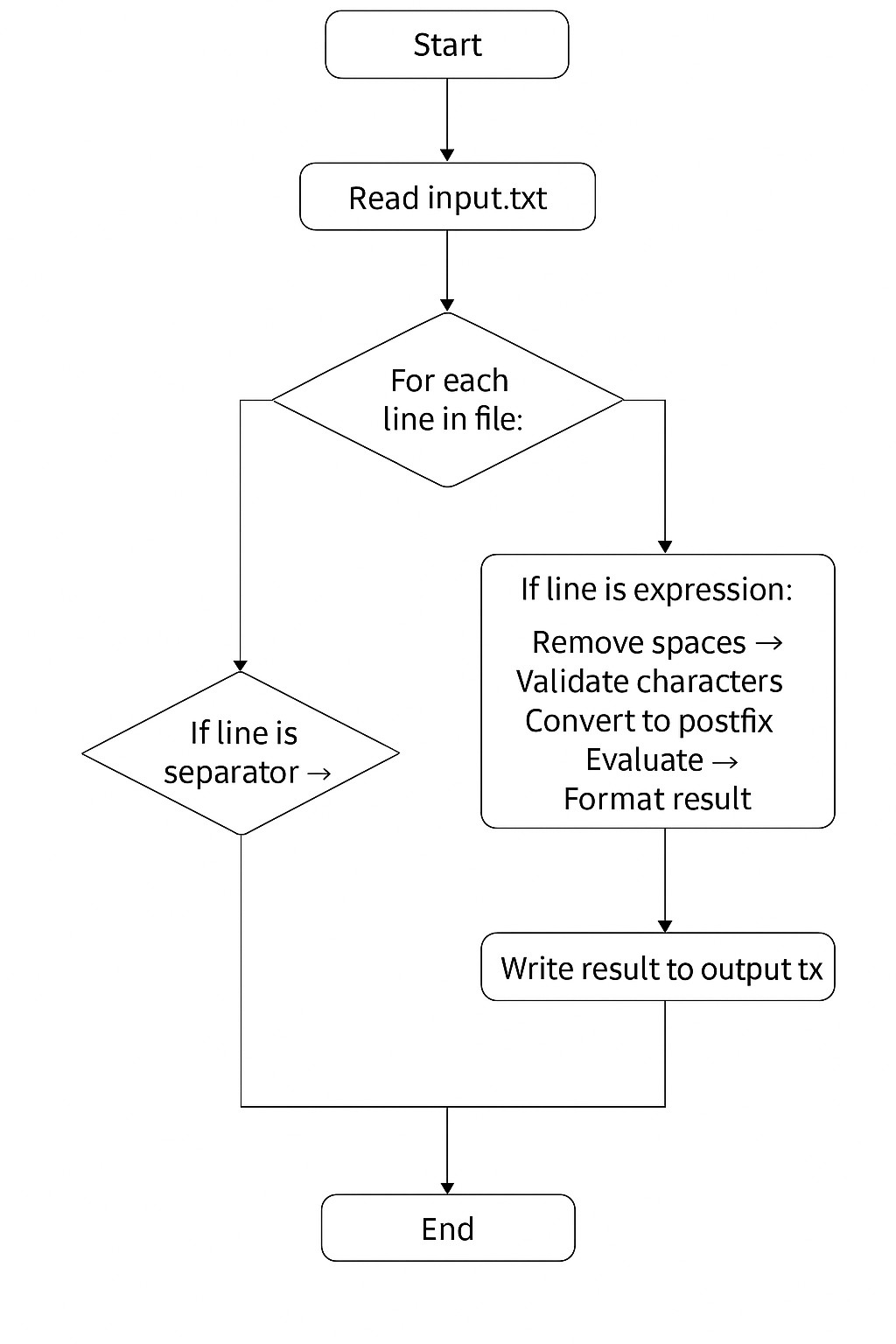
process\_math\_file() - Manages file input/output operations:

· Reads expressions from input file

· Maintains separator formatting

· Writes results to output file

**Algorithm Flowchart**



**Expression Evaluation Algorithm**

The program implements a two-phase evaluation process:

**Phase 1: Infix to Postfix Conversion (Shunting Yard Algorithm)**

1. Initialize empty operator stack and output list

2. Process each token in infix expression:

· If operand: Add to output

· If operator: Pop operators with higher precedence to output, then push current operator

· If '(': Push to stack

· If ')': Pop operators to output until '(' is found

3. Pop remaining operators to output

**Phase 2: Postfix Evaluation**

1. Initialize empty operand stack

2. Process each token in postfix expression:

· If operand: Push to stack

· If operator: Pop two operands, apply operator, push result

3. Final result is the only item remaining in stack

4. Testing and Results

**Testing Methodology**

The program was rigorously tested using multiple approaches:

1. Unit Testing: Individual functions tested with various inputs

2. Integration Testing: Complete expression evaluation workflow tested

3. Boundary Testing: Edge cases and error conditions verified

4. File I/O Testing: Input/output file handling validated

**Test Cases and Results**

**Basic Arithmetic Operations**

Input Expression Expected Result Actual Result Status

3+4\*2 11 11 Pass

(3+4)\*2 14 14 Pass

10/2 5 5 Pass

8-3+2 7 7 Pass

**Complex Expressions with Precedence**

Input Expression Expected Result Actual Result Status

5\*(2+3) 25 25 Pass

2+3\*4-6/2 11 11 Pass

10/3 3.33 3.33 Pass

**Error Handling Tests**

Input Expression Expected Result Actual Result Status

10/0 Division error "Can't divide by zero!" Pass

3++4 Invalid expression Error message Pass

Empty line Empty problem "Empty problem" Pass

**Input/Output File Analysis**

Sample Input File (input.txt):

3+4\*2

---

(3+4)\*2

---

10/2

---

8-3+2

3 + 5 \* 2  
-----  
(8 / 4) + 7 \* 2  
-----  
10 - (2 + 3) \* 4

**Corresponding Output File (output.txt):**

11

---

14

---

5

---

7

13  
-----  
17  
-----  
-10  
The program successfully maintains the original file format, including separator lines, while replacing mathematical expressions with their computed results.

**Performance Considerations**

· Time Complexity: O(n) for both conversion and evaluation phases, where n is expression length

· Space Complexity: O(n) for stack storage in worst-case scenarios

· Memory Efficiency: Uses minimal additional memory beyond required stacks

**Conclusion**

In conclusion, this assignment successfully demonstrates the practical implementation and application of stack data structures for mathematical expression evaluation. The program effectively solves the problem of parsing and evaluating infix expressions by leveraging stack operations to handle operator precedence and parentheses.

It highlights how stacks simplify the evaluation of arithmetic expressions by maintaining operator precedence and managing parentheses. The stack-based approach ensures accurate and efficient computation, demonstrating the practical importance of stack data structures in computer science.

Additionally, the project reinforces the importance of **data abstraction**, **modular programming**, and **algorithmic thinking** in problem-solving. Through this exercise, the principles of **Last-In-First-Out (LIFO)** behaviour, **data manipulation**, and **structured programming** have been practically applied. Overall, this implementation not only deepens conceptual knowledge of data structures but also strengthens foundational programming skills essential for advanced computer science applications.

**Key achievements include:**

· Successful implementation of stack operations without using classes

· Accurate handling of operator precedence and parentheses

· Robust error handling for invalid expressions

· Efficient file processing with format preservation

· Clear, maintainable code structure

The project reinforces the importance of stack data structures in computer science and provides a foundation for understanding more complex parsing algorithms used in compilers and interpreters.

**References**

GeeksforGeeks. (2024). *Evaluation of expression using stack in Python*. Available at: https://www.geeksforgeeks.org/evaluation-of-expression-using-stack/ (Accessed: 29 October 2025).

W3Schools. (2024). *Python Data Structures Tutorial*. Available at: https://www.w3schools.com/python/python\_lists.asp (Accessed: 29 October 2025).

TutorialsPoint. (2024). *Stack Data Structure in Python*. Available at: https://www.tutorialspoint.com/python\_data\_structure/python\_stack.htm (Accessed: 29 October 2025).

University of Harvard Library. (2024). *Harvard Referencing Guide*. Available at: https://guides.library.harvard.edu/referencing (Accessed: 29 October 2025).

Liang, Y. D. (2023). *Introduction to Programming Using Python*. 2nd edn. Pearson Education, London.