Project 2: Higher-Order Functions for Image Processing

Introduction

We have learned how to use higher order functions to perform tasks on arrays. We also learned in Project 1, how to write several image filtering functions. However, note that all the filtering functions in that project had a common structure: an iteration over the 2D array, and the computation of each pixel value. In this project, you will combine these two concepts to define higher order functions for image processing and use them to write write image processing functions with far less code than in Project 1.

In the rest of this document, we use the types `Pixel` and `Image` to describe our functions:

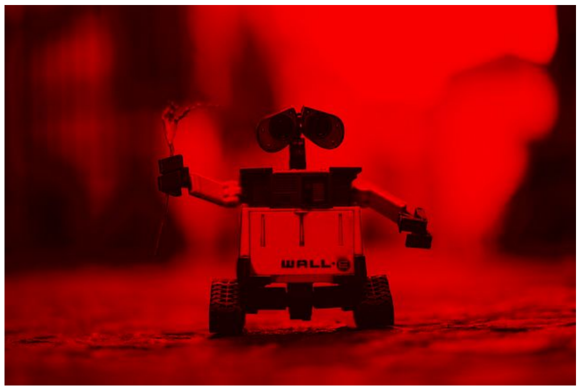
1. A **Pixel** is a three-element array, where each element is a number in the range 0.0 to 1.0 inclusive.
2. An **Image** is a 2D array of Pixels.

Programming Task

1. Write a function called `imageMap` with the following type:

```
imageMap(img: Image, func: (img: Image, x: number, y: number) => Pixel): Image
```

The result must be a new image with the same dimensions as `img`. The value of each pixel in the new image, should be the result of applying `func` to each pixel of `img`.



Your function will have use either `lib220.createImage` or `image.copy` to create a new image, or make a copy of the original. Note that the `fillColor` parameter for `lib220.createImage` needs to be a pixel value. The figure on the right shows an example output of using the `imageMap` function as follows:

```
let url = 'https://people.cs.umass.edu/~joydeepb/robot.jpg';
let robot = lib220.loadImageFromURL(url);
imageMap(robot, function(img, x, y) {
  const c = img.getPixel(x, y);
  return [c[0], 0, 0];
}).show();
```

2. Write a function called `imageMask` with the following type:

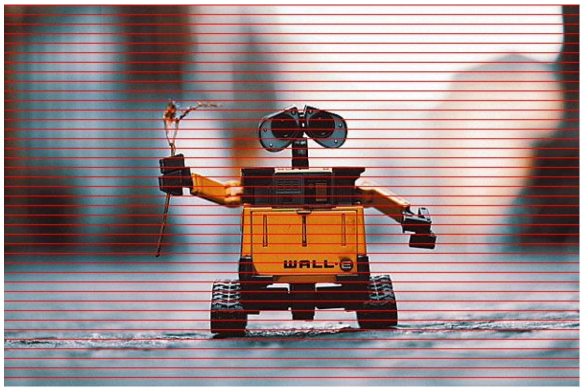
```
imageMask(img: Image,
          func: (img: Image, x: number, y: number) => boolean,
          maskValue: Pixel): Image
```

The result must be a new image, where the value of pixel *(x, y)* is either (a) identical to the value of the pixel *(x, y)* in the original image when `func(img, x, y)` returns `false` or (b) the value `maskValue` when `func(img, x, y)` returns `true`.

You may not use loops within this function. Instead, use imageMap defined above.

The following image shows an example output of using the `imageMask` function:

```
let url = 'https://people.cs.umass.edu/~joydeepb/robot.jpg';
let robot = lib220.loadImageFromURL(url);
imageMask(robot, function(img, x, y) {
  return (y % 10 === 0);}, [1, 0, 0]).show();
```



3. Write a function called `blurPixel` with the following type:

```
blurPixel(img: Image, x: number, y: number): Pixel
```

The result must be the blurred value of the pixel at coordinates *(x, y)*. To calculate the blurred value, take the mean of all the individual color channels for the five pixels to the left of x,y, and five pixels to the right of x, y (i.e. over 11 pixels in total).

4. 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. The output of this function should be identical in behavior to the `blur` function in Programming 1.

You may not use loops within this function. Instead, use one of the higher-order functions defined above.

5. Write a function called `isDark` with the following type:

   ```
   isDark(img: Image, x: number, y: number): boolean
   ```

   The result must be true if all the color channels of the pixel at *(x, y)* are less than 0.5, and false otherwise.

6. Write a function called `darken` with the following type:

   ```
   darken(img: Image): Image
   ```

   The result must be a new image, where each dark pixel is replaced with black (pixel value `[0, 0, 0]`) and other pixels are identical to the corresponding pixel in `img`.

   You may not use loops within this function. Instead, use one of the higher-order functions defined above.

7. Write a function called `isLight` with the following type:

   ```
   isLight(img: Image, x: number, y: number): boolean
   ```

   The result must be true if all the color channels of the pixel at *(x, y)* are greater than, or equal to 0.5, and false otherwise.

8. Write a function called `lighten` with the following type:

   ```
   lighten(img: Image): Image
   ```

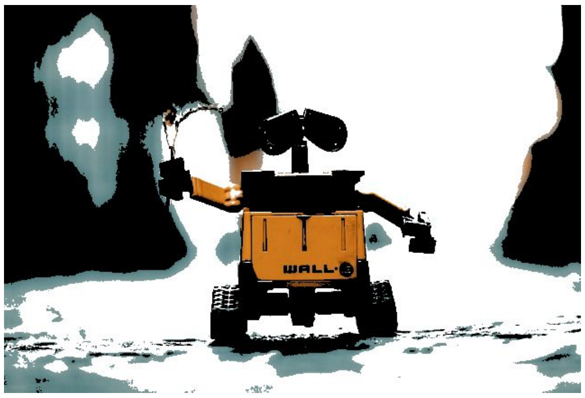
   The result must be a new image`,` where each light pixel is replaced with white (pixel value `[1, 1, 1]`) and other are identical to the corresponding pixel in `img`.

   You may not use loops within this function. Instead, use one of the higher-order functions defined above.

9. Write a function called `lightenAndDarken` with the following type:

   ```
   lightenAndDarken(img: Image): Image
   ```

   The function must first apply the lighten operation and then the darken operation to the image, to produce an image with black and white sections. An example result is shown below.

   You may not use loops within this function. Instead, use one of the higher-order functions defined above.

Note: in the above functions, that your code must not produce any run-time errors, irrespective of the input image dimensions.

Testing Your Code

As you already know, an important part of this project is testing your code thoroughly. In this project, in addition to testing with a variety of input images, you will also have to test with a variety of input *functions* to test your higher-order functions for correctness. To help you get started, we have provided a few test cases here. It is up to you to define additional tests to check your solution for correctness.

- The value returned by imageMap should be an image, and it must be distinct from the input image.

```
test('imageMap function definition is correct', function() {
  let identityFunction = function(image, x, y) {
    return image.getPixel(x, y);
  };
  let inputImage = lib220.createImage(10, 10, [0, 0, 0]);
  let outputImage = imageMap(inputImage, identityFunction);
  // Output should be an image, so getPixel must work without errors.
  let p = outputImage.getPixel(0, 0);
  assert(p[0] === 0);
```

```
  assert(p[1] === 0);
  assert(p[2] === 0);
  assert(inputImage !== outputImage);
});
```

- Test an identity function with imageMap. The resulting image should be unchanged. For this test, we will re-use the pixel equality testing helper function from project 1.

```
function pixelEq (p1, p2) {
  const epsilon = 0.002;
  for (let i = 0; i < 3; ++i) {
    if (Math.abs(p1[i] - p2[i]) > epsilon) {
      return false;
    }
  }
  return true;
};

test('identity function with imageMap', function() {
  let identityFunction = function(image, x, y ) {
    return image.getPixel(x, y);
  };
  let inputImage = lib220.createImage(10, 10, [0.2, 0.2, 0.2]);
  inputImage.setPixel(0, 0, [0.5, 0.5, 0.5]);
  inputImage.setPixel(5, 5, [0.1, 0.2, 0.3]);
  inputImage.setPixel(2, 8, [0.9, 0.7, 0.8]);
  let outputImage = imageMap(inputImage, identityFunction);
  assert(pixelEq(outputImage.getPixel(0, 0), [0.5, 0.5, 0.5]));
  assert(pixelEq(outputImage.getPixel(5, 5), [0.1, 0.2, 0.3]));
  assert(pixelEq(outputImage.getPixel(2, 8), [0.9, 0.7, 0.8]));
  assert(pixelEq(outputImage.getPixel(9, 9), [0.2, 0.2, 0.2]));
});
```