COMP 311 HOMEWORK 1: EXTENDING THE NUMERIC TOWER

Due Thursday, September 10 at 2:30pm
NO LATE SUBMISSIONS

Although the numeric types `Int` and `Double` are useful for many computational tasks, there are others for which they are ill-suited. In this homework assignment, we will make use of compound and abstract datatypes in Scala to define several new datatypes inspired by algebra and scientific notation.

Please submit your homework via the turnin system, in a folder named `hw_1` with three files:

`Rationals.scala`, `Intervals.scala`, `Physics.scala`

containing your solutions to the three sections of this homework, along with their three test files:

`RationalsTest.scala`, `IntervalsTest.scala`, `PhysicsTest.scala`.

For each section, please turn in only your final program resulting from completion of the section.

1. The Rationals

In mathematics, the Rational Numbers (denoted by the symbol $\mathbb{Q}$) are defined as equivalence classes of pairs of integers. We can use pairs of the fixed sized type `Int` to model a fixed size type `Rational`. Of course, there are many pairs of integers that denote mathematically equivalent rational values, such as $1/2$ and $2/4$ and even $-1/2$ and $1/-2$.

(1) Define a case class `Rational`. Your class should include methods that support the following operations:

- The standard algebraic operations addition, multiplication, subtraction, and division among pairs of Rationals, denoting these operators with the standard symbols `+,-,*,/`
- Raising a Rational to an integer exponent, using the operator `**` to denote exponentiation. (We follow Fortran tradition and choose the symbol `**` for exponentiation rather than the ASCII symbol `^` because Scala defines the precedence of `^` as lower than that of `**`, which can be misleading in mathematical formulas. The precedence of `^` was chosen because of its use as the bitwise exclusive or operator on `Int` values, a notation borrowed from Java, which borrowed from C.)

Be sure to follow the Design Recipe discussed in class, along with standard function templates for algebraic functions, conditional functions, and functions over compound data. Also be sure to include adequate tests to convince yourself of the correctness of your program, especially along boundary cases.

(2) The greatest common divisor (GCD) of two integers (at least one of which is not zero) is defined as the largest integer that divides both with a remainder of zero. The most obvious way to compute a GCD is to find the prime factorizations of each number and find their largest common subsequence of primes. But this approach is expensive and even intractable for large numbers.

Fortunately, Euclid discovered a more efficient algorithm in the *Elements* (c. 300 B.C.). The Euclidean Algorithm is one of the oldest numeric algorithms still in common use. Euclid's approach relies on the observation that the GCD of two numbers does not change if the larger number is replaced with the difference of the larger and smaller numbers. The algorithm is defined by the following equations:

$$\begin{aligned} gcd(a, 0) &= a \\ gcd(a, b) &= gcd(b, a \bmod b) \end{aligned}$$

The mathematical operator `mod` returns the modulus of two `Int` values. For positive integers, this operation is equivalent to the remainder of integer division, which is denoted over `Int` values in Scala by the operator `%`.

(a) Define a singleton object `Arithmetic` to hold your definitions of algebraic functions.

(b) In this new object, define a function `gcd`, which takes two positive values of type `Int` and returns their GCD.

(c) Define a new method `reduce` on class `Rational` that takes no arguments and returns a mathematically equivalent value of type `Rational` in which:

  (i) The number is reduced to lowest terms

  (ii) The denominator is positive

  For every pair m, n of mathematically equivalent `Rational` values, the result of calling `reduce` on either of these values should be *structurally identical*, which can be checked on instances of case classes with the Scala operator `==` or the method `equals`.

(d) Define a new binary equivalence method `EQ` which returns true iff its arguments are mathematically equivalent.

(e) Similarly, define binary methods `LT`, `GT`, `LE`, `GE`, which compute the Rational operations $<, >, \leq, \geq$ respectively.

(f) Modify your definitions of the algebraic operators on Rationals (`+,-,*,/,**`) so that the result of each operation is reduced to lowest terms.

2. Interval Arithmetic

In scientific measurement, we include upper and lower bounds on the precision of a measurement. For example, consider the mass of the Higgs Boson as measured by the Large Hadron Collider at CERN:

$$126 \pm 1 \text{ GeV/c}^2$$

There is a rich collection of work defining computation with *closed intervals* of the Reals, delimited by pairs of floating point numbers denoting the lower and upper bounds of a measured value. In the example above, the Higgs mass would be represented by the interval:

$$[125, 127] \text{ GeV/c}^2$$

Arithmetic operations on intervals over the Real numbers $\mathbb{R}$ are defined as follows:

$$\begin{aligned} [a, b] + [c, d] &= [a + c, b + d] \\ [a, b] - [c, d] &= [a - d, b - c] \\ [a, b] \cdot [c, d] &= [\min(ac.ad, bc, bd), \max(ac, ad, bc, bd)] \\ \frac{[a, b]}{[c, d]} &= [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)] \text{ where } 0 \ \notin [c, d] \end{aligned}$$

Division by an interval that contains zero is undefined.

A computation on two intervals *must* result in an interval that contains the exact real value of the underlying mathematical operation. Endpoints of a new interval must be rounded conservatively so as to ensure that the exact value is contained despite approximations introduced due to rounding. To simplify rounding considerations, we will work with intervals that have `Rationals` as endpoints rather than `Doubles`.

Because an interval denotes an unknown real value bounded by its endpoints, mathematical comparison becomes more complicated. We must distinguish between cases where we are *certain* that the underlying real value is ordered with the respect to another, and cases where it is *possible* that the two underlying values have a particular ordering.

(1) Define a case class `Interval` with fields `left` and `right` of type `Rational`. The value of field `left` should always be less than or equal to the value of field `right`.

(2) Include definitions for the algebraic operators (`+,-,*,/,**`). As before, you need only support integer exponents.

(3) Define the following comparison methods on intervals:

  (a) `PEQ` indicating that two intervals denote *possibly equal* values.

  (b) `LT` indicating that one interval denotes a value that is necessarily less than another.

  (c) `GT` indicating that one interval denotes a value that is necessarily greater than another.

3. Computing With Bounded Precision

We have now built the machinery to perform scientific calculations with strictly bounded precision. Let's use our machinery to calculate something!

Ever since the (dwarf) planet Pluto was discovered in 1930, its size has been hotly debated. This debate was put to rest on July 14, 2015 by the New Horizons spacecraft, which measured Pluto's diameter to be:

$$2,372 \pm 2 \text{ km}$$

In contrast, the mass of Pluto (measured by its gravitational effect on other heavenly bodies) has long been known with precision. In 2004, The Institute of Applied Astronomy of the Russian Academy of Sciences published the *EPM 2004* ephemeris, a very precise measurement of the mass of all planets and large bodies in the Solar System, compiling data collected over 90 years. The measurement of the mass of Pluto in this publication was:

$$1.304 \pm 0.005 \times 10^{22} \text{ kg}$$

(1) Define a singleton object `Physics` that contains a function `volume`, which takes as input an `Interval` denoting the diameter of a ball (be sure to document the physical units expected of your parameters!). Your function should return an `Interval` denoting the volume of the ball.

(2) Now define a function `density` that takes a weight and a volume (as intervals, with expectations of units documented) and returns an interval denoting the mean density of the ball.

(3) Using your newly defined functions and the measurements provided above, define a constant `densityOfPluto` in object `Physics` to be an interval containing the true density of Pluto. (It is permissible to approximate the shape of Pluto to be a perfect sphere.)