Melbourne Anderson

Project 2

Fall 2024

CMSC 330 Advanced Programming Languages

Approach Used in the Project

The approach I used for this Project was systematic and iterative, aiming to build the program incrementally while adhering to the project specifications and ensuring modularity. The Project required implementing various classes for handling arithmetic and logical operations, parsing expressions, and managing variable assignments. I broke down the requirements into manageable components to achieve this, focusing on one piece at a time. My approach can be summarized in the following steps:

Class Structure and Modularization:
- I first designed and implemented the core classes (`Expression,` `SubExpression,` and their subclasses such as `Plus,` `Minus,` `Negate,` and `Ternary`). The emphasis was on creating a clean class hierarchy that follows the object-oriented principles of inheritance and polymorphism.
- Each operator class was defined with a constructor and an `evaluate` method that adhered to the base class structure, allowing easy integration and extension.

Parsing and Expression Evaluation:
- I implemented parsing functions (`SubExpression::parse,` `Operand::parse,` and `parseAssignments`) that interpreted the expressions and variable assignments based on the provided input format.

Debugging statements were strategically placed to trace the execution flow, particularly during parsing and evaluation. This helped identify and rectify issues as they arose, such as mismanagement of parentheses and variable parsing.

Debugging and Validation:
- Throughout the Project, I used debugging statements extensively to verify that each component behaved as expected. This step-by-step debugging approach allowed me to isolate issues quickly, such as handling edge cases with the unary and ternary operators.
- I validated the implementation by running various test cases, incrementally building complexity from simple arithmetic operations to more complex expressions that included unary and ternary logic.

However, despite these systematic efforts, the Project only ran successfully a few times before further steps were taken to improve the program. Many of these changes failed as I tried to extend and optimize the code, causing cascading issues throughout the codebase. Most of the time was spent trying to repair and revert these changes, which proved challenging due to dependencies between different components.

Lessons Learned from the Project

This Project provided several valuable lessons, particularly in object-oriented design and the nuances of C++ development. It also highlighted the challenges of managing and debugging a complex codebase:

Importance of Modular Design:
- Splitting classes into header and source files improved code organization and readability. This reinforced the importance of modularity, especially when working on a larger codebase where different

components need to interact seamlessly.
   - The modular approach also facilitated testing individual components, allowing me to verify and debug each class independently before integrating them into the more extensive system.

****Debugging and Tracing Execution**:**
   I learned the value of systematic debugging and tracing. By placing debugging statements in strategic locations, such as during parsing and evaluation, I was able to observe the program's behavior step-by-step. This not only helped me identify logical errors but also provided insights into how different parts of the program interacted with each other.
   - Understanding the flow of Execution in C++ was crucial, especially when dealing with pointer-based operations and dynamic memory management.

****Challenges of Code Modifications and Version Control**:**
   - The difficulty of making changes without fully understanding their impact on interconnected components became evident. As modifications were made to optimize and extend the code, several issues often required reverting changes and revisiting previous versions.
   This experience emphasized the importance of version control and regular commits. Detailed commit messages and backup points could have made reverting to a stable state more efficient.

****Managing Complexity in Parsing Expressions**:**
   - One of the challenges was correctly parsing nested and complex expressions while ensuring that the program maintained the correct state. This required an in-depth understanding of recursive functions and how to structure them effectively.
   - I also learned the importance of handling edge cases, such as mismatched parentheses or unexpected operators, which required careful validation at each parsing stage.

****Improvements and Future Enhancements****

If I had more time, I would focus on the following improvements to optimize the codebase and enhance the program's functionality:

 ****Optimizing the Parsing Logic**:**
   While the current parsing logic should be functional, it could be fixed and optimized to reduce complexity and improve efficiency. For example, implementing a more advanced parser that uses a look-ahead mechanism could better handle different expression types and their precedence without excessive backtracking.
   - Additionally, the code could be refactored to use data structures such as stacks to manage operator precedence and parentheses more effectively.

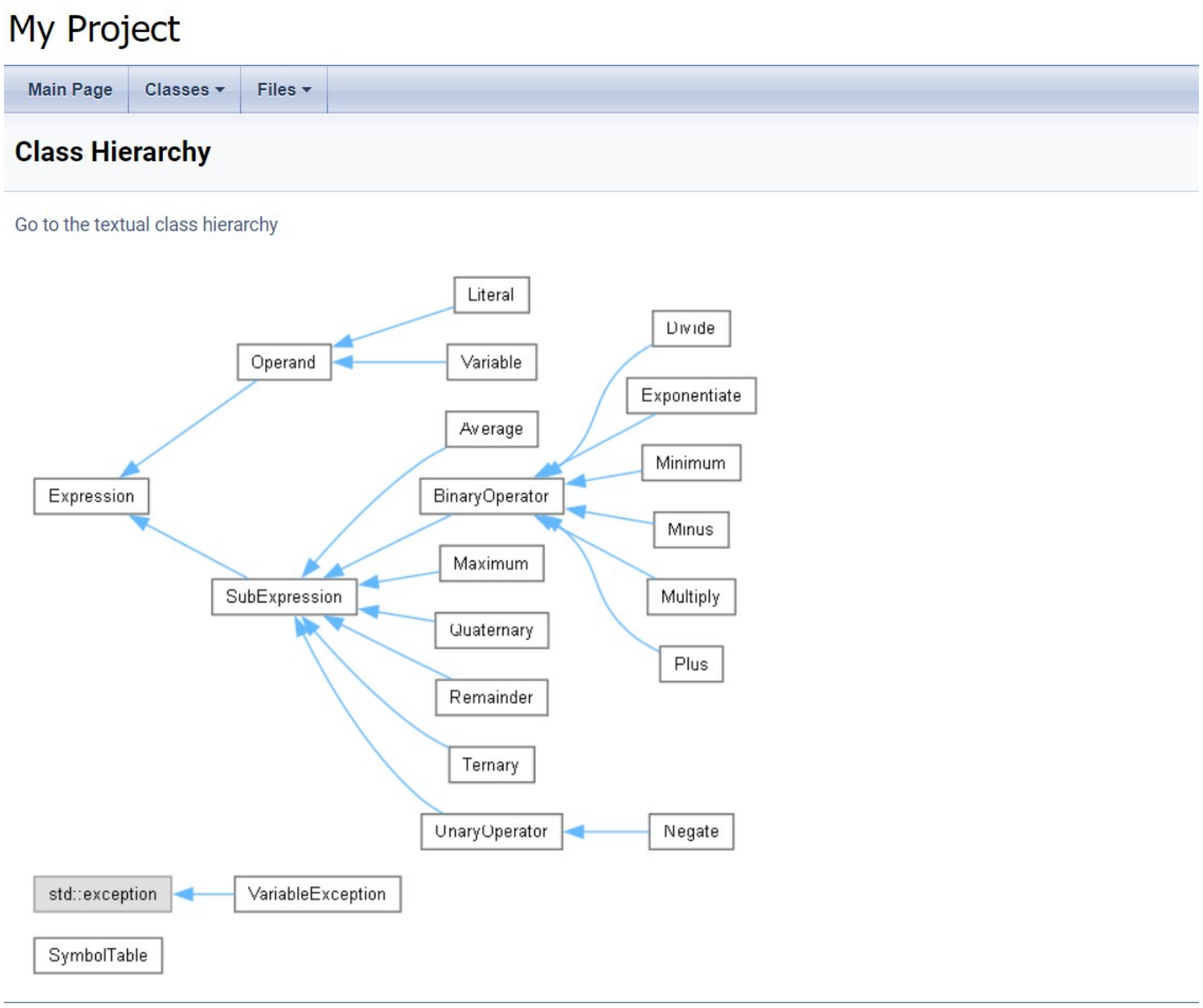
****Refactoring and Extending the Symbol Table**:**
   - The symbol table currently stores variable values and supports basic lookup functionality. However, it could be extended to support additional features, such as scope management, which would benefit expressions involving nested scopes or function-like expressions.
   I would also refactor the symbol table to use more advanced C++ features, such as templates or smart pointers, to optimize memory usage and improve flexibility.

 ****Improving Documentation and Code Readability**:**

- While the code includes debugging statements, adding detailed comments and documentation for each function and class would make the codebase more understandable for future developers.
  Moreover, creating a comprehensive README file outlining the program's structure, design choices, and how to run and test it would further enhance the Project's usability and maintainability.

In conclusion, while the Project provided a solid learning experience in object-oriented design, C++ syntax, and debugging practices, the complexity of changes and their cascading effects made it challenging. The Project highlighted the importance of version control, testing, and modular design to manage and maintain the codebase effectively. With more time and a refined approach, I could likely produce a working program and ensure a more stable release.

Class Hierarchy



See Project 2 Class Diagrams.html – shortcut for full expand.