universität
innsbruck

Functional Programming | WS 2020 | LVA 703025

Exercise Sheet 8, 6 points | Deadline: Wednesday, December 2, 2020, 6am

- Please write all the Haskell code into a single .hs-file and upload it in OLAT.
- You can use the template .hs-file that is provided on the proseminar page.
- Your .hs-file should be compilable with ghci.
- Don't forget to mark your completed exercises in OLAT.
- **On December 2, there is no proseminar because of the inday students. Therefore, sheets 8 and 9 will be shorter and both sheets will be discussed on December 9. Note that sheet 8 still has to be solved by December 2.**

  We highly recommend to attend the program at the inday students, in particular since there will be a talk on *Escape from the Ivory Tower: The Haskell Journey* by Simon Peyton Jones, the inventor and developer of Haskell.
  More information on the inday is available at:

  https://www.uibk.ac.at/informatik/studium/inday-students/

Exercise 8.1 *Approximation algorithm* **2 p.**

Implement a slight variation of the approximation algorithm for the square root from the lecture. First, consider the following series x which converges and its limit is $\sqrt{a}$.

$$x_0 = a \qquad\qquad x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$$

Implement the function `approx :: (Fractional a, Ord a) => a -> a -> (a, Integer)`. `approx a e` should approximate the square root of `a` by using the iterative method from above. Let x_i be the approximation in the i-th step, then `approx` should stop if $|x_i^2 - \texttt{a}| < \texttt{e}$ and return the tuple (x_i, i). That means, it returns an approximation that is within the specified error and the number of iterations it took to calculate the approximation. In the other case, it improves the approximation using the formula from above.
You can test this function with the data types `Double`, `Float` and `Rational`. `Rational` is Haskell's built-in data type for rational numbers. For using it, you have to `import Data.Ratio` at the start of your file and then you can construct rational numbers with the `(%)` operator (see examples).

Examples:
```
approx (1024 :: Double) 0.001 == (32.0000071648159, 8)
approx 2 10 == (2, 0) -- here the error value is so high that no iteration is needed
approx (2 % 1) (1 % 10) == (17 % 12,2)
approx (1024 :: Float) 0.0001 == (32.0,9)
approx (1024 :: Double) 0.0001 == (32.0000000000008,9)
```
(2 points)

Exercise 8.2 *Instantiating* `Num`*, representing integers as strings of* `-`,`+` **4 p.**

Throughout history people have used different ways to represent numbers. Starting with tally marks in the Upper Paleolithic, various other representations were developed e.g. the sexagesimal system of the Sumerians, decimal and binary systems in various cultures, and the Roman numerals. Each of these is a good representation, in fact all of them are still in use today, since each allows one to do the basic operations, such as addition and multiplication (although they are quite awkward for Roman numerals).
Haskell's `Num` type class allows for such different representations of numbers, guaranteeing that the basic numerical operations can be performed on them; see the Prelude documentation for its exact functionality.
This exercise asks you to make an instance of the `Num` type class for 'positive and negative tally marks'. Such a number is a string of `+` and `-` symbols, with the former standing for +1 and the latter for −1, with the string as a whole representing the sum of these (the empty string represents the empty sum, 0). Thus, we define:

```
data MP = MP String
```

For example, each of `MP "+-+-+"` and `MP "--+++"` and `MP "+"` represents 1, and `MP "---+-"` represent −3.

1. Write a function `negateString` that *negates* a string by replacing each `+` by `-` and vice versa, corresponding to negating the number represented. Also write a function `nrmString` that *normalises* the representation such that the string no longer contains both `+` and `-`, without changing the number the string represents. That is, `+` and `-` should be removed in pairs, based on that $1 - 1 = 0 = -1 + 1$. The result should be either a string of "+"s or a string of "-"s (or the empty string).

   ```
   negateString :: String -> String
   nrmString :: String -> String
   ```

   Example:

   ```
   nrmString "+-+-++" = "++"
   nrmString "-+-+--" = "--"
   nrmString "---++-++" = ""
   negateString "+-+-++" = "-+-+--"
   negateString "-+-+--" = "+-+-++"
   ```

   Applying `negateString` twice should yield the original string, and applying `nrmString` twice should be no different from applying it once. (1 point)

2. Make `MP` an instance of both `Show`[1] and `Eq` such that showing `MP s` just shows `s`, and two `MP`s are equal if their normalised strings (as in the previous item) are equal as strings.

   Example:

   ```
   (MP "+-+-+") == (MP "--+++") = True
   (MP "+-+-+") == (MP "") = False
   (MP "+-+-+") = "+-+-+"
   ```

   with the last line indicating the result of showing the expression. (1 point)

3. Make `MP` an instance of `Num`. To that end, look up what functions should be defined in any instance of `Num` (see for this so-called minimal complete definition the documentation of the Haskell Prelude `https://hackage.haskell.org/package/base-4.14.0.0/docs/Prelude.html`).

   In your definitions of these functions *no* (other) `Num` instances may be used (no `Integer`, `Float`, ...).[2] E.g. you are not allowed to define `(+)` on `MP`s by first converting to `Integer`s, add those, and then convert back to `MP` again. Hint: The functions defined in the first item may be useful. Hint for implementing multiplication: it can be viewed as repeated addition. For instance, multiplying `+++` with `--` can be done by adding `--` to `--` to `--`.

   Also implement a function `fromMP` that converts an `MP` into an `Integer`, such that converting back (using `fromInteger`) to `MP` should yield an `MP` that is `==` to the original one.

   ```
   fromMP :: MP -> Integer
   ```

   Defining `three,four,mfive :: MP` as

   ```
   three = fromInteger 3
   four = fromInteger 4
   mfive = fromInteger (-5)
   ```

[1] See slide 58 of part 3 for details on the `Show`-class.
[2] The exception is of course `fromInteger`.

We should have:

```
fromMP ((three + four) * mfive) = -35
fromMP (four * (three + mfive)) = -8
```

(2 points)