CS355: Programming Paradigms Lab

Lab 3: Data Abstraction

August 19^{th}, 2024

Q1. Define a recursive procedure `gcd` that takes two positive numbers as arguments and returns their greatest common divisor. You can use *Euclid's algorithm*, which states that the gcd of two numbers `a` and `b` is the same as the gcd of `b` and `r`, where `r` is the remainder when `a` is divided by `b`.

Q2. Notice that the implementation of rational numbers that we used in the class does not reduce a rational number to its lowest terms. For example, after multiplying `2/3` and `3/4`, our implementation would give `6/12` instead of `1/2`.

(A) Use the `gcd` procedure from Q1 to give reduced rational numbers as the outcomes of our rational-number operations.

(B) Notice that you can apply the reduction logic either in `make-rat`, or in the `numer` and `denom` procedures. Explain to your TA which is better.

Q3. Recall coordinate geometry. What is a point in a 2D plane? A pair of x and y coordinates. Similar to rational numbers, define procedures `make-point`, `get-x` and `get-y` that return a 2D point, its x-coordinate and y-coordinate, respectively (you are free to choose any implementation strategy). The following should work:

```
(define p (make-point 2 3))
(get-x p)
> 2
(get-y p)
> 3
```

What is the simplest combination we can build up using points? A straight line!

(A) Write a `make-line` function that constructs a line.
You know what's coming next:

(B) Write functions `get-first-point` and `get-second-point` that take a line and return its start and end points, respectively.

There is no end to abstraction. So next:

(C) Write functions `get-x1`, `get-y1`, `get-x2` and `get-y2` that should take a line and use the above functions to retrieve the respective coordinates of each end point of the line.

Now let's start creating more points and more lines. Define the following:

(D) A function `mid-point` that takes a line and returns a point consisting of the x and y coordinates of the center of that line.

(E) A function `length` that returns the length of the line taken as input.

(F) A function `rotated-line` that rotates a line `{(x1,y1),(x2,y2)}` clock-wise by 90°, such that the start point of the new line is `(x2,y2)`.

(G) Two points `p1` and `p2`, a line `ln` between `p1` and `p2`, and the mid-point `pMid` of `ln`.

(H) Play with the defined lines and points to make sure they work as expected.

Finally, replace the header `#lang sicp` with the following:

```
#lang racket
(require 2htdp/image)
```

and paste the following in the interpreter:

```
(define (draw-p lnV lnH pMid length)
  (let ((vx2 (get-x2 lnV))
        (vy2 (get-y2 lnV))
        (hx2 (get-x2 lnH))
        (hy2 (get-y2 lnH)))
    (let ((i1 (line vx2 vy2 "black")))
      (let ((i2 (add-line i1 0 0 hx2 hy2 "black")))
        (let ((i3 (add-line i2 hx2 hy2 hx2 (- vy2 (/ length 2)) "black")))
          (add-line i3 hx2 (- vy2 (/ length 2)) (get-x pMid) (get-y pMid) "black"))))))
```

Call the above function as follows:

```
(draw-p ln (rotated-line ln) pMid (length ln))
```

Report what shape does DrRacket react with. Change the above function to get different shapes. If further enthusiastic, Google/DDG “drawings in drracket” and enjoy!