Samantha Lane
Data Structures Lab 1
Analysis

Description and Justification of Data Structures Used
The primary data structure used in this lab is a stack, implemented manually using a Python list. This structure was chosen because the logic of prefix-to-postfix conversion aligns closely with Last-In, First-Out (LIFO) behavior. As the expression is read from right to left, operands must be stored temporarily until the corresponding operator is encountered. At that point, the two most recent operands are retrieved, reordered, and the result is pushed back onto the stack. This behavior is naturally and efficiently modeled using a stack. Implementing the stack manually allowed for full control over push and pop behavior and ensured compatibility with assignment restrictions, which prohibited the use of built-in data structures like Stack, ArrayList, or Vector.

Appropriateness to the Application
The prefix-to-postfix conversion algorithm is fundamentally stack-based. The conversion requires tracking a dynamic set of operands and operators, maintaining the correct hierarchical order of operations. A stack provides an intuitive and efficient mechanism to model this process. Each operator combines exactly two operands, so the stack structure also helps identify malformed expressions by checking the number of items available to pop. This structure is particularly appropriate because of its simplicity, constant-time operations, and compatibility with both compact and space-separated prefix inputs.

Design Choices
Several deliberate design decisions were made to improve program robustness and flexibility:

1. The program accepts both compact (+AB) and space-separated (+ A B) prefix expressions by detecting the presence of spaces in each input line.
2. To comply with the course guidelines, the stack was implemented as a list. This allowed for full transparency and avoided reliance on built-in libraries.
3. All input and output is handled through user-specified files, not hardcoded paths. This design choice supports repeatable testing and allows the program to be used with varied data sets.
4. Malformed expressions (e.g., insufficient operands, extra operands, or invalid characters) are caught using exception handling. The program does not terminate on error but records and reports the issue per expression.
5. Expressions are read from right to left and tokenized either by space or individual character. This was critical for supporting both standard and compact input formats.

Efficiency Analysis
Time Complexity: O(n), where n is the number of tokens in the expression. Each token is processed once, and stack operations are constant time.
Space Complexity: O(n) in the worst case, as all operands may temporarily reside on the stack.
The algorithm is both time- and space-efficient for this class of problem and scales linearly with the size of the input expression.

Skills and Concepts Learned

This project reinforced the fundamental behavior of stacks and their practical application to expression parsing. It highlighted the importance of thoroughly validating input, especially when dealing with nested structures or user-generated data. I also gained experience in designing flexible input-handling logic and in anticipating malformed input. Writing and testing the program clarified the structural relationship between prefix and postfix notation. This understanding helped me verify correctness without relying on output alone, and it prepared me to work with expression trees in future assignments.

Improvements for the Future

In a future iteration, I would consider the following enhancements:

- Implement more detailed error messages to specify why an expression is invalid (e.g., "too few operands" vs. "too many operands").
- Extend the program to support unary operators or ternary expressions.
- Add support for full expression trees and optional visualization.
- Create a user-facing interface for direct interaction rather than requiring file-based I/O.

Enhancements Implemented

Beyond the basic requirements, the following enhancements were implemented:

- Support for both compact and space-separated prefix expressions
- Validation of malformed input with detailed exception handling
- A comprehensive set of test cases, including valid, malformed, and edge-case inputs
- Modular design that separates logic, error handling, and file management
- These enhancements were developed to improve the usability and reliability of the program and to anticipate real-world usage beyond the lab requirements.

Conclusion

This lab offered a practical application of stack-based logic and recursive expression evaluation. It required careful attention to input structure, validation logic, and control flow. The final program is efficient, extensible, and thoroughly tested, with a modular and well-documented implementation. It meets and exceeds the assignment expectations in terms of error handling, design clarity, and functional robustness.