Note: This is an extra credit assignment. Up to 100 points of extra credit can be earned with this assignment.

Instructions

- This assignment is to be done in Haskell
- Please submit your answers via CSNet.
- For this assignment submit exactly one file, 7.hs (use exactly this name), and nothing else. This file should contain the code requested in the exercise.
- Remember to put a comment at the top of your file, including your name and acknowledgements of any help received.
- You must provide example code showing how your routines are used. This is a more open-ended assignment, and you should test a range of cases for each part. You will need to show your tests as part of your turned-in code.
- In comments at the top of your file, clearly indicate which parts of the assignment you believe you have implemented.

Representing Algebraic Numbers

A source of several robustness problems in numerical and geometric computing is the failure of what is known as the “Real RAM” model of computation. The Real RAM model states that computations on numbers are done in constant time over the Reals, and we often write algorithms assuming the Real RAM model of computation. However, it is impossible to achieve the Real RAM model in the real world! Most commonly the “Real” part of the model is given up. Instead of computing with real numbers, we use floating-point numbers or integers. These numbers can be handled in constant time by the processor, but are only approximate, and it is well-known that we have to round off many values to a nearby floating-point number. This roundoff error can cause significant problems in some circumstances.

An alternative is to say that we will represent real numbers exactly, but in doing so, we can no longer perform computations on those numbers in constant time. Some numbers might have complex representations, and the time needed for computation is related to the complexity of that representation.

In this assignment, we will show how we can work with an important subset of the real numbers, the algebraic numbers. Algebraic numbers are the numbers that can be defined as roots of polynomials with integer coefficients. For example, all integers and all rational numbers (i.e. fractions) are algebraic, as are numbers like the square root of 2.
One way of representing an algebraic number is as a polynomial and an upper and lower bound between which there is exactly one root of the polynomial. For example, \( \sqrt{2} \) can be represented as the polynomial \( x^2 - 2 \) and the range \([0,2]\).

Notice that \( -\sqrt{2} \) would be represented by the same polynomial, but a different range, such as \([-2,-1]\). Also, notice that we can shrink the interval surrounding the root as much as we’d like. Instead of \([0,2]\), we could have used \([1,2]\) or \([1.4, 1.5]\) or \([1.41, 1.42]\) or \([1.414, 1.415]\), etc. If the root is a single root, evaluating the polynomial at one bound will be positive, and at the other bound it will be negative. We will assume that all of our examples are single roots. If this is the case, we can evaluate any point between the high and low bound on the root – if the polynomial is positive at that point, we can update the positive bound, and if it’s negative, we can update the negative bound. We can make a bound arbitrarily small in this way.

If a value is known exactly, we can represent it with an interval of length 0. For example, the number 3 could be represented by the polynomial \( x - 3 \) and the interval \([3,3]\). Notice that there can be many polynomials that represent the same number. For example, 3 could have been represented by the polynomial \( x^2 - x - 6 = (x-3)(x+2) \) instead.

Since algebraic numbers can be defined to any level of accuracy desired, we need to use arbitrary precision values for the interval bounds. For this assignment, you should use arbitrary precision rational numbers. You can use the Rational class (see: https://www.haskell.org/hoogle/?hoogle=Rational) to represent these numbers.

The goal of this extra credit assignment is to write a system for representing and operating on algebraic numbers. You can receive partial credit for completing parts of this assignment. There are several stages to this assignment, and completing different parts can generate points. While you should generally work on these parts in order, you can skip some parts to address others. Points possible per part are given in brackets.

1. Defining a basic bounding interval type [16]
   a. You should, at minimum, have an interval type with a lower and upper bound. You should be able to easily generate the midpoint of the interval, output the interval, and retrieve the high and low bounds.
   b. Additionally, you should be able to compare two interval using the > and < operators. One interval is above another if its lower bound is above the upper bound of the other. Note that intervals represent some point inside the interval, and thus if two intervals overlap, they are neither > nor < each other.
   c. As another stage, you should allow interval arithmetic. Intervals can be added, subtracted, multiplied, or divided. You can find the rules for basic interval arithmetic online (e.g. https://en.wikipedia.org/wiki/Interval_arithmetic).

2. Defining a Polynomial type [6]
   a. A polynomial type should have a degree and (arbitrary length) integer coefficients
   b. You should be able to evaluate the polynomial at a given rational value
3. Defining a Root type [6]
   a. A root should have, at minimum, a polynomial and a bound
   b. You should define a shrink function that shrinks the interval. This can be done by evaluating the midpoint of the interval in the polynomial, and then shrinking the interval by adjusting one of the two interval endpoints (or, if you happen to have found a root, adjusting both).

4. Comparing two Roots [10]
   a. Two roots should be able to be compared by > and <. You can assume that no two roots are equal. If the bounds of two roots overlap, you will need to shrink them until they no longer overlap.

5. Arithmetic on Roots [20]
   a. You should allow roots to be combined with addition, subtraction, multiplication, and division. Note that you might wish to use a tree representation to represent these arithmetic combinations. You can assume that no numbers are equal to zero, and that no numbers formed this way are equal to each other.
   b. The resulting numbers can be compared, just as roots were. To find the bound, you will perform interval arithmetic on the bounds of the individual numbers.
   c. If these algebraic numbers need to be refined, you can shrink the interval of every root in the algebraic expression, and recalculate a new bound.

6. Parsing with variables [16]
   a. You should parse an input in which the values can be stored and retrieved by name.
   b. Values can be stored by name if written <name> = ...  
      i. For example, “b = 3” would define a variable b with the value 3.
   c. Values can be retrieved by name in an expression
      i. For example, “2+b” would give the value 5, since b has the value 3.

7. Parsing algebraic numbers [14]
   a. You should define a format for describing roots as a bound and as a polynomial.
      i. For the bound, you should be able to read in the upper and lower bound.
      ii. For the polynomial, you should read in the degree, and then each of the coefficients.
      iii. You may define keywords that will specify that you are defining a bound/polynomial/root if you wish
   b. You should then combine this with the variable parsing to allow roots to be stored by a name.
   c. You should then allow these roots to be combined by algebraic expressions, so that arbitrarily complex algebraic numbers can be formed.

8. Parsing queries [12]
   a. You should allow the > and the < operators to be parsed and then compare the numbers involved
      i. It is intended that the numbers involved will be algebraic numbers.
ii. Output either true or false for each comparison. Since you can assume that no two algebraics are equal, and no algebraic number is 0, you output either true or false for every comparison.

For example, the following might be an input (the details will differ depending on your implementation):
A = Root (1, 2) Poly 2 [1, 0, -2]
B = Root (1, 2) Poly 2 [1, 0, -3]
C = Root (2, 8) Poly 1 [1, -3]
D = A+B
C > D
D > B

And would output
False
True