CS302: Paradigms of Programming

Lab 2: Representing Data using Functions

March 2^{nd}, 2020

Usually we are taught that programs consist of code and data. In Scheme, we will gradually be observing that both are essentially the same – text surrounded by a forest of parentheses! :-)

In the last lab we saw that we could represent numbers and operations over numbers with lambdas themselves (as *Church numerals*). Today let us use lambdas to build complex data items, in a way realizing that like different special forms (such as `let`), even different data items could be expressed using functions.

Recall coordinate geometry. What's a point in a 2D plane? A pair of x and y coordinates. Do we know how to represent coordinates together as a point in Scheme? Not yet. But we know that this λ stuff is very powerful, so let's try representing a point using a lambda:

```
(define (make-point x y)
  (lambda (bit)
    (if (= bit 0) x y)))
```

Notice what do we have above. The function `make-point` returns a lambda (read 'point'), such that the parameters (read 'coordinates') `x` and `y` are encapsulated[1] in the closure of the returned function. The next task is to be able to access the x and y coordinates. Our $point$ lambda takes a bit to either return the x or the y coordinate, so we could do something like this:

```
(define (get-x point) (point 0))
(define (get-y point) (point 1))
```

Convince yourself that the following code will work:

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

What's the simplest combination we can build up using points? A straight line! Which gives the first exercise for today's lab:

Q1. Write a `make-line` function that constructs a line.

[1] You might get a coffee if you can build-up on this fact when we later learn about the OO paradigm!

You know what's coming next:

Q2. 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:

Q3. 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:

Q4. 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.

Q5. A function `length` that returns the length of the line taken as input.

Q6. 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)`.

Q7. Two points `p1` and `p2`, a line `ln` between `p1` and `p2`, and the mid-point `pmid` of `ln`.

Q8. 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 (poorly written) 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!