A Symbolic Algebra Calculator

Specifications

**Problem Statement:**

For this assignment, I will be creating a symbolic calculator. This program reads and expression from cin through an overloaded istream it then converts the expression from infix to postfix using a stack to appropriately push and pop values and creates the binary tree used to simplify each expression. The simplified expression is then printed to the console using cout.

**Assumptions:**

* Output will be directly printed to cout and does not require an overloaded stream output operator.
* Will need to implement a move and copy constructor.

**Program Inputs:**

5 + 7

X := 1

X + 8

Z := x + y

Y := 8

z

Inputs will be read by cin which taken in a sequence of chars. This will be done through an overloaded istream operator which will make use of tokens and will break down each element in the sequence of chars. Examples of program inputs are shown to the right.

**Program Outputs:**

The output will be printed to cout in postfix notation. If a variable does not contain a numerical value then the actual variable is printed. Examples of the outputs corresponding to the inputs above are shown to the right.

12

1

9

1 + y

8

9

Design

For this program I will have 2 files. **Calculator.h** , **Calculator.cpp** , and **Main.cpp**. Calculator.h will hold all the definitions for the token classes and the AST class. Calculator.cpp will contain the implementation of everything defined in Calculator.cpp and Main.cpp will be used to methods, inputs, and outputs along the way. Although the various classes could be put into various different files, I find it easier to use just 2 files.

The **TokenType**, **iTokenStream**, and **AST** class will all be separate of each other.

**AST Class:**

**+AST(Vector<Token>& postfixExpr);**

* Constructor which takes the vector of the postfix expression and creates a binary tree corresponding to the given expression.

**+~AST();**

* Destructor which nulls out all the pointers and deletes all the nodes of the binary tree.

**+AST simplify(variableStore& v) const;**

* Parameter: a variable within the map and its corresponding expression
* Simplify algorithm takes the binary tree and evaluates the nodes within it to produce a simplified version.

Starting at bottom most leaf nodes

Take them and parent and apply operation using right and left values – do this for every parent working way up.

Switch for operators

Case ‘+’

Result = lVal + rVal;

Case ‘-‘

Result = lVal – rVal;

Case ‘\*’

Result = lVal \* rVal;

Case ‘/’

Result = lVal / rVal

Return root\_

**+string toInfix() const;**

* Parameters: none
* Returns: string which represents the infix expression.
* This method is marked as const as the value of the infix expression is unchangeable. It takes an expression in postfix and divides the rhs and lhs if there is a variable associated with the expression which will represent the lhs. If there is no variable associated with the expression, then this step can be skipped. The algorithm is then ran to convert the rhs to infix. The variable and associated infix expression is then stored into a map.

Stack s;

If there is a variable

Split after the := to separate the variable from the expression and store rhs in string

Loop from 0 to rhs.length

If is an operand

s.push(operand)

else if it is an operator

pop operends from stack

op1 = s.top()

s.pop()

op2 = s.top()

s.pop()

expression = op2 + rhs[i] + op1

s.push(expression)

return s.top

**-struct Node**

* **Node (token t);** //creates a node given a token which is passed in
* **Token tok\_;** //represents the token
* **Node\* left\_;** //pointer to the node on the left
* **Node\* right\_;** //pointer to the node on the right

**-Node\* root\_;** ///pointer to the root node

**ITokenStream Class:**

**+ITokenStream(Istream& inputStream);**

* Parameters: istream&
* Takes an istream and assigns it to is\_

**+ITokenStream& operator<<(Token& rhs);**

* Parameters: Token&
* Returns: iTokenStream& which is an overload to cin
* This method takes the rhs and reads each char, figures out its type and corresponding value using the TokenType class. It will also display error messages prompting the user of bad input.

While there are more chars to read – read the next one

If == + = \* /

Return {operator, value}

if is a digit

return {number, value}

If is a paren

Return {lparen or rparen, value)

**+ explicit operator bool() const;**

* The tokenizer can be cast to a bool which will return if the token is valid by returning true and false if It is not.

**-Istream& is\_;** //represents the istream operator

**Enum class TokenType:**

**{operator, variable, number, lparen, rparen, end}**

* Operator represents \* / + - ^
* Variable represents for examples x, y, z, a, b, c which are the single chars in which the expression is stored within (left-hand side).
* Number represent the right-hand side which could be any positive int.
* lparen represents ( and rparen represents )
* End represents a single dot character ( . ) which indicates that the user is done entering input and wants to exit the program.

**Struct Token**

* **Token(TokType t, String v);** //token that identifies token type and associated value.
* **TokType\_;** //the type of token which could be an operator, number, variable, paren, etc.
* **String value\_;** the string value associated to each token.

Implementation Plan

1. I will begin by implementing the ItokenStream and enum TokenType. I will assign the correct values to each type in the enum to make sure that the istream operator is assigning the right type and value for each character read. I will also make sure that error messages are being printed for bad input and will also assure that the program exits once a . is entered.
2. I will then move to implementing the postfix to infix method and use some cout statements to ensure the right expression is being computed.
3. I will then work on the constructor and destructor. The constructor will take the converted postfix expression and will create a binary tree out of it. The destructor will then null out all the pointers of in the tree and will delete all the nodes.
4. I will then work on the simplify method. For this method I will need to ensure that the right simplified expression is being printed. I will also need to make sure that each operator is performing the correct operation by stepping through the program. I will also need to ensure that the operations start from the bottom of the tree and work their way up to the root.
5. I will then implement the copy and move constructor which should be similar to the SkipList program.
6. I will then test on the Linux machines and run on Valgrind to make sure I have no memory leaks.