Group 3-2: Justin Bisignano, Tylor Childers, Hanif Ahmed, Ian Van Stralen, Kyle Stratis

Symbolic Calculator Project Design

Design Theory

Our group will utilize polymorphism to abstract elements of the calculator's implementation and allow individual group members to work on different aspects of the calculator. The goal is to create an infix notation calculator that runs from the command line. Some form of UI may be explored, time permitting. Our classes will be implemented in a way such that the possible addition of the UI will require no modification to the driver or calculator class. This design will allow for simple implementation of additional features in the future. The driver class will create a new calculator from user input and the calculator class will parse the input, delegating operation to the appropriate number class.

Program Flow and Inner Workings

Our main class will be the first called and will prompt the user for input from the command line. This input will be stored as a full string, to be passed to a new instance of our driver class. This driver class stores all previous answers for retrieval later, much like many modern graphing calculators. The input is passed to a calculator for parsing and, you guessed it, calculating. The parsing will convert the input string into an ordered set of defined number types, values, and operations according to PEMDAS and grouping by type (logs, radicals, and fractions). This ordering can be done in place leaving the input as infix notation, or by using an algorithm to make working with the eventual chain of numbers and operators easier, such as the shunting-yard algorithm.

Parenthesis are handled by selecting the text between the parenthesis and passing it to a new calculator class. That class then returns a simplified, calculated version of the parenthesized expression which is replaced into the original input string in the calculator class, thereby allowing further simplification and computation of the parenthesized expression.

Each number type (radicals, rationals, integers, etc.) will be polymorphically derived from a base Number class. The four primary arithmetic methods (add, subtract, multiply, divide) will be overloaded to account for the different possible types of numbers and their behaviors. Furthermore, the Radical, Rational, Exponent, and Logarithm classes will have a simplify function that will be called prior to output. Simplifying Rational numbers can be done using a recursive implementation of Euclid's algorithm. Logarithms have special properties will need to be handled. For example, the logarithm of a product can be decomposed into the sum of the logarithm of the factors. Thus, we consider the most simplified form of a logarithmic expression to be the sum of the logarithm of prime numbers. Also, we could exploit the change of base property for fractions where both the numerator and denominator are logarithms of the same base. For Radical simplification, it would be useful to list the prime factorization of the radicand (this can either be done in the Radical class or in the Number class depending on whether other classes also rely on prime factorization).

From the ordered input of numbers and their types, the calculator will take each number and the next operation to be performed, calling that action on the following number. For example, in 2 + log(3), the number object representing the 2 will have the add method called with the object representing log(3) as an argument. That should then return a new number, which in this case would be ‘2 + log(3)’ as it can’t be simplified further. When calling math methods on number types than can be simplified, those methods will first do the math and then call their respective simplify() method, returning that result. In the case of a division such as 2 / sqrt:2, we will create a new fraction (Rational), and then simplify it. This would entail multiplying by the fraction sqrt:2 / sqrt:2, giving us 2*sqrt:2 / 2 as the ultimately returned result.

Certain constants, such as pi and e, will remain as their string representations, essentially being treated as variables would be in algebra. pi + pi will simply return 2*pi, and e * e / e will return e. These constants will have to be treated as special cases in our math methods to handle this behavior.

Each subclass of Number will have specific errors to throw. For example, in the Rational class, an error should be thrown if the denominator equals 0. In the Logarithm class, we throw an error if the argument of the logarithm is less than or equal to 0. In the Radical class, we throw an error for radicands less than 0, unless we choose to add a Complex class at a later time. The errors are then caught by the Calculator class and handled appropriately.

Once all of these calculations have been completed in order, the calculator should contain a fully simplified number, as each sub-calculation has itself been simplified. This final output is stored as a string and is passed back to the driver class for storage. The driver class can then refer to it as the previous answer, inserting it directly into the next calculator without having to recalculate the previous input as well. The driver class then passes this result back up to the calling main class, which will print the output for the user.