Samantha Lane
Data Structures - Lab 2
July 13, 2025

This project uses a list of tokens as the primary data structure to represent prefix expressions. Each token corresponds to either an operator or an operand, which can be a single-letter variable or a multi-digit number. This choice was made for simplicity and efficiency, as it allows straightforward sequential traversal of the expression during recursive parsing. Rather than building explicit trees or using stacks, the program leverages recursion to naturally reflect the hierarchical structure of prefix expressions. This approach aligns well with the grammar of prefix notation, where each operator expects two operands or sub-expressions.

The recursive parsing method makes sense because it directly mirrors the structure of prefix expressions, making the implementation both elegant and easy to follow. Additionally, thorough validation of the input tokens precedes the conversion process, ensuring that invalid characters or mismatched numbers of operands and operators are caught early. This not only prevents runtime errors but also provides informative feedback to the user. To handle different input formats, including both space-separated and compact expressions, a custom tokenizer was developed. This tokenizer effectively manages multi-digit operands and supports flexible input styles, enhancing the program's usability.

From a performance standpoint, the program operates with linear time complexity relative to the number of tokens, processing each token once during tokenization, validation, and conversion. The space complexity is also efficient, limited primarily to storing the tokens and the recursion stack, which corresponds to the depth of the expression tree. The program avoids unnecessary data structures, which keeps both time and space overhead minimal.

Throughout this project, I solidified several key factors. Recursion proved to be a powerful tool for parsing and converting hierarchical expressions, offering a clear and natural way to implement the conversion without additional data structures. The importance of robust tokenization became apparent, especially when handling varied input formats and multi-digit numbers. Validation emerged as a critical step in catching errors early and simplifying debugging. Finally, organizing the code modularly with clear documentation enhanced maintainability and clarity.

If the project were to be revisited, I would consider implementing an explicit parse tree structure to provide greater flexibility for potential future features, such as expression evaluation or transformation. Adding positional information to tokens could improve error reporting precision. Expanding operand support to include floating-point numbers or multi-character variables would make the program more versatile. These enhancements would build on the strong foundation of the current implementation, making it even more robust and user-friendly.