COP 3503 – Project Overview

Purpose

To develop a functioning piece of software while exercising techniques from throughout the course.

Description

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.

In contrast, 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 $1.4142 \approx \sqrt{2}$.

Requirements + Specifications (your contract)

In the remainder of this document, you'll note numerous specification components in red font. Any red-typed component is optional and will be considered for extra credit.

- Your program must be able to support the following number types:
  - Integers
  - Rationals
  - nth-root irrationals (such as $2^{\frac{1}{3}} = \sqrt[3]{2}$).
  - pi, e (as their plain-text symbols)

- 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 nth root of x" == n rt x, where "n" is replaced by a number.
        - Examples: 3 rt 8 = 2, 4 rt 16 = 2, etc.
        - More complex: (6 / 2) rt (5 + 3) = 2 [Evaluates first to 3 rt 8.]
    - "log base b of x" == x log b
        - Example: 8 log 2 = 3, (13 + 14) log (2 rt 9) = 3.

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.)
        - Consider "log" and "rt" to be on the exponential tier.
        - Any evenly-matched operators evaluate from left to right.
    - 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 use the order of operations to preserve the prior answer, as the last answer *could* be a lowest-terms expression that would otherwise be mangled if dumped in as plain-text.)

- 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 must *not* crash your program to the desktop, and should report meaningful information to the user.
    - Example error – "2 rt -1"…
        - "Error: Cannot take the square root of a negative number!"
          "Source: 2 rt -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 "2 rt (1-2)". "2 rt -1" is still all that I'd expect to see reported.

Thoughts for Later Stages

In case you wish to plan ahead, here's a rough roadmap of my present plan for the project stages:

Stage 1: implementation of the Shunting-Yard algorithm.

Your program must take in a possible expression for the eventual program and parse it into post-fix form.

Stage 2: design of the program's overall structure.

You will be required to present a roadmap / plan of your eventual structure for the full program – the set of objects you wish to code to meet the stated goals.

I will provide a suggested starting point for your design. Although it won't be necessary to utilize, use of said "starting point" will give more practice with advanced topics from this course and can actually make your implementation easier come Stage 3.

Stage 3: Implementation of the overall program.

You may wish to start designing and 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.