COP 3503 – Project, Stage 1

Purpose

To determine the design/architecture for a potential software project.

Description

The vast majority of computer-based mathematical systems operate through the use of the floating-point representation of numbers. This representation is quite analogous to scientific notation, which seeks to preserve the magnitude of a number and its "most significant digits." As such, we may note that in many cases, floating-point numbers are not exact representations – and their representation often does not give us insight into how any particular number came to be computed or constructed. There is nothing inherent in 1.4142 that indicates its relationship to the number 2 – in particular, that $1.4142 \approx \sqrt{2}$.

This project, as a whole, is to build a computer-based symbolic math system that avoids floating-point representations as much as possible. Rather than say $\frac{4}{6} = .666\ldots$, it would say that $\frac{4}{6} = \frac{2}{3}$ which represents the result, directly, as a fraction. Likewise, the system would prefer the representation $\sqrt{2}$ over 1.4142, and would be capable of performing calculations with this representation.

Instructions

Given the specifications listed below, generate a design/architecture (even an implementation plan?) for your eventual program. Be sure to indicate relationships among classes (polymorphism + inheritance included) and the role that each designed class will play in the overall program.

Be sure to include visual, UML-ish diagrams in your design documentation. It may be prohibitive to have every class represented simultaneously in a diagram, depending upon your solution for this stage; feel free to break the overall diagram down into sectors within which the groupings of classes are highly related. (Take a look at the UML diagrams in March 19th's lecture PPT, slides 18+19 for a rough example of what I'm suggesting.)

Requirements + Specifications (your contract)

- Your program must be able to support the following number types:
  - Integers
  - Rationals
  - pi, e, nth-root irrationals.

- Unless specifically requested by the user, your program should ***never*** show floating-point/decimal numbers.

- Your program must support the following operations on EACH supported number type:
    - Addition / Subtraction
    - Multiplication / Division

    - Note that if two numbers cannot actually be added (say, 2 + pi), no error should result; the expression should be left as is.

- Your program must support the following operations, where reasonably feasible:
    - Exponentiation (by rationals only – do NOT worry about "x"^pi or "x"^e.)
    - Taking the nth root of a number.
        - Do not worry about supporting even roots of negative numbers. Things will already be "complex" enough as they are. Though, if you want to support it, feel free.
    - Logarithms of base 2 or greater.
        - This includes at least base e and base pi.

- Your program should support parentheses to any depth, so long as they match by the end.

- Operator specifications:
    - "x to the power of y" == x^y
    - "the square root of x" == sqrt:x
    - "the nth root of x" == nrt:x, where "n" is replaced by a number.
        - This could always just be done through x ^ (1/y), of course, so the "n"rt:x operator does not necessarily have to be used.
            - Examples: 3rt:8 = 2, 4rt:16 = 2, etc.
    - "log base b of x" == log_b:x

Lowest-terms specifications:

- Your program should keep rational numbers in "lowest terms" whenever possible.
  - (The "greatest common factor" between the numerator and the denominator should be 1.)
  - No square/nth roots should remain in the denominator – if necessary, multiply both numerator and denominator as appropriate to meet this condition.

- For square/nth roots, the final form should resemble $2\sqrt{2}$ instead of $\sqrt{8}$.

- For logarithms, each separable term should be listed separately: $log_7(6)$ should be written as $log_7(2) + log_7(3)$.
  - Your implementation should utilize the change-of-base property of logarithms in the case that a fraction has same-base logarithms in both the numerator and denominator.

- Do not worry about reducing any fractional forms that require algebraic-style factoring (anything that requires inversion of the distributive property). For example, $\frac{e-1}{e^2-1}$ *is sufficiently lowest terms* for this assignment, *as is* $\sqrt{\pi^2 + 2\pi + 1}$. This is completely optional, as it could easily take a lot of undue time and effort to get it right. (Getting it right would almost surely result in some extra credit, though.)

Input specifications:

- Your program will NOT be hardcoded with the operations we want to perform. Instead, your program ***must*** take input from the user at run-time by console.
  - Feel free to add in file support, but it's not necessary: there exist built-in Windows/Mac/Linux methods for passing a file's contents to a program as if they were console input. (It's called "piping.")

- Your program must allow the user to input a full, arbitrary mathematical expression in a single step – i.e., before any attempts at processing it begin.
  - You may expect the input to have a space between every pair of sequential operators and every operator-number pair. Whether or not this expectation is utilized by your program is up to you.
  - Standard order-of-operations *must* apply to the input expression. (PEMDAS must be followed.)
  - Your program must be able to handle expressions such as "3 * pi", but the ability to handle "3 pi" (without explicit multiplication) is optional.

    - Expressions with a leading negative number must be acceptable by your program, though “2--3” (as in “2 - (-3)”) is optional.

- Your program should accept the “ans” keyword, which will indicate to the program to “use the answer from the last problem.” Said answer, when inserted into the new expression, should function as if it were within parentheses. (This is to avoid the order of operations from taking effect, as the last answer *could* be a lowest-terms expression.)
- Optional input: floating-point numbers may be read in, then converted to fractional forms by the program.

- Your program should have the following menu options:
    - Compute new expression.
        - This mode should stay active unless the “back” or “quit” commands are used.
    - Help.
        - Ideally, help will have suboptions of its own regarding the format for different operations and program modes.
    - Review previous expressions and answers.
        - Suboption: Show floating-point form for the answer for the previous expression “n”.
        - Suboption: Set “ans” to previous expression “n”s answer.

          The most recent expression should be marked “1”, the second-most recent marked “2”, etc.
    - Quit.

- Any expressions which generated an error should *not* be added to the list of “previous expressions” which may be accessed by the menu.

Errors

- Errors should *not* crash your program to the desktop, and should report meaningful information to the user.
    - Example error – “sqrt:-1”…
        - “Error: Cannot take the square root of a negative number!”
          “Source: sqrt:-1”
    - Being able to tell the user which part of the expression caused the error is not required, though it’d be nice.
        - If you do attempt this, you only need to report the actual operation that caused it – that “-1” above could result from an original expression of “sqrt:(1-2)”. “sqrt:-1” is still all that I’d expect to

see reported.

Thoughts for Later Stages

In case you wish to plan ahead, the next (and final) stage which will be requested of you: the complete implementation of your program, fully tested and everything. You may wish to start coding early, as I anticipate the implementation efforts to get quite involved – in fact, the coding process may help you to think through elements of your design.

- One more “thought” for later, to make sure that there's plenty of effort to go around: your prime factorization method (I'm assuming you'll have one because of the need for putting fractions in lowest terms) **must be recursive**.
    - This is really more of an "implementation" detail that you’ll only need to worry about later. I simply want to warn you about this early, so that you can get to thinking about what this would involve.
    - If you don’t have a prime factorization method, that’s okay so long as your program does meet the full specifications. Though, I’m interested to see how that might work out for the project as a whole.
- An alternative to prime factorization is Euclid’s method; if you choose to use this instead, this **must be recursive**.
    - Again, implementation detail for later.

Submission

Please put your submission in one of the following forms:

- *.jpg
- *.gif
- *.png
- *.pdf
- *.doc / *.docx

If you end up with a lot of files, and thus wish to submit them grouped together in a compressed file, please use one of the following forms:

- *.zip
- *.tar.gz

Diagrams can easily be quite tedious to create from scratch on a computer, either with office-style software or with photo-editing tools. Feel free to create all of your diagrams by hand; just scan them in to *.pdf files if possible, or into one of the other three mentioned standard image formats.