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 binary tree 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:

The Binary Tree Expression parser was made in order to efficiently convert infix to postfix expression, Build an expression tree based on the postfix expression and evaluate the expression tree. This system uses five classes which are Expression_Tree, Convert_to_postfix, Build_Tree, Evaluate_Tree, Token.

- **Expression tree class:**

  The Expression tree class is used in order to convert, build and evaluate an infix string into a postfix string, building a binary tree based on the postfix expression and returning the result by using the parse_and_evaluate() function. The function precedence() serves as the purpose of determining the precedence level of operators, which is crucial for correctly parsing and evaluating mathematical or logical expressions. The power_function() which will be called whenever the power operator is parsed.

- **Convert to postfix class:**

  In order to evaluate an infix expression, it first needs to be converted to a postfix expression. The Convert to postfix class converts an infix expression into a postfix expression with the use of its only class-member function, infix_to_postfix(), using a stack to iterate through the infix string appending and popping when needed.

- **Evaluate tree class:**

  The Evaluate tree class is where the expression tree, built using the postfix expression, will be evaluated. The prominent function in this class is the Evaluator(). With the use of binary trees, the left subtree is first evaluated, followed by the right subtree evaluation. Digits are then stored in the tree while processing tokens appending and popping when needed.

- **Build tree class:**

  The Build tree class is responsible for constructing a binary tree representation of a postfix expression. The tree_builder() function takes a postfix expression string as input and builds a binary tree of the expression. It return the root node of the constructed binary tree.

- **Token class**

  The Token class represents a token in the expression, which can be either an operand or an operator. The class has a constructor that initializes the Token_type and value members. It has getters functions for returning the token type and value. Token class is used to construct the binary tree in the Build_Tree class.

Data Structures:

- **Binary Tree:**

  The system developed uses the data structure binary tree, the binary tree data structure can be seen in the Build_Tree & Evaluate_Tree classes within the tree_builder() & Evaluator() function.

  - **Tree_builder():**

    The tree_builder() function utilizes a binary tree structure to build an expression tree from a postfix expression. It iterates over each token in the postfix expression and constructs the expression tree accordingly.

  - **Evaluator():**

    The Evaluator() function evaluates the expression stored in the binary tree. It traverses the tree recursively, performing arithmetic, logical or comparison operations as dictated by the operators encountered. Operand values are retrieved from the leaf nodes and operations are applied based on the operators stored in internal nodes.

- **Stack:**

  The system developed uses the data structure stacks , the stacks data structure can be seen in the Build_Tree & Convert_to_Postfix classes within the tree_builder() & infix_to_postfix() function.

  - **Tree_builder():**

    In the tree_builder() function, a stack of integers is used to store operands during evaluation of a postfix expression. Operands are pushed onto the stack, and when operators are encountered, they are applied to the operands popped from the stack.

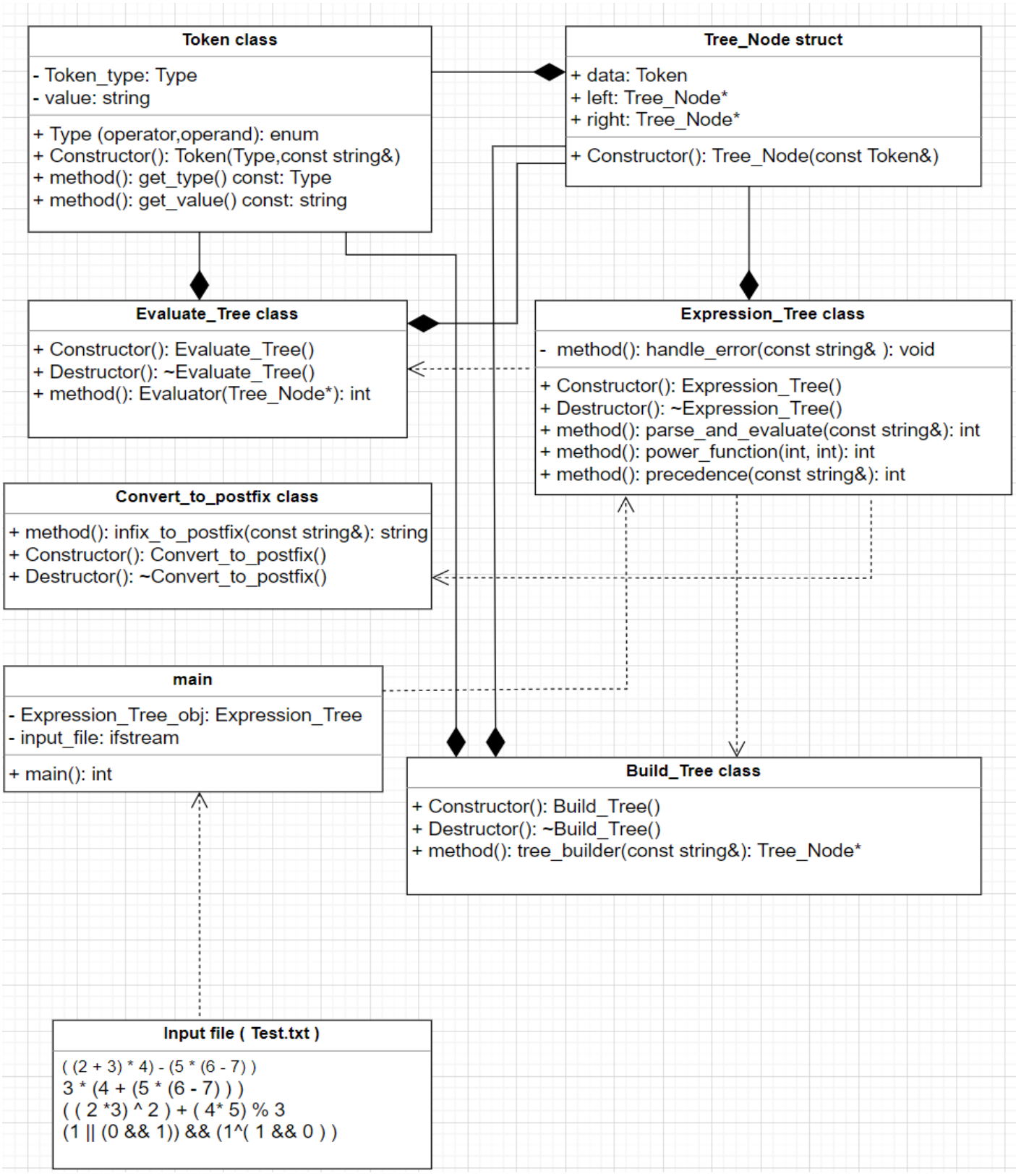
  - **Infix_to_postfix():**

    The infix_to_postfix() function uses a stack of strings to store operands while converting an infix expression to a postfix expression. The infix string is parsed pushing operations onto the stack based on the rules given and popping the elements when needed. After processing, remaining elements in the stack are appended to an output string.

- **String:**

  The String data type was used to store the infix expressions, when converting to postfix expressions.

UML:


Token class
- Token_type: Type
- value: string
+ Type (operator,operand): enum
+ Constructor(): Token(Type,const string&)
+ method(): get_type() const: Type
+ method(): get_value() const: string
Tree_Node struct
+ data: Token
+ left: Tree_Node*
+ right: Tree_Node*
+ Constructor(): Tree_Node(const Token&)
Evaluate_Tree class
+ Constructor(): Evaluate_Tree()
+ Destructor(): ~Evaluate_Tree()
+ method(): Evaluator(Tree_Node*): int
Expression_Tree class
- method(): handle_error(const string&): void
+ Constructor(): Expression_Tree()
+ Destructor(): ~Expression_Tree()
+ method(): parse_and_evaluate(const string&): int
+ method(): power_function(int, int): int
+ method(): precedence(const string&): int
Convert_to_postfix class
+ method(): infix_to_postfix(const string&): string
+ Constructor(): Convert_to_postfix()
+ Destructor(): ~Convert_to_postfix()
main
- Expression_Tree_obj: Expression_Tree
- input_file: ifstream
+ main(): int
Build_Tree class
+ Constructor(): Build_Tree()
+ Destructor(): ~Build_Tree()
+ method(): tree_builder(const string&): Tree_Node*
Input file (Test.txt)
((2 + 3) * 4) - (5 * (6 - 7))
3 * (4 + (5 * (6 - 7)))
((2 *3) ^ 2) + (4* 5) % 3
(1 || (0 && 1)) && (1^(1 && 0))

Test Cases:

Test Case #1:

The first input file is shown below:

```
((2 + 3) * 4) - (5 * (6 - 7))
3 * (4 + (5 * (6 - 7)))
(( 2 *3) ^ 2 ) + ( 4* 5) % 3
(1 || (0 && 1)) && (1^( 1 && 0 ))
((2 >= 3) && (4 < 5)) || ((6 == 7) || (8 != 9))
10 / (5 - 5)
10 / (3 * 0)
2 ^ (3 ^ 2)
2 ^ 3 ^ 2
((2 ^ 3) ^ 2)
( ( ( ( ( (3 ^ 2) * 2) / 3) % 2 ) + 20) - 5) == 15
(1 || 0) && 0
```

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:

```
The Result is: 25
The Result is: -3
The Result is: 38
The Result is: 1
The Result is: 1
Error: Divide by zero
Error: Divide by zero
The Result is: 512
The Result is: 64
The Result is: 64
The Result is: 1
The Result is: 0

C:\Project_3\Project-3A-AJ-Timothy-Katie-Tyler\Binary Tree Expression Parser\x64\Debug\Binary Tree Expression Parser.exe
 (process 4868) exited with code 0.
Press any key to close this window . . .
```

Test Case #2:

The second input file is shown below:

```
1+2*3
2+2^2*3
1==2
1+3 > 2
(4>=4) && 0
(1+2)*3
2%2+2^2-5*(3^2)
(3 + 4) || 1
(2 > 3) - 2
5 ^ 2 % 7 && (4 - 4)
3 / (6 * 5 - 30)
```

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:

```
The Result is: 7
The Result is: 14
The Result is: 0
The Result is: 1
The Result is: 0
The Result is: 9
The Result is: -41
The Result is: 1
The Result is: -2
The Result is: 0
Error: Divide by zero

C:\Project_3\Project-3A-AJ-Timothy-Katie-Tyler\Binary Tree Expression Parser\x64\Debug\Binary Tree Expression Parser.exe
 (process 19576) exited with code 0.
Press any key to close this window . . .
```

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.
- **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 two 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.

Future Improvements:

- **Support for more data types:**

  Extend the parser to support a wider range of data types beyond just integers. This could include floating-point numbers, strings, or even-defined data types, thereby increasing the versatility of the parser.

- **Memory Management:**

  Make sure memory usage is optimized, especially when dealing with larger expressions. Techniques like object pooling or even smart pointers can efficiently manage memory while avoiding any unnecessary allocations or deallocations.

- **Enhanced Operator Support:**

  Introduce support for additional operators or functionalities, such as mathematical functions (e.g., square root, exponentiation), bitwise operations, or custom-defined operators. This would broaden the capabilities of the parser and make it more adaptable to diverse use cases.

- **Error Handling and Reporting:**
  Implement more robust error handling mechanisms to handle various edge cases gracefully. Provide informative error messages to users to aid in debugging and understanding issues.

- **Database Integration:**
  Integrating a database system into the application to store and manage expression data, user preferences and evaluation results. This integration can provide advantages such as persistence, scalability and data management capabilities.

- **History Session:**
  Implementing a history and session management would allow the users to refer back to previous calculations.

- **Graphical User Interface (GUI):**
  Developing a GUI application to provide a user-friendly interface for inputting expressions, displaying the results and possibly visualizing the parsing, building and evaluating process. A GUI can improve accessibility and usability, especially for users less familiar with command-line interfaces.

- **Cross-Platform Compatibility:**
  Ensure compatibility with multiple platforms (e.g., Windows, macOS, Linux) by optimizing code and leveraging platform-agnostic libraries or frameworks. This would maximize the reach of the parser and make it accessible to a broader user base.