Homework 3: More Image Processing

Due Wednesday, September 28 at 11:59pm

Programming Task

1. Write a function `lineBlur3p` with the following type:
   ```
   lineBlur3p(img: Image, lineNo: number): void
   ```
   The function should modify the image, only for any pixels having y-coordinate equal to `lineNo`. The new value of each color channel is computed as a weighted sum of the original color value in the pixel and in its neighbor(s) on that line. The weight for any neighbor is `1/3`, and the weight for the original pixel is (1 - sum of neighbor weights).
2. Write a function `lineBlur5p` with the following type:
   ```
   lineBlur5p(img: Image, lineNo: number): void
   ```
   The function should modify the image, only for any pixels having y-coordinate equal to `lineNo`. The new value of each color channel is computed as a weighted sum of the original color value in the pixel and in all other pixels on that line which are at distance up to 2. The weight for any such pixel is 1/5, and the weight for the original pixel is (1 - sum of other pixel weights).
3. Write a function `blurLines` with the following type:
   ```
   blurLines(img: Image, blurLine: (img: Image, lineNo: number) => void): Image
   ```
   It returns a new image. In this image, each line has been blurred using `blurLine`.
4. Write a function called `pixelBlur` with the following type:
   ```
   pixelBlur(img: Image, x: number, y: number): Pixel
   ```
   The result is the blurred value of the pixel at coordinates (x, y), assumed to be valid for the image. Each color channel of the resulting pixel should be the mean of the same channels of the (x, y) pixel itself and its neighbors in `img`. Two distinct pixels are neighbors if both their x-coordinates and y-coordinates differ by at most 1 in absolute value. Avoid code duplication.
5. Write a function `imageBlur` with the following type:
   ```
   imageBlur(img: Image): Image
   ```
   The result is a new image that is the blurred version of the argument, with pixels obtained by applying `pixelBlur` to each pixel of the input image. You may not use loops within this function. Instead, use `imageMapCoord` from HW2.
6. Write a function called `composeFunctions` with the following type:
   ```
   composeFunctions(fa: ((p: Pixel) => Pixel)[] ): ((x: Pixel) => Pixel)
   ```
   It returns a single function (from pixel to pixel) that composes all functions in the argument array, with the function at index 0 applied to the pixel first. Use `reduce()` to implement it.
7. 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 `removeRed`, and then twice by `flipColors`, in this order. Use `imageMap` (of HW1) and `composeFunctions.`

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