**CPT-281 Team Project 3A: Binary Tree Infix Expression Parser**

Contributors: Athul Jaishankar, Timothy Huffman, Kathleen Dunn, Tyler Blackmore

Project Summary:

This project is a Binary tree Infix expression parser system that helps parse an infix expression that supports arithmetic and logical operators with specified precedencies. The system utilizes binary trees and stacks for efficient management of expression data.

Technical Requirements:

▪ The Binary Tree Infix expression parser system will support:

Operator Precedence Example

1) Power ( ‘^’ ) 7 2 ^ 8

2) Arithmetic ( ‘\*’, ‘/’, ‘%’ ) 6 6 \* 2

3) Arithmetic ( ‘+’, ‘-’ ) 5 6 - 2

4) Comparison ( ‘>’, ‘>=’, ‘<’, ‘<=’ ) 4 6 > 5

5) Equality Comparison ( ‘==’, ‘!=’ ) 3 6 != 5

6) Logical And ( ‘&&’ ) 2 6 > 5 && 4 > 5

7) Logical Or ( ‘| |’ ) 1 1 | | 0

▪ The infix expression parser is flexible with the given expressions. The user don’t need to worry about writing the spaces between operands and operators

▪ The file that keeps track of the infix expression is a plain text file. An original file input format is made based on this example:

((2 + 3) \* 4) - (5 \* (6 - 7))

(1 | | (0 && 1)) && (1^ ( 1 && 0 ))

(( 2 \*3) ^ 2 ) + ( 4\* 5) % 3

In the example above, each line stores a valid infix expression with appropriate suitable operators and operands.

**System Design:**

**Data Structures:**

**UML:**

A diagram of a tree

Description automatically generated with medium confidence

**Test Cases:**

Test Case #1:

The first input file is shown below:

A black screen with white text

Description automatically generated

The expected output is a list of the evaluation results of these infix expressions. For comparisons and logical and/or, the result is 1 for true, and 0 for false. For example, in the second to last input line on, the left side of the “==” equation evaluates to 15. So then “15 == 15” is true, therefore the output is 1.

The output from this first test case is shown below:

A black screen with white text

Description automatically generated

Test Case #2:

The second input file is shown below:

A black background with white text

Description automatically generated

The expected output again is a list of the evaluation results from the infix expressions above. For division expressions, it will output an integer result. If the infix expression includes a division by zero, the output writes “Error: Divide by zero” to the console.

The output from the second test case is shown below:

A black screen with white text

Description automatically generated

**Team Member Contributions:**

* **Athul Jaishankar:**
* **Build\_Tree.h:** Implemented the Build\_Tree class header file, which defines the class responsible for constructing the expression tree from a postfix string. The class includes necessary header files such as “Tree\_Node.h”, “Expression\_Tree.h”, “Token.h” to support its functionality. Defined the class with constructor, destructor and a method tree\_builder() to build the expression tree.
* **Build\_Tree.cpp:** Contributed to the implementation of the Build\_Tree class in the Build\_Tree.cpp file. Implemented the tree\_builder() method, which iterates through the postfix string to construct the expression tree using a stack-based algorithm.
* **Convert\_to\_postfix.h:** Defined a class called Convert\_to\_postfix with a method infix\_to\_postfix to convert infix expression to postfix notation.
* **Convert\_to\_postfix.cpp:** Implemented the functionalities declared in the header file for the Convert\_to\_postfix class. This implementation includes the constructor and destructor for the class, as well as the infix\_to\_postfix method.
* **Evaluate\_Tree.h:** Defined a class called Evaluate\_Tree with a constructor, destructor and a method Evaluator() to evaluate expression tree.
* **Evaluate\_Tree.cpp:** Contributed to the implementation of the Evaluate\_Tree class in the Evaluate\_Tree.cpp file. Implemented the Evaluator() method which recursively evaluates the expression tree nodes based on their operators and operands, handling arithmetic and logical operations.
* **Project Management:** Took the initiative to lead the project by designing the overall structure and goals of the infix expression parser system. Scheduled and organized team meetings to facilitate communication and collaboration among team members, ensuring smooth progress throughout the project.
* **Task Division:** Effectively divided tasks among team members, assigning responsibilities for coding, testing and documentation.
* **Testing:** Collaborated with team members to create test cases covering various expressions and scenarios. Verified the correctness of the program by comparing the actual output with the expected output.
* **Quality Assurance:** Ensured code quality by writing clean, well-commented code with meaningful variable names and function names. Maintained an organized repository structure and adhered to coding standards to facilitate code review and future maintenance. Effectively divided tasks among team members, assigning responsibilities for coding, testing and documentation.
* **Timothy Huffman:**
* **Expression\_Tree.h:** Developed the Expression\_Tree class header file, which defines the class responsible for handling expression parsing, evaluation and related operations. Defined the Expression\_Tree class with a constructor, destructor and methods for parsing infix expression, evaluating postfix expression, calculating operator precedence and handling errors.
* **Expression\_Tree.cpp:** Implemented the parse\_and\_evaluate() method, which parses infix\_expression into postfix notation using Convert\_to\_postfix class and evaluates the resulting expression tree using the Evaluate\_Tree class. Created the precedence() method to determine the precedence of operators and power\_function() method to calculate the exponentiation.
* **System Design Explanation:** Provided insights into the overall system design in the project report. Explained the architecture and structure of an binary tree infix expression parser system, ensuring clarity and coherence in the documentation.
* **Data Structures Explanation:** Detailed the role of data structures used in the binary tree infix expression parser system. Explained how each data structure contributed to efficient expression parsing and evaluation.
* **In-Line Comments**: Added in-line comments to the Expression\_Tree.h and Expression\_Tree.cpp files, improving code readability and comprehension of team members.
* **Meeting Attendance and Questions**: Actively attended team meetings, contributing to discussions on project progress and asking follow-up questions to clarify requirements or resolve issues effectively.
* **Kathleen Dunn:**
* **Tree\_Node.h:** Defined the Tree\_Node struct, including the data fields such as “Token data”, “Tree\_Node\* left”, “Tree\_Node\* right”, to store the token and pointers to the left and right children.
* **Tree\_Node.cpp:** Provided the constructor definition, initializing the data, left and right pointers. Ensured proper encapsulation by including the “Tree\_Node.h” header file.
* **Token.h:** Defined the Token class, including the enumeration “Type” for distinguishing between operator and operand tokens. Provided the constructor, getter methods and private data members.
* **Token.cpp:** Contributed the constructor definition, initializing the Token\_type and value. Implemented the get\_type() and get\_value(), ensuring proper encapsulation of the class’s data.
* **Test Cases:** Responsible for creating test cases to validate the correctness of the infix expression parser program. Ensured that the test cases covered various expressions and scenarios, documenting them in the project report for future reference.
* **Program Correctness:** Verified the correctness of the program by executing the test cases and comparing the actual output with the expected output.
* **Meeting Attendance and Questions**: Actively participated in team meetings, providing valuable input on system design, discussing project progress and asking follow-up questions to clarify requirements or resolve issues effectively.
* **Tyler Blackmore:**
* **Error Handling in Expression Tree:** Implemented the “handle\_error()” function within the Expression\_Tree class. This method serves to handle errors that may occur during the construction or evaluation of expression trees. By implementing error handling mechanism, Ensured that the project can gracefully handle unexpected situations and provide informative error message to the user, enhancing the reliability and user experience of the software.
* **Main.cpp:** Implemented the logic to read infix expression from an input file, parse and evaluate each expression using Expression\_Tree class and displaying the result to the console. Integrated file I/O operations for input file handling. Implementing error handling to detect and notify the users if the input file cannot be opened. Ensuring robustness and reliability of the program.
* **Future Requirements**: Contributed nine ideas for future improvements to the binary tree infix expression parser system. These ideas were aimed at enhancing the functionality and usability of the system. Documented these suggestions in the project report to guide future development efforts.
* **Meeting Attendance and Questions**: Actively participated in team meetings, providing valuable input on system design, discussing project progress and asking follow-up questions to clarify requirements or resolve issues effectively.

**Future Improvements:**