Homework 3: More Image Processing with Higher-Order Functions

Due Wednesday, February 16 at 11:59pm

Programming Task

1. Write a function called `blurPixel` with the following type:
   ```
   blurPixel(img: Image, x: number, y: number): Pixel
   ```
   The result is the blurred value of the pixel at coordinates *(x, y)*. To do this, consider the red, green, and blue channel of each pixel independently. The red channel of the resulting pixel should be the mean of the red channels of the pixels neighboring (*x, y*) and the (*x, y*) pixel itself. Compute the blue and green channels in the same way. Two distinct pixels are neighbors if both their *x*-coordinates and *y*-coordinates differ by at most 1 in absolute value. Avoid code duplication.

2. Write a function called `blurImage` with the following type:
   ```
   blurImage(img: Image): Image
   ```
   The result must be a new image that is the blurred version of the argument, with pixels obtained by applying `blurPixel` to each pixel of the input image. You may not use loops within this function. Instead, use one of the higher-order functions defined in the previous homeworks.

3. Write a function called `diffLeft` with the following type:
   ```
   diffLeft(img: Image, x: number, y: number): Pixel
   ```
   which returns a grayscale pixel representing the intensity difference between pixel *(x, y)* and pixel *(x-1, y)* if the latter exists (otherwise, consider there is no intensity difference). Calculate the mean value of the three color channels for pixel *(x, y)* (call it *m1*) and the mean value of pixel *(x-1, y)* (call it *m2*). Set all three channels of the result to $|m2-m1|$.

4. Write a function called `highlightEdges` that takes an image as argument and returns a new grayscale version of the input image with highlighted edges. To do so, apply `diffLeft` to each pixel. Do not use loops. Use one of the higher-order functions defined in the past homeworks.

5. Write a function called `reduceFunctions` with the following type:
   ```
   reduceFunctions(fa: ((p: Pixel) => Pixel)[] ): ((x: Pixel) => Pixel)
   ```
   That is, `reduceFunctions` takes an array of functions, each taking a pixel and returning a pixel. It returns a single function (also from pixel to pixel) that composes all functions in the array, with the function at index 0 applied to the pixel first. Use `reduce()` to implement it.

6. Write a function called `combineThree` with the following type:
   ```
   combineThree(img: Image): Image
   ```
   The result is a new image where each pixel is transformed successively as done by the functions `makeGrayish`, `blackenLow` and `shiftRGB`, in this order. Use `imageMap` and `reduceFunctions.`

Note: your functions must not produce any run-time errors, irrespective of the input image dimensions. You should write tests for your functions, following the same principles used so far.