Machine Problem: Basic Functions

Overview

This machine problem is in 2 parts. In part 1, we give you a number of polymorphic type declarations, and you'll provide working definitions. In part 2, you'll implement a number of different functions of varying complexity, given their specifications.

Starter code repository

Each machine problem will require you to access and clone a separate, private Git repository containing the starter codebase, implement and test your solution on your own machine, then push your changes so we can evaluate them.

You will claim your private repository using a GitHub invitation link -- these will always be found next to the assignment writeup links on the course website. For this assignment, the invitation link is https://classroom.github.com/a/cCuTvXXX. After following the link, you'll be prompted to create an account on GitHub (or sign in, if you already have one) and pick your IIT username from a list to accept the assignment. GitHub will then clone the starter code into a private repository and take you to a URL that looks something like https://github.com/cs340ppp/mp1-USER (where *USER* is your own GitHub username).

This is your repository's homepage on GitHub. You can always come back here to see the status of your work as reflected on GitHub. To submit work you will push your committed changes to this repository, and we will pull them for grading. Remember, if your work isn't here we can't see it!

Next, you should clone this repository on the computer where you'll be doing your work. On your repository's GitHub page, you should see a green button labeled "Clone or download". Click it to obtain the repository URL. You can then use the Git client of your choice (command line or graphical) to clone it locally.

Before continuing, take a moment to sign the "package.yaml" file in your repository. Edit the values in the the "author" and "maintainer" lines, replacing them with your own name and email address, in so doing also signing the honor pledge. You must do this *for every machine problem*.

Finally, a heads-up: the first time you run `stack build` in the repository (or any other command that triggers compilation) it will likely take a while, as we included a graphics library in the dependencies list which takes some time to build. If you encounter a build failure (which is likely on Windows), you should follow the instructions in this video to install a library called freeglut to fix it. Reach out to the TA if you need help.

Part 1

All your code for this machine problem will go into "src/MP1.hs", which already contains starter code. Note that the source files in the repository are *not* literate Haskell source files, so don't start lines with ">" characters!

In "MP1.hs" you'll find the functions `p1_1`, `p1_2`, `p1_3`, and `p1_4`, all of which have polymorphic type declarations but lack definitions. Based solely on the type declarations, provide a definition for each function that uses its arguments (in some cases, not all of them!) to compute and return the correct result type.

Part 2

Each of the 7 exercises below corresponds to a function in "MP1.hs" that you must implement, in some cases using recursion. For each exercise we provide a description and a top-level function declaration which you mustn't change.

In your implementations you may use any language construct --- e.g., pattern matching, guards, `let`/`if-then-else` expressions, `where` clauses, etc. You may also use any of the `Prelude` functions discussed in class so far; *check with us first* if you want to skip ahead and use built-in data types or other library functions that I've yet to mention in class!

1. `transposeTup`

Task: Transpose a 2-row x 2-column tuple.

```
transposeTup :: ((a,a),(a,a))  -- input matrix
             -> ((a,a),(a,a))  -- transposed matrix
```

To "transpose" a matrix is to interchange its rows and columns --- i.e., row 1 becomes column 1, row 2 becomes column 2, etc.. It'll be nice when we can deal with matrices of arbitrary size, won't it?

Examples:

- `transposeTup ((1,2),(3,4)) = ((1,3),(2,4))`

2. `sort3Tup`

Task: Sort the elements of a 3-tuple.

```
sort3Tup :: Ord a
         => (a,a,a)  -- input 3-tuple
         -> (a,a,a)  -- sorted 3-tuple
```

If you're thinking it's silly to write a function to specifically sort 3-tuples, you're right! Won't it be nice when we can sort collections of arbitrary size?

Examples:

- `sort3Tup (2,1,3) = (1,2,3)`
- `sort3Tup (3,2,1) = (1,2,3)`

3. `compoundInterest`

Task: Compute the compound interest earned.

```
compoundInterest :: Floating a
                 => a   -- principal
                 -> a   -- rate
                 -> Int -- num of compounding periods
                 -> a   -- amount of compound interest earned
```

Compound interest is the interest earned from some quantity over multiple "compounding periods", where for each period the amount of interest is based on the original quantity (the principal) and on accumulated interest from all previous periods.

E.g., given a principal of 100 and a rate of 20%, the first period earns us $100 \times 0.2 = 20$. In the second period we earn $(100 + 20) \times 0.2 = 24$, for a total accumulated interest of $20 + 24 = 44$. In the third period we earn $(100 + 44) \times 0.2 = 28.8$, for a total accumulated interest of $44 + 28.8 = 72.8$.

While there are formulae that will let you directly compute the amount of compound interest given the other parameters, we *strongly encourage* you to implement this function using recursion -- it's good practice!

Examples:

- `compoundInterest 100 0.2 1` = 20
- `compoundInterest 100 0.2 2` = 44
- `compoundInterest 100 0.2 3` = 72.8

4. `collatzLen`

Task: Compute the length of the Collatz sequence starting at the input.

```
collatzLen :: Integer  -- start value of the sequence
           -> Integer  -- length of sequence
```

The Collatz conjecture has to do with the sequence starting with any positive integer n, where each subsequent term in the sequence is defined by applying the function C to the preceding term, where:

$$C(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

The conjecture states that all Collatz sequences eventually end in 1. E.g., if we start with $n = 10$, we have the sequence $10, 5, 16, 8, 4, 2, 1$. Your function will return the length of the Collatz sequence given a starting value for n.

Examples:

- `collatzLen 1` = 1
- `collatzLen 10` = 7
- `collatzLen 27` = 112

5. `newtonsSqrt`

Task: Compute the square root of the input using Newton's method.

```
newtonsSqrt :: (Floating a, Ord a)
            => a -- x
            -> a -- square root of x
```

Newton's method is an ingenious algorithm that generates successively better approximations to the solutions of an equation. It is an instance of a more general technique we'll discuss later on known as *fixed-point iteration*.

Applied to the problem of finding the square root of some number x, it works as follows:

1. Start with some guess g
2. Check if our guess is the solution we want -- i.e., if $g^2 = x$. If so, we are done.
3. If not, *improve our guess* and repeat step 2. To do this, we need a formula to compute a new guess g' where g'^2 is *closer* to x then g^2. We can use the formula:

$$g' = \frac{g + \frac{x}{g}}{2}$$

E.g., say we want to compute the square root of 2. We can start with the guess 1. $1^2 \neq 2$, so we compute a new guess:

$$\frac{1 + \frac{2}{1}}{2} = 1.5$$

$1.5^2 = 2.25 \neq 2$, so we compute a new guess:

$$\frac{1.5 + \frac{2}{1.5}}{2} = 1.41666\ldots$$

$(1.41666\ldots)^2 = 2.069444\ldots \neq 2$, so we compute a new guess:

$$\frac{1.41666\ldots + \frac{2}{1.41666\ldots}}{2} = 1.41421\ldots$$

$(1.41421\ldots)^2 = 2.0000\ldots$. If we want to get a more accurate root, we can keep going, but we'll stop here. For a given input x, your solution should find a root r with error tolerance $\varepsilon = 0.0001$; i.e., $|x - r^2| < 0.0001$

Examples:

- `newtonsSqrt 2` ≈ 1.4142...
- `newtonsSqrt 1000` ≈ 31.6227...

HINTS:

- you should implement this using recursion (at least to start)
- we recommend defining -- at a minimum -- the helper functions `goodEnough` and `improve`, in a `where` clause to keep your code legible
- after you get the recursive version working, check out the `until` library function --- see if you can write `newtonsSqrt` using `until`

6. `drawOrbit`

Task: Draw a planet in a circular orbit given an orbital radius and period.

```
drawOrbit :: Float  -- radius
          -> Float  -- period
          -> Float  -- time
          -> Picture
```

For this and the next function we'll be using a graphics library called Gloss, which makes it easy (and fun!) to create drawings and animations in Haskell.

A planet in a uniform, circular orbit with radius r and period p can have its position described (relative to the origin) at time t with the polar coordinate $(r, \frac{2\pi t}{p})$. Recall that converting between polar (r, θ) and Cartesian (x, y) coordinates can be done with the equations:

$$x = r\cos\theta$$

$$y = r\sin\theta$$

To produce a picture of a "planet" in Gloss, we will draw a simple, solid circle. We can do this with the function call `circleSolid 10`. To draw the circle at Cartesian coordinate (x, y), we would do `translate x y (circleSolid 10)`. Note that both `circleSolid` and `translate` (documentation linked) return type `Picture`, which is what `drawOrbit` must also return.

We call `drawOrbit` for you from "Main.hs", and when you're done with your implementation and want to test it out, do a fresh `stack build` then run the executable by entering the command `stack exec mp1` (in your repository, outside of GHCi). If all is correct, this

should bring up an animation of the circular orbit in a separate Gloss viewer window. Close the window by hitting the Escape key.

If you look in "Main.hs", you'll find the relevant line:

```
color azure  $ drawOrbit 150 8 t
```

Which is calling your function with fixed values of `r` and `p`, and a (perpetually increasing) time `t`, then coloring the returned "planet" azure and drawing it in a window of size 500 by 500 centered at the origin.

7. `drawOrbit'`

Task: Draw a planet in an *elliptical* orbit based on Kepler's equation.

```
drawOrbit' :: Float  -- semi-major axis
           -> Float  -- eccentricity
           -> Float  -- period
           -> Float  -- time
           -> Picture
```

Now for a bit of a challenge. Kepler's laws describe the path of a planet on an *elliptical* orbit around the sun. In particular, we can use Kepler's equation to compute the polar coordinates (r, θ) of a planet at time t based on these parameters:

- a: the semi-major axis (half of the longest diameter of the ellipse)
- ε: the *eccentricity* of the ellipse, $(0 \leq \varepsilon < 1)$
- P: the period of an orbit

Calculating (r, θ) requires five steps:

1. Compute the *mean motion*
2. Compute the *mean anomaly*
3. Compute the *eccentric anomaly*
4. Compute the *true anomaly* (i.e., θ)
5. Compute the *heliocentric distance* (i.e., r)

Read through the "Position as a function of time" section of the Wikipedia entry on Kepler's laws for the details. We recommend defining a separate top-level function for each of the five steps.

"Main.hs" also contains code for calling this new `drawOrbit'` function, and compositing its returned images with those from `drawOrbit`. After successfully completing your implementation, `stack exec mp1` should reward you with a composite animation of a circular and elliptical orbit.

Good luck!

HINTS:

- Computing the eccentric anomaly in step 3 requires solving for E in the equation

$$M = E - \varepsilon \sin E$$

  Good news: we can do this using Newton's method, from earlier! We can rewrite the equation as

$$E = M + \varepsilon \sin E$$

and we can try to guess a value g for E. If our guess is off, then g is either greater or less than $M + \varepsilon \sin g$ (i.e., the two sides of the equation aren't equal), and we can improve our guess by computing:

$$g' = \frac{g + (M + \varepsilon \sin g)}{2}$$

and we can keep repeating this until our guess is good enough.

- Solving for the true anomaly in step 4 just requires a bit of algebraic manipulation. We want to solve for θ in

$$(1 - \varepsilon) \tan^2 \frac{\theta}{2} = (1 + \varepsilon) \tan^2 \frac{E}{2}$$

Some manipulation gives us:

$$\theta = 2 \arctan \left(\sqrt{\frac{1 + \varepsilon}{1 - \varepsilon}} \tan \frac{E}{2} \right)$$

$\arctan$ is available in Haskell as `atan`

Evaluation

Part 1

- Each function that compiles correctly with your added type declaration is worth 3 points.

Part 2

- Each exercise is worth 5 points. If your code passes all our tests and uses only permitted functions you get full credit. Partial credit may be awarded for functions that only work for some inputs. Code that doesn't compile or uses prohibited functions will receive no credit.

Maximum points = $3 \times 4 + 5 \times 7 = 47$.

Submission

First, make sure you correctly signed and committed the "package.yaml" file. We won't be able to easily map your repository to your name if you don't!

To submit your work, simply commit all changes and push to your GitHub repository.

Last updated: Mon Jan 31 23:54:06 2022