Problem Set 3

Always code as if the guy who ends up maintaining your code
will be a violent psychopath who knows where you live.
— *John Woods*

- This problem set is due **at 9:00am** on **February 17, 2025** .
- This problem set comprises 3 problems.
- Submit early. Do **not** wait till the last moment.
- Each solution should start on a new page.
- We will give full credit **only** for correct solutions that are described clearly and convincingly.

Problem 3-1. My Type? Rational, Well-Defined, and Denominator-Stable [10 points]

Consider the custom type `nat` defined as:

```
type nat = Zero | Succ of nat
```

Use the above to define type `rational` that represents rational numbers. Remember that the set of rational numbers is defined as:

$$\mathbb{Q} = \left\{ \frac{p}{q} : p, q \in \mathbb{Z}, q \neq 0, \gcd(p, q) = 1 \right\} \tag{1}$$

That is, the type `rational` can be defined as pairs of integers p (numerator) and q (denominator), where q is non-zero and p and q do not have any non-trivial common divisors. Remember that rational numbers may be negative.

Problem 3-2. Rational Numbers [60 points]

Use the `rational` type defined in the previous problem for designing the following functions:

(a) [15 points] Define the binary operation `add_rational` with the signature:

```
add_rational: rational -> rational -> rational
```

That is, `add_rational` takes two numbers `p:rational` and `q:rational` as an input and adds them and returns them in the *reduced* form, that is, where the numerator and denominator have no common divisors.

(b) [15 points] Define the binary operation `sub_rational` with the signature:

```
sub_rational: rational -> rational -> rational
```

That is, alike the previous subproblem, `sub_rational` takes two numbers `p:rational` and `q:rational` as an input and returns their difference (in the reduced form).

(c) [15 points] Define the binary operation `mult_rational` with the signature:

```
mult_rational: rational -> rational -> rational
```

That is, `mult_rational` takes two numbers `p:rational` and `q:rational` as an input and returns their product in the reduced form.

(d) [15 points] Define the binary operation `div_rational` with the signature:

```
div_rational: rational -> rational -> rational
```

That is, `div_rational` takes two numbers `p:rational` and `q:rational` as an input and divides them and returns them in the *reduced* form, that is, where the numerator and denominator have no common divisors. We also assume that the denominator is a non-zero value. If you need to make any assumptions for this part of the code, please state them clearly.

Problem 3-3. The Square Root of All Evil [30 points]

Given a positive rational number r, its square root $\sqrt{r}$ can be approximated by a simple recursive process. Begin with an initial estimate a_0 (say $a_0 = 1$). If this estimate is *good enough*, return it. Otherwise, refine the approximation using:

$$a_{i+1} = \frac{1}{2}\left(a_i + \frac{r}{a_i}\right) \tag{2}$$

Each successive a_i is an improved approximation. Moreover, one can prove that a_i *converges* to $\sqrt{r}$ in limit. In this problem we will approximate the value of square root within an ε bound, that is, we want to compute a value a_N such that $|r - a_N^2| \leq \varepsilon$. For instance, if the given value of r is 2 and ε is 0.01, we will first pick a value for a_0 say 1, and then get the following sequence of approximations using the recursion in Equation 2. The sequence we get is:

$$1, \frac{3}{2}, \frac{17}{12}, \frac{577}{408}, \ldots$$

Observe that $(\frac{17}{12})^2 - 2 = \frac{1}{144} \leq \varepsilon$, and hence we can stop at the term a_2. Define a recursive function `approx_sqrt_2` of the form:

```
let rec approx_sqrt_2 (r : rational) (e: rational) : rational =
    ...
```

You may use the functions defined in the previous problems to define `approx_sqrt_2`. For this purpose you may also need to define auxiliary (helper) functions such as for computing the absolute value, comparison ($\leq$) operator, squaring, etc. Please define these auxiliary (helper) functions as a part of your code.

Exploration (Optional). The recursive method to approximate square roots can be extended to finding roots of arbitrary polynomials. Given a polynomial $f(x)$, its root can be approximated using the recurrence relation:

$$a_{i+1} = a_i - \frac{f(a_i)}{f'(a_i)} \tag{3}$$

Where $f'(x)$ is the derivative of $f(x)$. You can implement a program to approximate the roots of polynomials (given a fixed polynomial).