**A Symbolic Algebra Calculator: Calculating Abstract Syntax Tree**

**Definitions**

The problem to be solved by this program is to create a Symbolic Algebra Calculator using an Abstract Syntax Tree as a key data structure. Solving this problem takes in tokenized input and produces an output. This Abstract Syntax Tree structure will need to be a postfix representation of an algebraic expression. The type of tree structure will need to be determined between an Array-Based Representation or a Link-Based implementation. The program will need to be able to handle numbers and variables and be able to substitute variables with their assigned expression if one is available. The result from these evaluations can range from a number to no change if there is a variable in the expression with no known assigned expression. Thus, the expression would just be outputted or if substitution is possible, then variables will be replaced with their assigned expression and the updated expression will be processed according to the previously indicated rules. The program should be able to process the following expressions, [+, -, /, \*, ^] and following their given precedence as well interpret parenthesis that modify precedence.

An example of what a possible Abstract Syntax Tree should look like in this program for the expression, (a+b)\*(c\*(d+e)) is as follows:

Abstract Syntax Tree

Shape, circle

Description automatically generated

**Specifications**

The goal of this program is to develop a functional Symbolic Algebra Calculator utilizing Abstract Syntax Tree data structures. This will employ a Calculator class which will act as a frontend to the Symbolic Algebra Calculator program and hide all other classes. Some key features the program needs to achieve are to read input and tokenize it, store these tokens in a multidimensional std::vector<std::vector<Token>> currentTokens\_ that is decided by the newline char, “\n”. This will be achieved by using a class called ITokStream which will create Tokens using a Token class.

The input will end when a “.” char is read which represents the end of our input, the program will call a method called calculateExpressions in the Calculator class. In this method the vectorized tokens will be echoed back to the user. After each initial echo back to the user, the program will convert a given token vector to postfix form from the original infix form that the expression was provided.

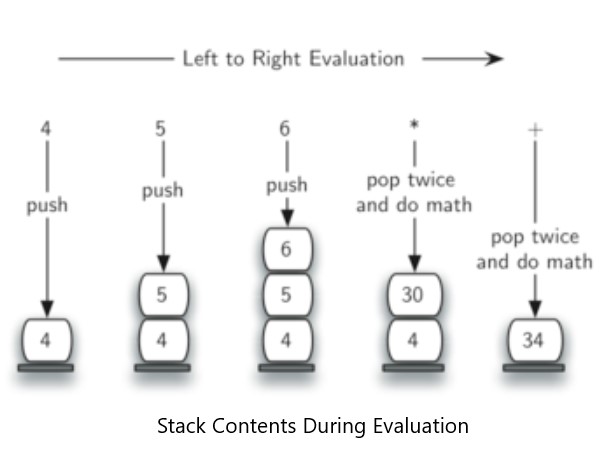
After the conversation has taken place, the system will build an AST (Abstract Syntax Tree) data structure. However, before the tree is constructed the expression that is stored in a vector will be checked if there is an assignment. If there is a variable assignment the program will store the variable and the half of the expression after assignment operator for later use. Additionally, the vector will be modified to remove the variable and the assignment operator before the AST tree is built. After this the AST constructor will call a method called build which will process the provided Token vector. This process will employ a std::stack<Node\*> nodeStack; and build the AST using a iterative/stack approach instead of recursion. The style on insertion into AST will be postfix and will mimic the AST image under the definition section.

Now that the program has built the AST for a given expression, a method called simplify will be called to simplify the expression if possible using a std::map called variableStore\_. If the expression can be simplified, then the recursive simplifyHelper method will recursively search for the given variable and replace it using a recursive replace method that replaces/inserts the AST that is represented by that variable.

Now that our expression has been possibly simplified, the calculateExpressions method in our Calculator class will need to call the containsVariable method in our AST class to determine if the expression can be calculated or not. If the expression has a variable, it cannot be calculated and the toInfix method of the AST class will be called to display the expression. Otherwise, the expression will be calculated.

Now that we’ve gone over the general flow of the program, we can discuss how the calculation might be achieved. The calculate method in the AST class will be a simple method that has a Token stack called answerStack that will hold the answer/result from the recursive calculateHelper method. The answerStack and the root\_ of the AST tree will be passed to this helper method. An additional method called doMath will be called inside the calculateHelper which will do the actual math.

You can see the general flow of the stack below:



**Input**

The Input of the program will be handled by the class ITokStream, known as a Tokenizer. This class will break down the input stream of characters into a stream of tokens by employing the Token class. The overload insert method in the ITokStream class will employ the following methods, std::istream::get, isdigit, isapha, and a helper method called determineTokenType.

The input process will begin by calling a getInput method in the Calculator class which will us the ITokStream class to insert tokens into vectors. A new vector will be made for each expression. This will be done by creating a new vector each time a “\n” char is hit. Input will stop when the “.” char is read. The validity of the syntax will be checked after each vector has been constructed. Invalid expressions will be discarded. If an expression has an assignment, the vector will be altered, and the variable and expression will be stored in the variableStore\_ data structure.

The input will be displayed as follows

5 + 7

x := 1

x + 8

z := x + y

y := 8

z

.

**Output**

Output of the for the program will be handled by the calculateExpressions method in the Calculator class which will employ/call other methods that will echo the user’s input back, convert the user input to postfix form, build an AST per the provided expression with a variable check, if there is a variable output the expression in infix form otherwise calculate and display the result/answer to the expression.

The output will be displayed as follows:

in [1]: 5 + 7

out [1]: 12

in [2]: x := 1

out [2]: 1

in [3]: x + 8

out [3]: 9

in [4]: z := x + y

out [4]: 1 + y

in [5]: y := 8

out [5]: 8

in [6]: z

out [6]: 9

**Error Handling**

* The program must verify if a valid expression was passed through the ITokStream. The created vector will be examined to confirm it’s a valid expression. If the expression is invalid an error message will be printed to the screen indicated that a valid expression was provided and has been discarded.
* The program can only maintain 26 variables in the variableStore\_ as such all tokens that represent a variable will automatically be converted to lower case for processing. If a number is provided as a variable “ 2 := 2 + 5” and error message will be printed out informing the user.

**Design**

This program is composed of multiple classes. The program will employ the following classes: ITokStream, Token (Struct), AST, Calculator, and a nested Node Struct in the AST class.

**Token**

This struct is a simple data object that is will be utilized thought the program. It will be the structure/variable in our Node struct that will represent a value and a type.

* **+constructor.** The Constructor will initialize a Token object with the default values of TokType::unknown for TokType type\_ and empty string for std::string value\_ if values are not provided.

Text

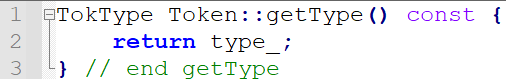
Description automatically generated

* **+getValue():** std::string. Return the string value stored in value\_.

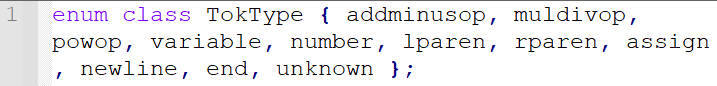
A picture containing chart

Description automatically generated

* **+getType(): TokType.** Returns the TokType value stored in type\_.



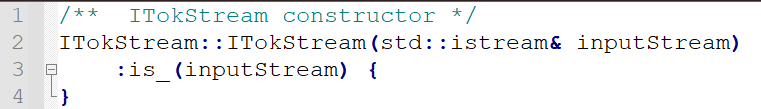
* **+type\_: TokType.** Holds the token’s value type.
* **+value\_: std::string.** Holds the token’s string value.
* **+TokType: <<enumeration>>.** This enum class holds the following items: addminusop, muldivop, powop, variable, number, lparen, rparen, assign, newline, end, unknown. It is used to determine value to determine the type of Token inputted into the system.



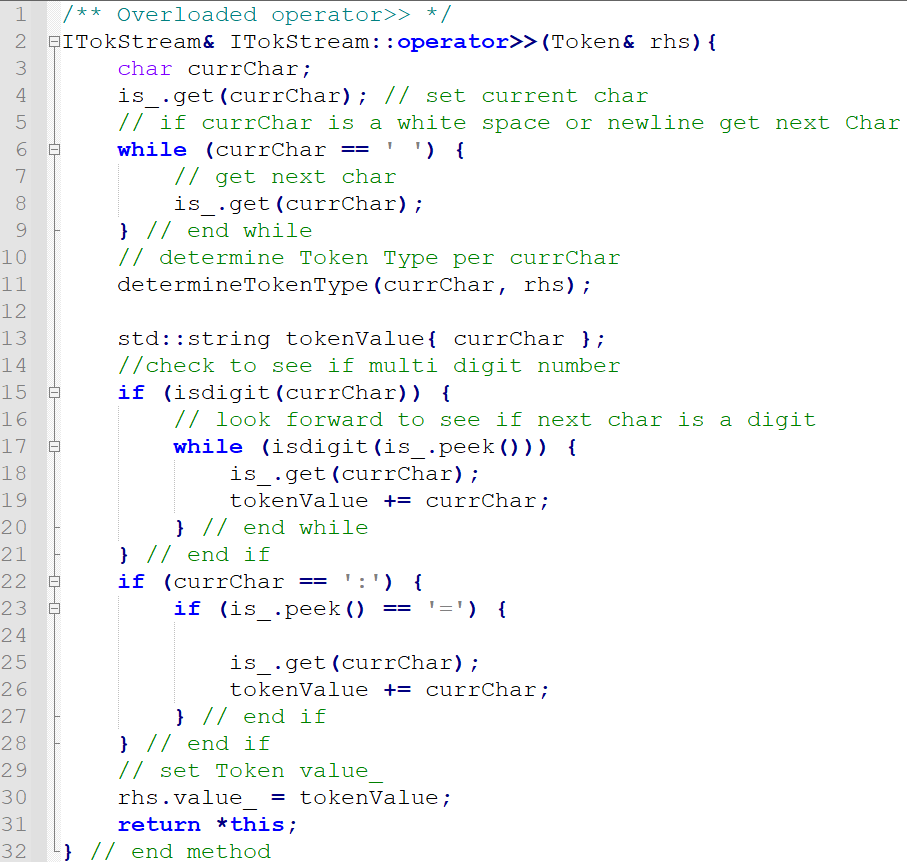
**ITokStream**

The ITokStream class purpose is to tokenize the input stream into manage Tokens for the Symbolic Algebra Calculator program.

* **+constructor.** The Constructor initializes the std::istream& is\_ variable to the provided std::istream.



* **+operator>>(rhs: Token&):** ITokStream&. Returns the ITokStream (\*this).

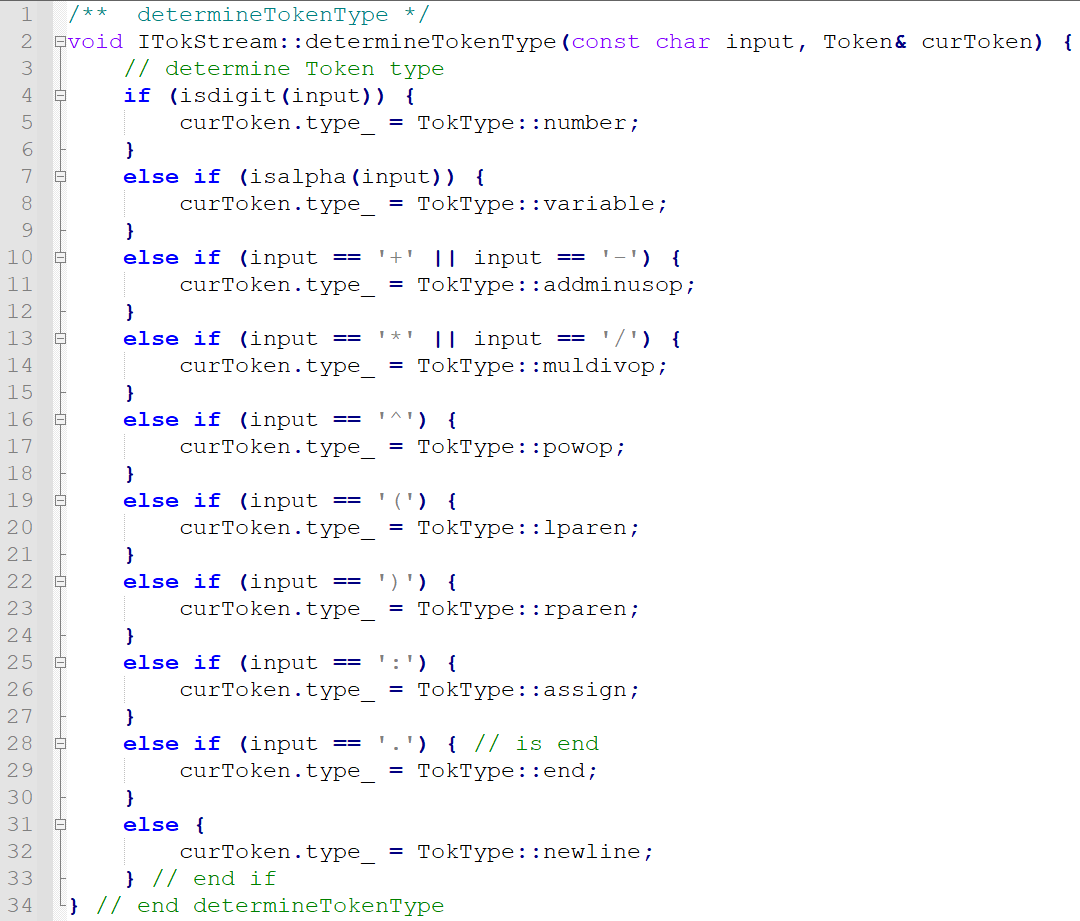


* **+operator bool().** Returns whether there is an error on the input stream.

Text

Description automatically generated

* - determineTokenType(input: char, curToken: Token): void. Assigns TokType to the Token Object.



* -is\_: std::istream&. Stores the reference to the provided std::istream.

AST

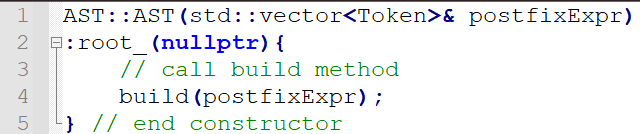
This class is the data structure class of the Symbolic Algebra Calculator. This class will do the bulk of the actions that can take place in the program and is an essential component. The implementation will be a Link-Based implementation of a tree structure.

* **+AST().** Default constructor for the AST class. Set’s the Node\* root\_ attribute to the nullptr.

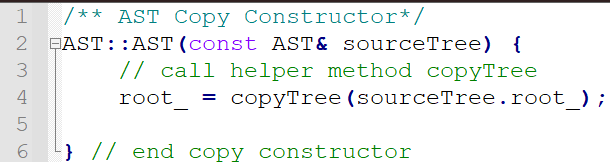
Text

Description automatically generated with medium confidence

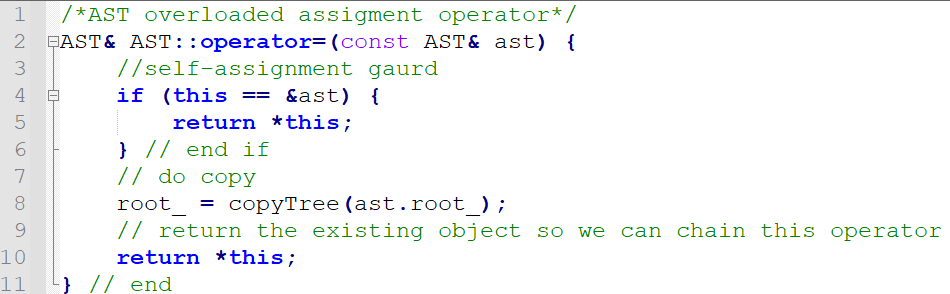
* **+AST(postfixExpr: std::vector<Token).** Additional constructor for the AST class that constructs AST based off the provided Token vector.



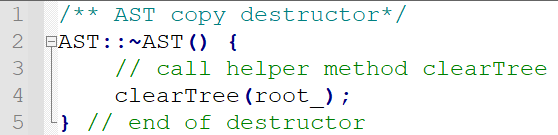
* **+AST(sourceTree: AST).** This is a copy constructor for the AST class. Will utilize a helper method called copyTree.



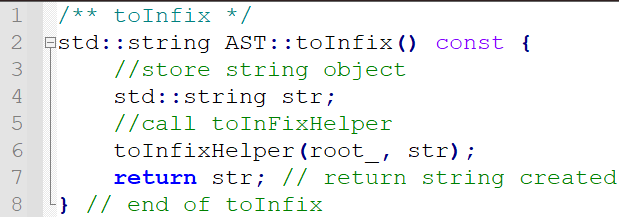
* **+operator=(ast, AST):** AST&. Returns a reference to an AST object.



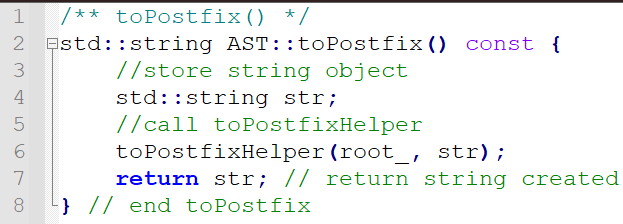
* **+~AST().** Destructor for the AST class. Class the clearTree method to destroy the tree object and release the memory.



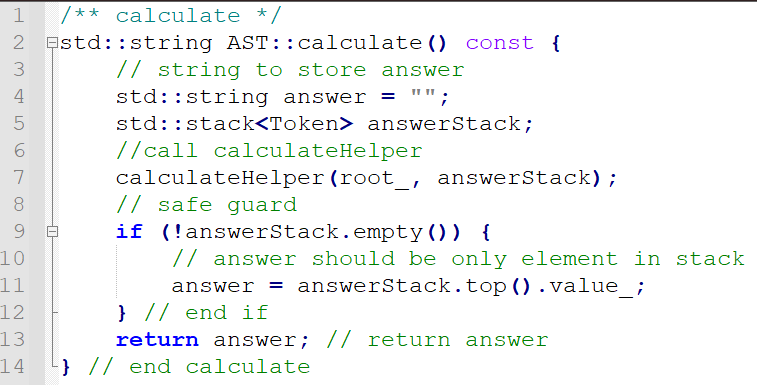
* **+toInfix(): std::string.** Returns a string object that represents an infix expression of the AST object. Calls the toInfixHelper method.



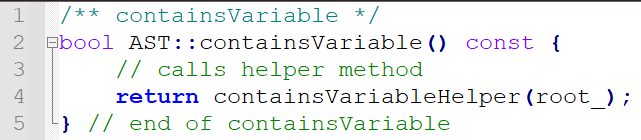
* **+toPostfix(): std::string.** Returns a string object that represents an postfix expression of the AST object. Calls the toPostfixHelper method.



* **+calculate(): std::string.** Returns a string object that represents that answer/result of the evaluated expression. Calls the calculateHelper method to achieve this.



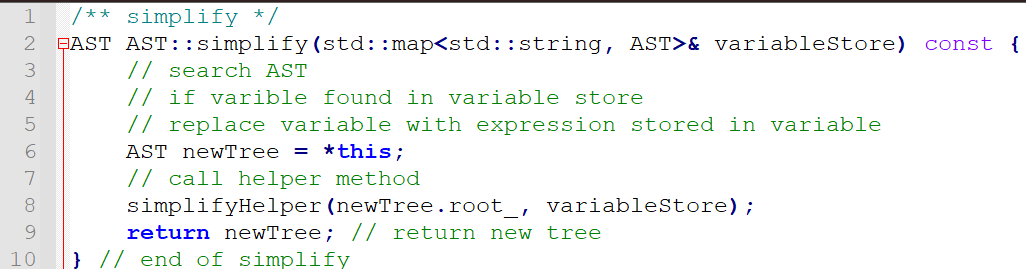
* **+containsVariable(): bool.** Returns true if the AST contains a variable type Token, otherwise false. Calls the containsVariableHelper method to do this.



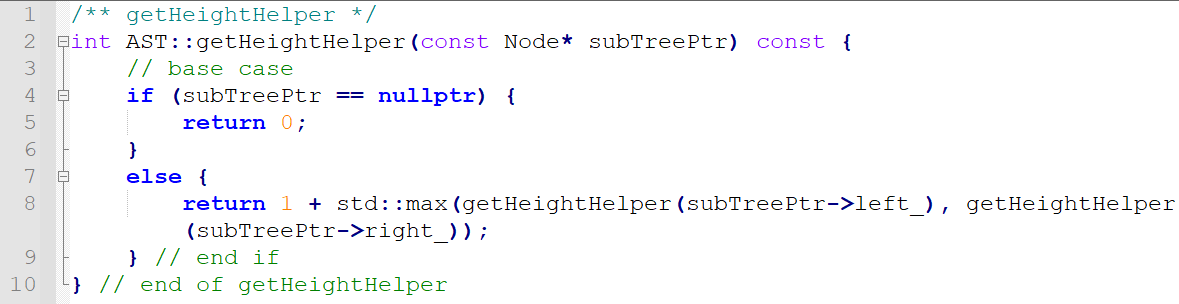
* **+build(tokensToAdd: std::vector<Token>): bool.** If the build operation is successful, it returns true, otherwise false. Calls the isOperator method.



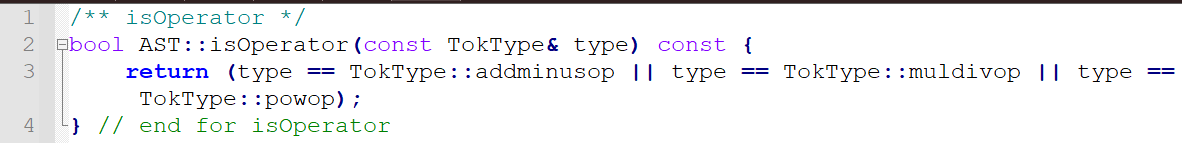
* **+simplify(variableStore: std::map<std::string, AST>): AST.** Returns a new tree that is a simplified version of the previous tree. This method utilizes the overloaded assignment operator and simplifyHelper method.



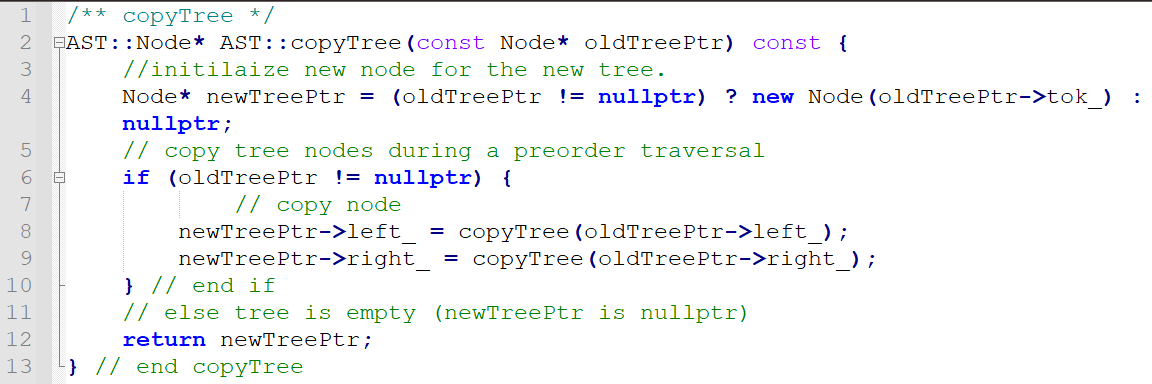
* **-getHeightHelper(subTreePtr: Node\*):** int. Returns the level/height of a particular node as a integer value.



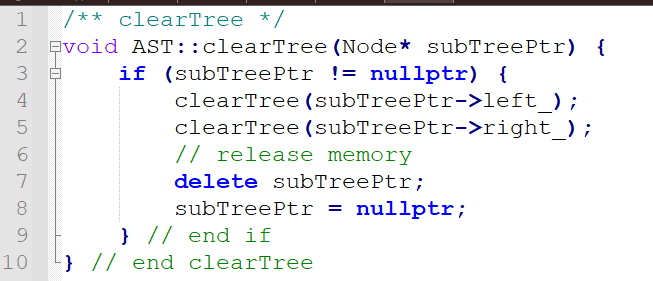
* **-isOperator(type: TokType): bool.** Returns true if the type is an operator, false otherwise.



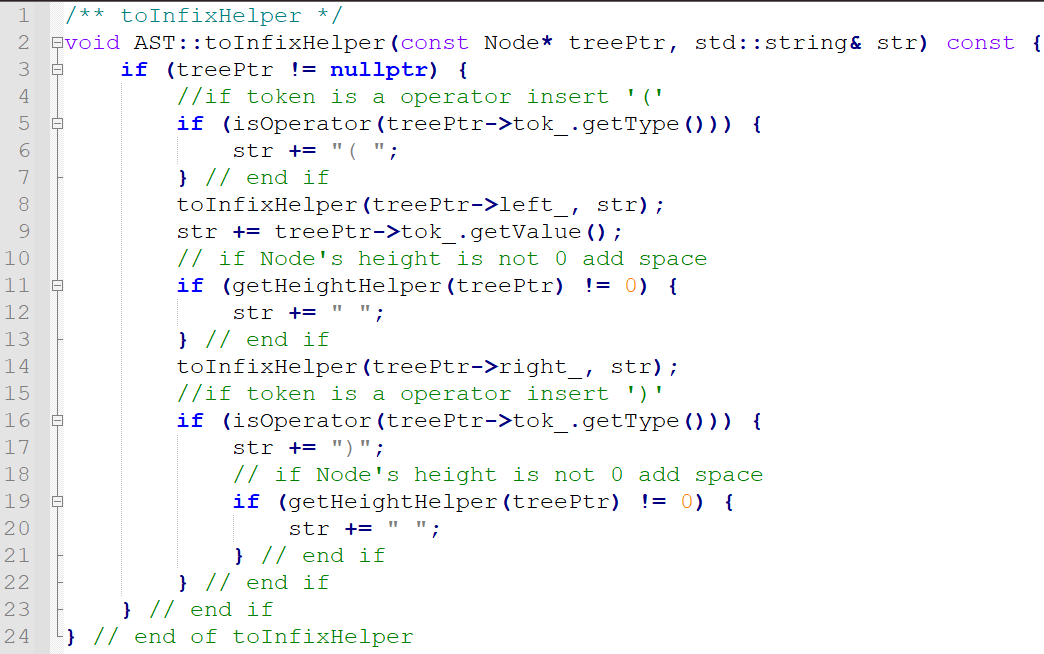
* **-copyTree(subTreePtr: Node\*): Node\*.** Returns a node pointer to the newly created tree that was copied from the source tree.



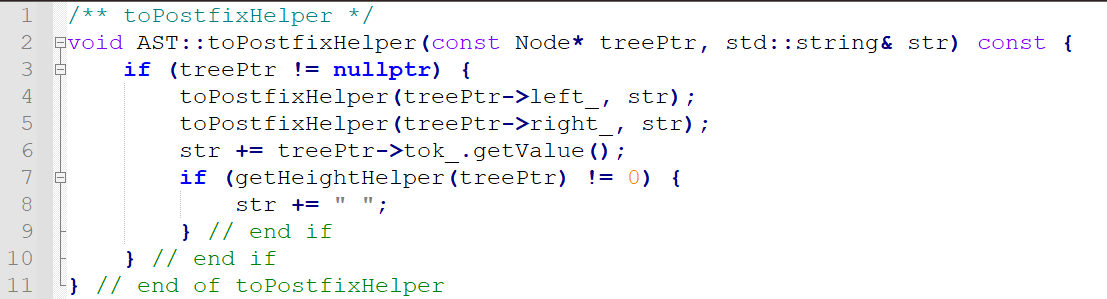
* **-clearTree(subTreePtr: Node\*): void.** Destroys the provided tree using a post order traversal to delete each node an release the memory.



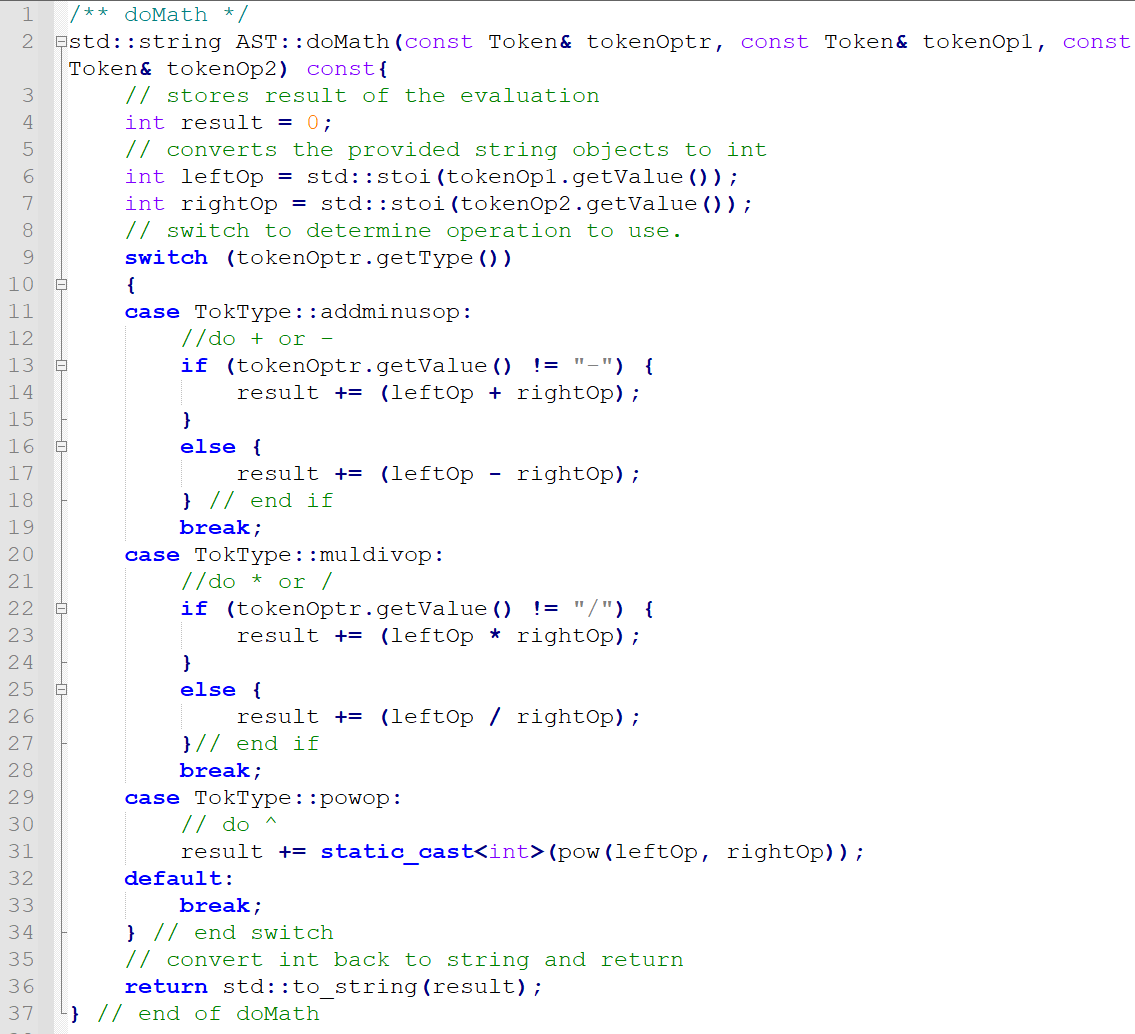
* **-toInfixHelper(treePtr: Node\*, str: std::string): void.** Returns a recursively created infix string presentation of the provided tree.



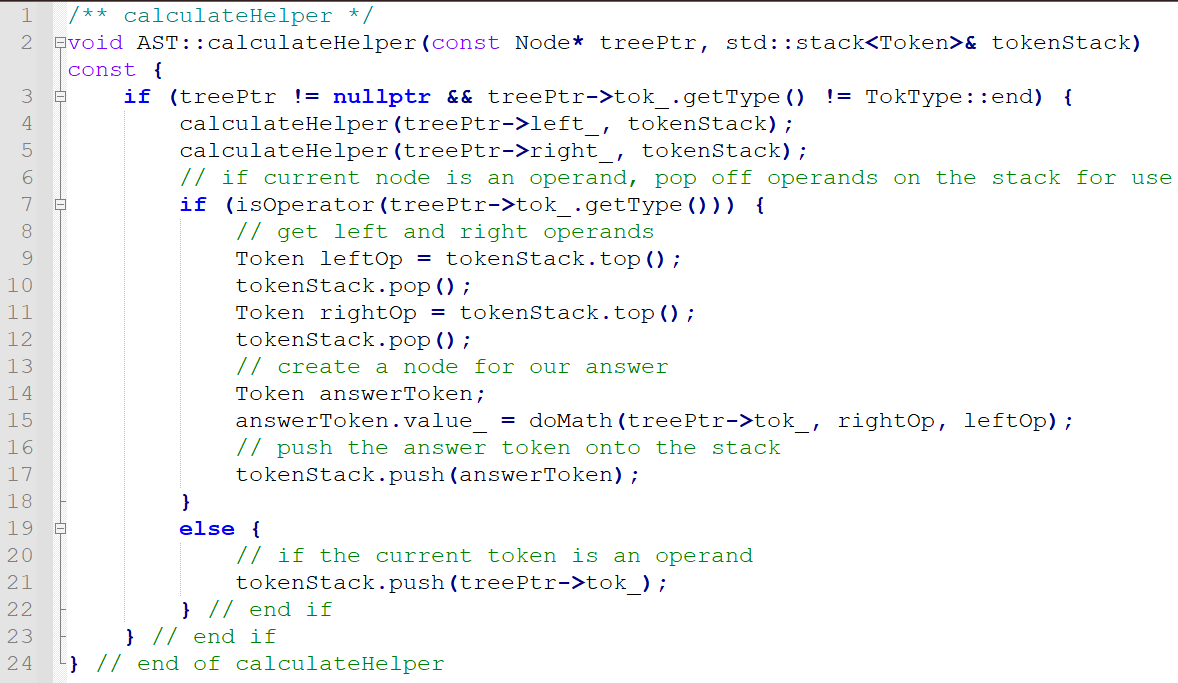
* **-toPostfixHelper(treePtr: Node\*, str: std::string): void.** Returns a recursively created postfix string presentation of the provided tree.



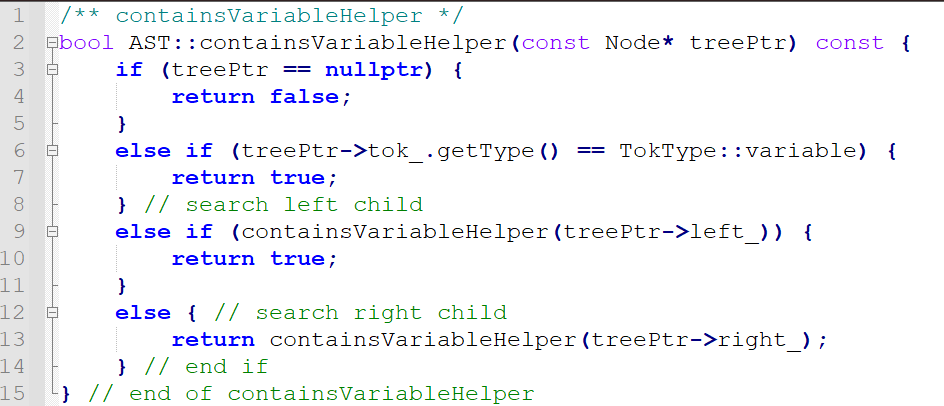
* **-doMath(tokenOptr: Token, TokenOp1: Token, TokenOp2: Token): std::string.** Returns the string object that represents the mathematical result of the provided operands and operator.



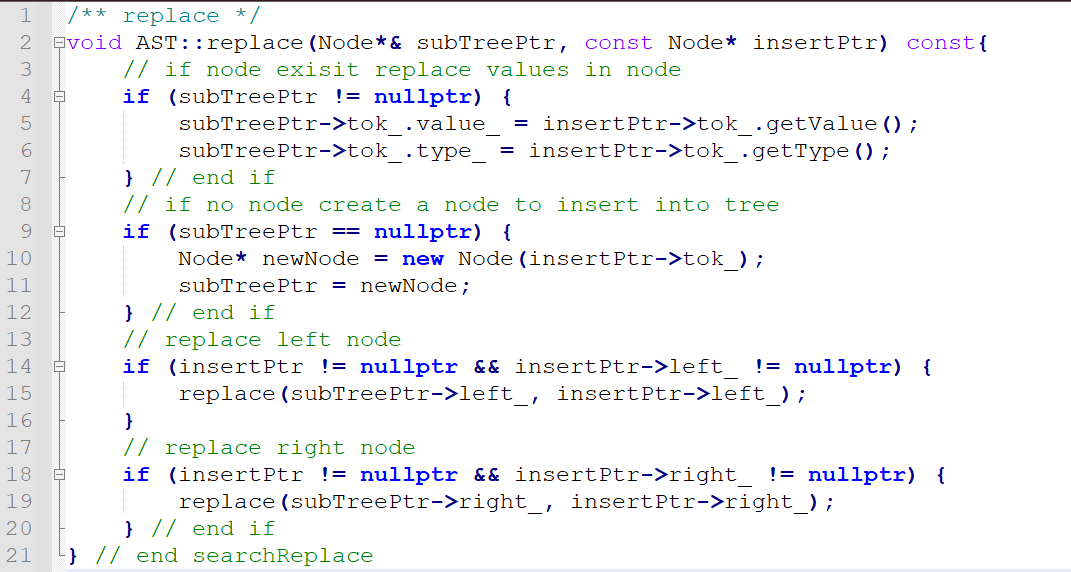
* **-calculateHelper(treePtr: Node\*, tokenStack: std::stack<Token>): void.** This method recursively traverses the tree to evaluate the expression. Uses a stack during this traversal to save operands and pop them off the stack when a operator is encountered and calls the doMath method and pushes the answer back onto the stack.



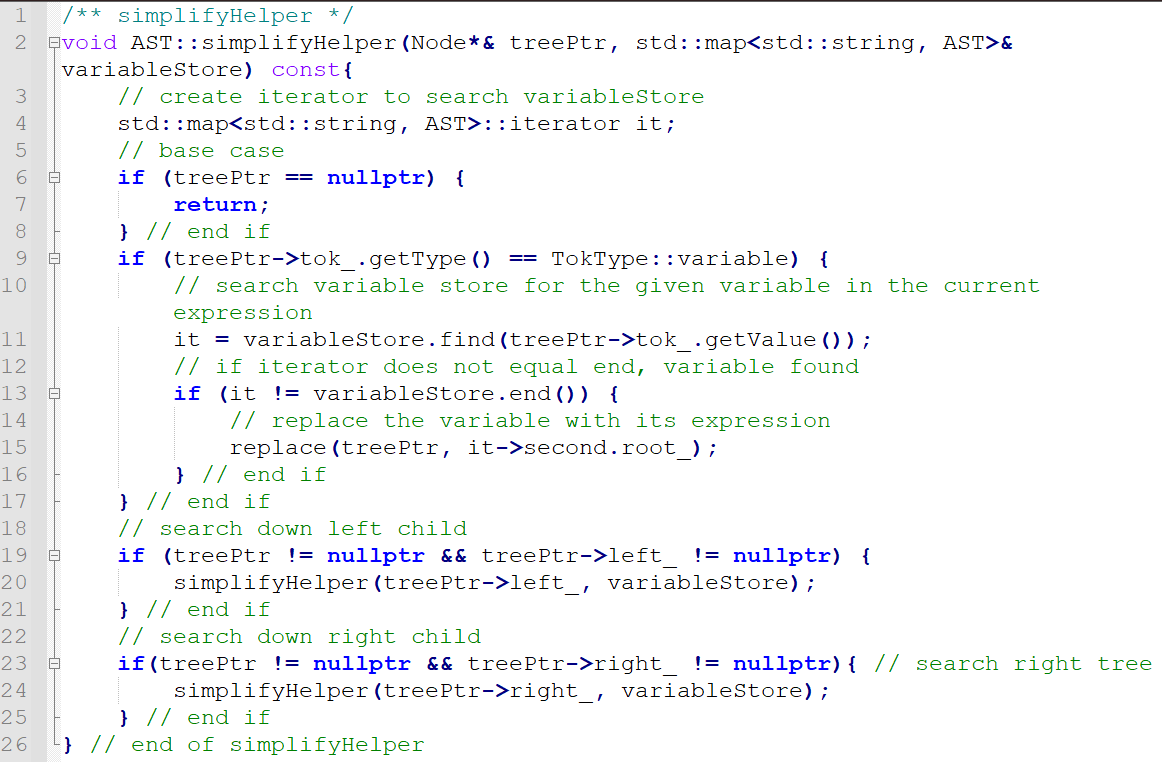
* **-containsVariableHelper(treePtr: Node\*): bool. Returns true if a variable is present in the AST object, false otherwise. This is done recursively.**



* **-replace(subTreePtr: Node\*, insertPtr: Node\*): void. Replaces/inserts a tree at the given node position. This method is use to substitute variables with their expressions in storage.**



* **-simplifyHelper(treePtr: Node\*, variableStore: std::map<std::string, AST>): void.** This method updates expression by searching for variables, if a variable is found then the variableStore is checked for this variable if found the replace method is called.



* **-root\_: Node\*.** This variable represents the root of the tree.

**Node**

The Node struct is a simple data structure that is a private member of the AST class and the Abstract Syntax Tree data structure is made of Nodes linked together.

* **+Node(t: Token).** Constructor for the node class. This class has no default constructor.

Text

Description automatically generated

* **+tok\_: Token.** This variable is a Token which is made up of TokType type\_ and a std::string value\_. This is the data object of the Node class.
* **+left\_: Node\*.** This variable is the left child pointer.
* **+right\_: Node\*.** This variable is the right child pointer.

**Calculator**

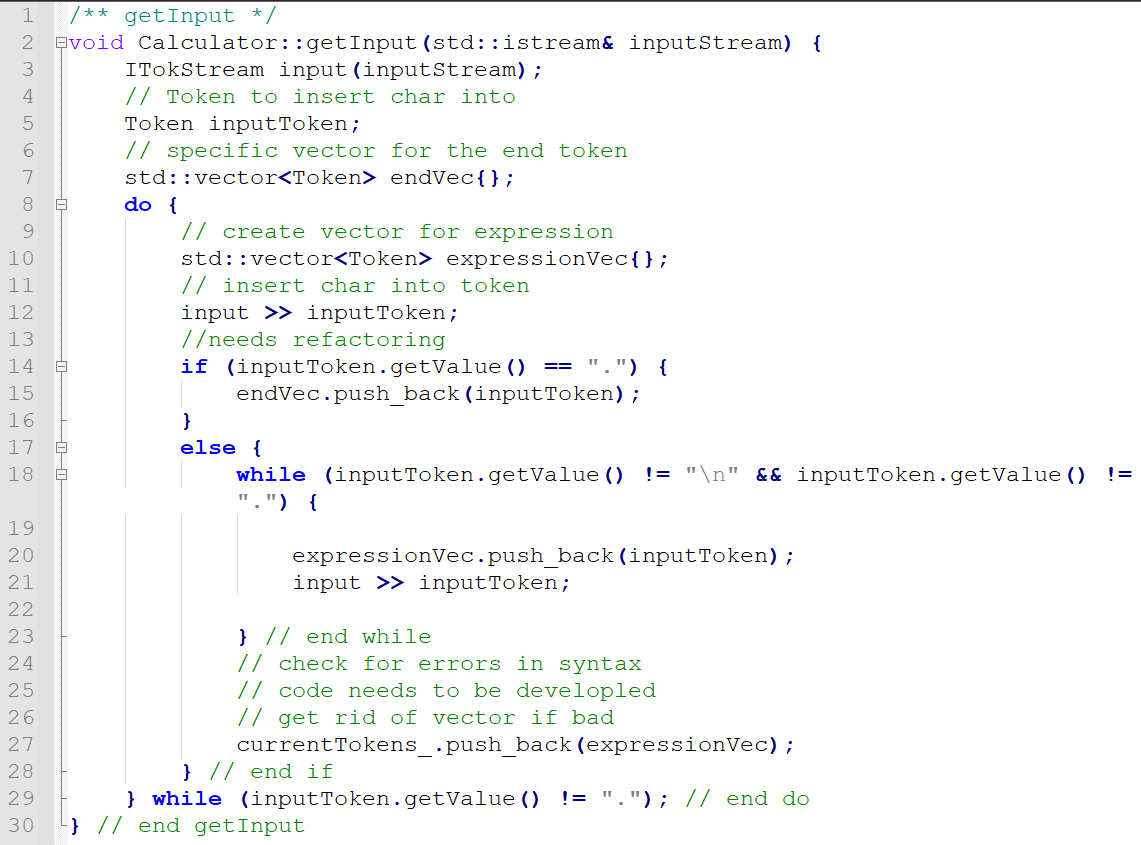
The Calculator class will be the frontend of the program and will be what the user interacts with. As such the user will only call methods from this class. This class will call the needed functions/methods from the other classes.

* **+Calculator().** The default constructor provided by the compiler will be suited for this class.

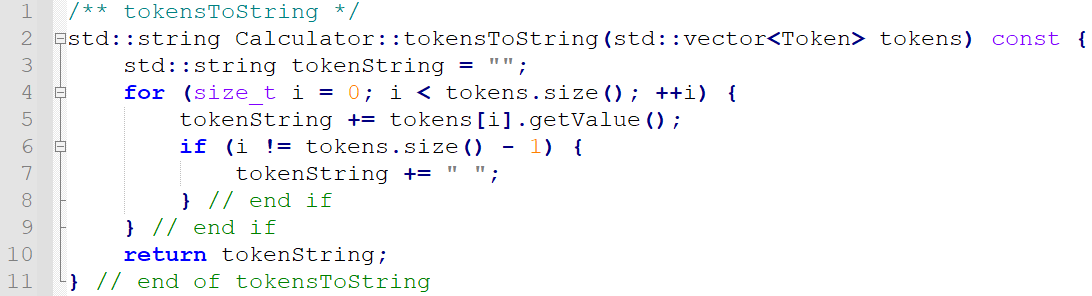
Graphical user interface, application

Description automatically generated

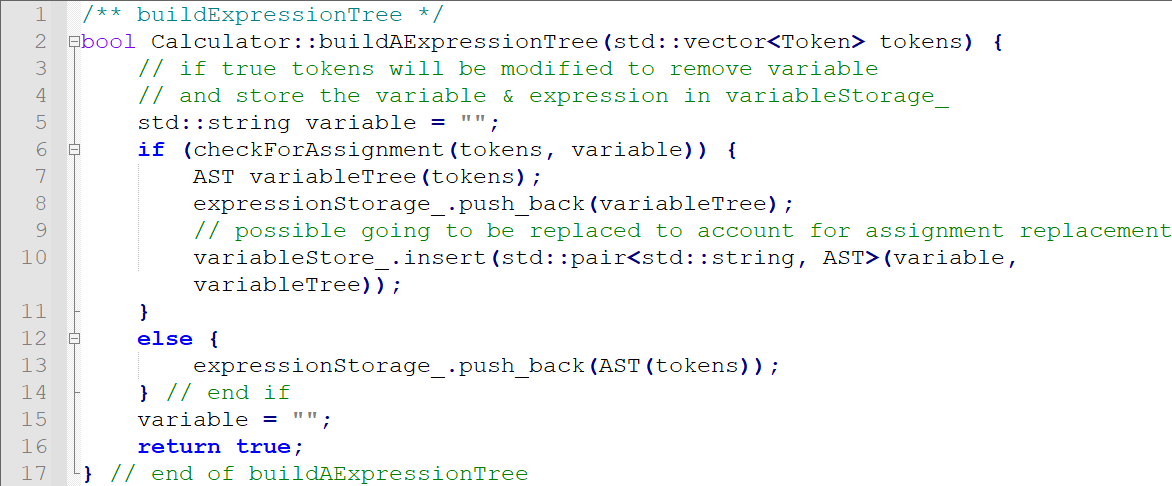
* **+getInput(inputStream: std::istream): void.** This method creates vectors for each expression and fills them by calling the ITokStream class. Each vector expression that is created is inserted into the vector storage currentTokens\_.



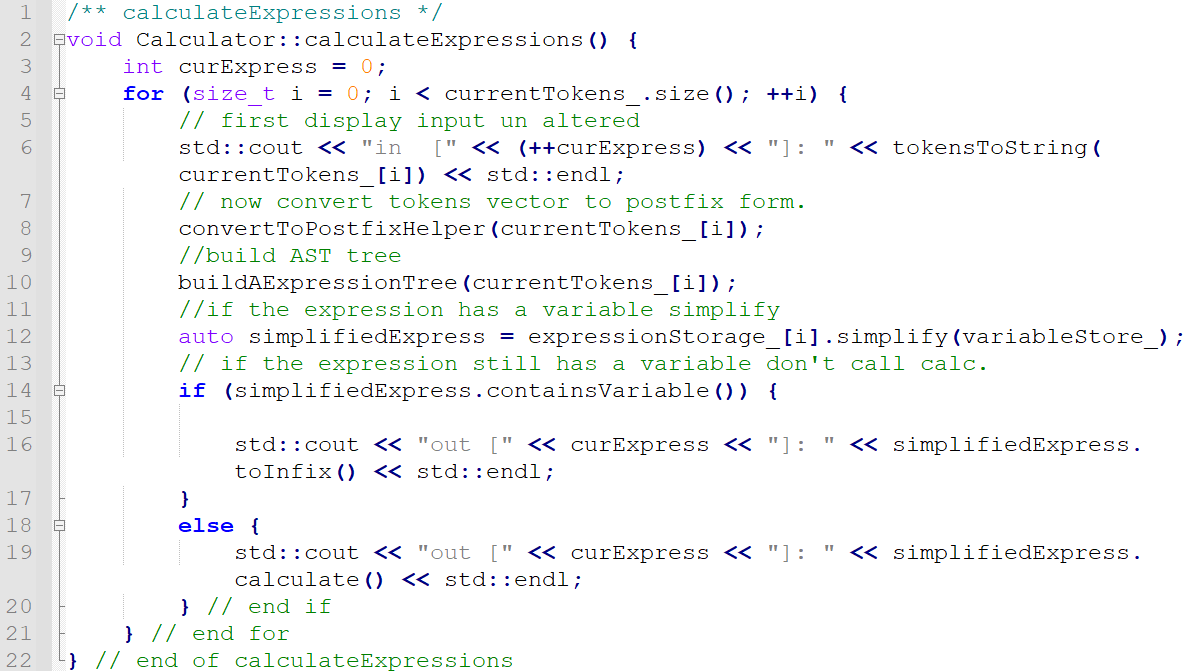
* **+tokenToString(token: std::vector<Token>): std::string.** Returns a string object that represents the provided Token vector.



* **+buildAExpressionTree(tokens: std::vector<Token):** **bool.** Returns true if tree was successfully built. This method also checks for assignment by calling the checkForAssignment. If there is an assignment the vector is updated before the tree is built and the variable and expression is stored in the variableStore\_. A created tree is added to the expressionStorage\_ variable for saving.



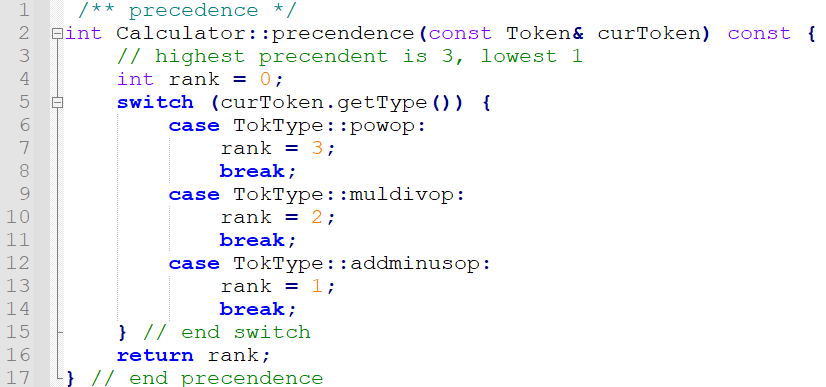
* **calculateExpressions(): void.** This method echoes the input back by calling tokensToString, then convertToPostfixHelper is called to convert the expression to postfix form before calling the buildAExpressionTree method. Once the AST is built the AST::simplify method attempts to simplify the expression before results are calculated and displayed.



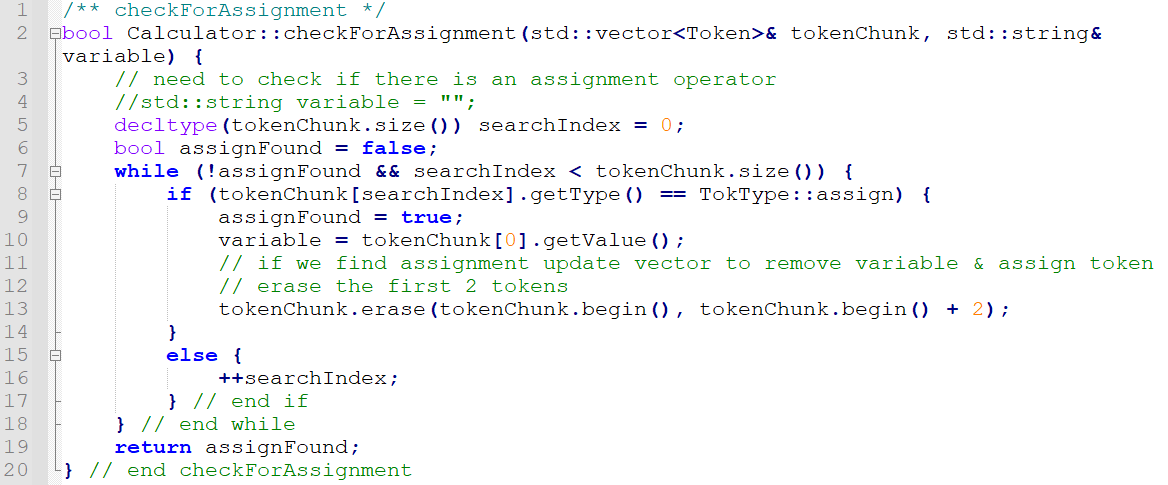
* **-convertToPostfixHelper(curTokenVec: std::vector<Token>): void.** Converts the provided vector from infix form to postfix form.



* **- precedence(curToken: Token):** int. Returns a integer value that represents the operators precedence level. 3 is the highest precedent and 1 is the lowest.



* **- checkForAssignment(tokenChunck: std::vector<Token>, variable: std::string):** bool. Returns true if a variable is found and updates the vector to remove the variable and assignment token, otherwise returns false.



* **- currentTokens\_: std::vector<std::vector<Token>>.** Token vector storage.
* **- expressionStorage\_: std::vector<AST>.** AST storage
* **- variableStore\_: std::map<std::string, AST>.** Variable storage for variables and their expression.

**Implementation Plan**

Much of the testing for this application will be handled by cout outputs strategically placed within methods and a main function. Additionally, the debugger and the watch variable ability of the IDE will play a key role in debugging and developing this program. Another possible testing approach that might be implemented would be simple display methods that will only be used during the testing phase of the system and discarded once the method/function has been validated. The complexity of this system will require some classes and methods to be made in tandem.

The plan is to implement the classes in this order:

Token->ITokStream->Calculator->AST.

Its important to not that the Calculator and AST classes will be codeveloped due to their coupling and function dependencies.

1. The Token class
   1. Constructor will be developed first and will be tested by examining the values of its attributes in the debugger or by using cout statement when a token object constructed
   2. The two getter methods of the Token class were implemented for safety purposes and will be tested by calling them on a Token with known values.
2. ITokeStream class
   1. The Constructor will be developed first and is simple enough that testing will not be implemented; however, if there are flaws in the constructor these should become apparent when input is processed.
   2. Next the determineTokenType will be developed in tandem with the overloaded operator>>. The testing will be down by printing out the char using cout statement that will be inserted into tokens for later use. Each cout statement should show the desired chars we want to store for use.
3. Calculator class
   1. The getInput method depends on the ITokStream’s overloaded operator>> method. This method will be tested by filling vectors with input and printing these vectors out. If the method is working properly the vectors should only contain the chars we desire.
   2. convertToPostfixHelper will be implemented and tested by converting each vector to the postfix form and printing each vector our using a for loop. Sample input would be (a+b)\*(c\*(d+e)) and the output should be a b + c d e + \* \*. It important to note that the precedence method will also be developed with this method as converToPostfixHelper is dependent on it.
   3. The last method of the Calculator class to be implemented at this time is the checkForAssignment. The testing will require examining a vector expression with a known variable/assignment and one with just a variable and no variables. The vectors will be printed out using cout and should reflect either the corrected modifications or no modifications to the vectors. Additionally, the variableStore\_ will be examined through the debugger to confirm the variable and its assigned expression have been inserted.
4. AST class, as further methods cannot be implemented in the Calculator class without.
   1. The constructors, build, and isOperator method will be implemented together as one of the constructors is dependent on the build method, and the build method is dependent on the isOperator method. The testing for these methods will utilize the debugger. As the AST tree is constructed and built, break points will be set watched to examine the structure of the tree is correct. This will be done by setting a watch to the key variable pointers that make up the tree.
   2. Next the destructor and its helper method clearTree will be developed. This will be tested by calling a memory leek test.
   3. Copy constructor, overload assignment operator, and its helper method copyTree will be developed. These will be tested by setting a Tree equal to another tree and confirm the memory location/ID in the IDE.
   4. Now that key methods have been constructed in the AST class we can begin to develop the toInfix, toPostfix, their helpers, and getHeightHelper methods. These will be tested by simply displaying the string they create to the console as the toInfix and toPostfix depend on the getHeightHelper. Any errors will be examined through the debugger.
   5. The calculate, calculateHelper, and doMath method classes will be tested only with simple expression such as 5+5 or (5 + 5) \* 10 / 15 and the results will be printed to the console at first till more the system has been built to deal with other complexities. Further testing of these methods will be revisited when variables are involved.
   6. containsVariable and containsVariableHelper will be tested on trees that are known to have variables and trees that don’t. The resulting bool should reflect true or false to their corresponding trees.
   7. Lastly, replace, simplify, and simplifyHelper will be developed and tested using the debugger follow specific changes in variables being watched at specific break points. These desired changes should be apparent variables change at each break point. Lastly, the simplified trees be printed out to the console to confirm the changes are correct.
5. Calculator class
   1. Now that the functionality is established in the AST Class the calculateExpressions will be developed and tested. This method mostly calls on other methods from the Calculator class and AST class. Its results depend on all other methods functioning properly. As such any undiscovered errors in previous methods should become apparent as these methods begin to interact to produce the desired result of the program. Since this method displays echoes the input and displays the results. The results will be the confirmation test that this method and the methods it calls are functioning properly.