Assignment 7/8: Instaλ

due: 3/24 & 3/31

I've got a great new idea for an app for editing pictures. Modern apps for this are all too user friendly, where you select your picture, and then you just apply filters on the screen. Instead, I want to edit my pictures through the command line. Oh yeah, this is gonna be big.

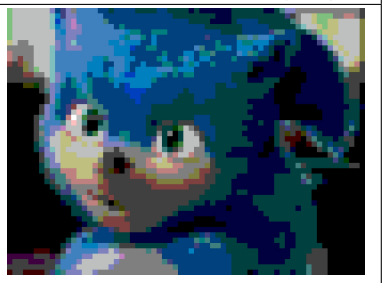
I've created most of the Instaλ app for you already. If you run the app right now, it can all ready read in the command line arguments. There are a lot of filter supported, as you can see by the help message.

```
Usage:
instaLam input output args
-add down image  : read image and add it below the current picture
-add up image    : read image and add it above the current picture
-add right image : read image and add it to the left of the current picture
-add left image  : read image and add it to the right of the current picture
-edge            : find the edges
-cell            : cell shade the image
-gray            : make the image grayscale
-blur n          : blur the image using Gaussian blurring with a size of n
-pixelate n      : pixelate the image with pixels of size nXn
-r90             : rotate the image 90 degrees to the right
-r180            : rotate the image 180 degrees to the right
-r270            : rotate the image 270 degrees to the right
-flip1           : flip the image over the first diagonal
-flip2           : flip the image over the second diagonal
-flipH           : flip the image horizontally (mirror the image)
-flipV           : flip the image vertically
-padH n          : pad the left and right of the image with n black pixels.
-padV n          : pad the top and bottom of the image with n black pixels.
```

We can see an example of this by fixing one of Hollywood's greatest mistakes. Back in 2019, Paramount released a trailer for the upcoming Sonic the Hedgehog movie. The trailer was (rightly) torn apart by the internet. One of the criticisms was that their CGI sonic just didn't look good. You can see what I mean in the picture below. Obviously the problem is that he didn't look like the video game sonic. Well, we have the power to fix that. Since sonic comes from a 16-bit video game with a cartoon art style, we'll cell shade him, and then pixelate the image. I can ran the command following command, and as you can see, we've saved Hollywood!

```
./InstaLam sonic.jpeg sonic_fixed.jpeg -cell -pixelate 8
```

.

Original: `sonic.jpeg`	Fixed: `sonic_fixed.jpeg`
	

Well, this is what should happen. Unfortunately, I haven't had time to write all of the functions yet. This is where I need your help! You need to write all of the filters for this app.

We'll break this into 2 assignments. Assignment 7 we will prepair by writing several useful functions, and a couple of transformations, and in assignment 8 we'll write all of the rotations, flips, and the pixelate functions.

Assignment 7: Picture Fundamentals

For this assignment, we're going to work on several general functions for working with lists, and write our first filters. Let's start with talking about pictures. Our picture is a really just a 2D list of `Pixel`s.

```
data Picture = [[Pixel]]
```

A Pixel has a red, green, and blue component. Each component for a pixel should be between 0 and 1. As an example `Pixel 0 1 0` is a single green pixel.

```
data Pixel = Pixel Double Double Double
```

Pixel functions:
We'll start with a few functions for working with pixels that we'll need next week.

```
pixelScale :: Double -> Pixel -> Pixel
pixelAdd :: Pixel -> Pixel -> Pixel
red :: Pixel -> Double
green :: Pixel -> Double
blue :: Pixel -> Double
```

The `pixelScale s p` should multiply each component in `p` by the values `s`.
Example: `pixelScale 2.0 (Pixel 0 0.5 0.25)` should return `Pixel 0 1 0.5`

`pixelAdd p1 p2` should add the `p1` and `p2` together componentwise.
Exmaple: `pixelAdd (Pixel 0 0.5 1) (Pixel 1 0.5 0)`
should return `Pixel 1 1 1`.

The `red`, `green`, and `blue` functions should return the red, green, or blue component of an image respectively.

list functions:
The next few functions are going to be for general list processing.

```
picMap :: (a -> a) -> [[a]] -> [[a]]
group :: Int -> [a] -> [[a]]
height :: [[a]] -> Int
width :: [[a]] -> Int
```

The `picMap` function is a 2D analog of the `map` function. We still take a function, and applies the function to everything inside of the matrix. You can do this recursively, but its a lot easier to use map to solve this.
Example: `picMap toUpper [['a','b','c'],['d','e','f']]`
returns `[['A','B','C'],['D','E','F']]`

The `group` function is the one we saw in class. This needs to be done recursively. You'll find the `take` and `drop` functions useful.
Example: `group 2 [1,2,3,4,5,6]` should return `[[1,2], [3,4], [5,6]]`

The `height` and `width` should return the height and width of the picture (or 2D list). The height function is pretty straightforward, but there's a technicality with width. The empty list `[]` is a 2D list. There are no rows to find the width of. In this case the width of the picture should be 0.
You'll find the `length` function helpful here.

Picture filters
We can finally get to functions that work with pictures.

```
blackBox :: Int -> Int -> [[Pixel]]
padTop :: Int -> Picture -> Picture
padBottom :: Int -> Picture -> Picture
padLeft :: Int -> Picture -> Picture
padRight :: Int -> Picture -> Picture
padH :: Int -> Picture -> Picture
padV :: Int -> Picture -> Picture
cellShade :: Picture -> Picture
grayScale :: Picture -> Picture
```

We'll start with making a black box of pixels. The `blackBox m n` function should create a box of black pixels with `m` rows and `n` columns.

There are several ways to do this. I recommend the `replicate` function, but you could also use list comprehension, or map.
The `pad` functions should pad the image with a box of black pixels. For example, `padRight 7` should add 7 columns of black pixels to the right of the image. You will find the list concatenation function (`++`) and the `zipWith` function useful here.

The `padH` and `padV` are the first functions that will add functionality to our program. They're very simple functions. `padH n` will add black boxes with of `n` columns to the left and the right of the image, while `padV` add boxes of `n` rows to the top and bottom of the image.

The final two transformations are transformations that are applied to each pixel of the image. First is the `grayScale` transformation. This transformation takes every pixel, and makes it gray. In RGB, a gray pixel has the same value for its red, green, and blue component. So, the easiest way to make a gray pixel is to average the red, green, and blue components.

The final transformation is the `cellShade` transformation. This is also a simple transformation. Modern images look realistic because they can support almost all of the colors we can actually see. If we have a 24-bit image, then each component will have 8 bits, so there are 256 different possible shades of red. In combination with the shades of green and blue, there are $256^3 = 16,777,216$ different possible colors each pixel can have.

The problem is that I want my image to look like a cartoon, and when I draw a cartoon, I don't have that many crayons. I only bought the 64 pack. So, I need to limit the number of colors I use. We can do this pretty easily. For each pixel, we take the components and round them to the nearest third. So, the only values we can have in our pixel are `0, 1/3, 2/3, 1`.

For example `(Pixel 0.2 0.58 0.88)` becomes `(Pixel .333 .666 1)` Now, I can only have $4^3 = 64$ different colors. You'll find that this makes your pictures look a lot more cartoony. This is a common shader used in video games to get a hand-drawn effect.

Running the Program/Submission

You just need to submit the Picture.hs file.

You can run the program with either `ghci` or compile it with `ghc`. I'm going to *strongly* recommend using `ghci` to test your program as you're writing it. Most of these functions work with lists, so you don't need to read an image to test them.

When compiling the program you should use

```
ghc -O2 InstaLam.hs
```

You can then run with

```
./InstaLam input_image output_image -transformations
```

The input and output images should work with most common image types like .bmp .jpg and .png. The -O2 flag is for compiler optimizations. This will be helpful for the next assignment when our transformations get more involved.

Examples

For these examples
`p1 = Pixel 0.1 0.2 0.3`
`p2 = Pixel 0.4 0.5 0.6`
`sonic` is the picture from the `sonic.jpg` file.

name	input	output
`pixelScale`	`pixelScale 2 p1`	`Pixel 0.2 0.4 0.6`
`pixelAdd`	`pixelAdd p1 p2`	`Pixel 0.5 0.7 0.9`
`red`	`red p1`	`0.1`
`green`	`green p1`	`0.2`
`blue`	`blue p1`	`0.3`
`picMap`	`picMap toUpper ["abc","def","ghi"]`	`["ABC","DEF","GHI"]`
`group`	`group 3 [1,2,3,4,5,6,7,8,9]`	`[[1,2,3],[4,5,6],[7,8,9]]`
`height`	`height [[1,2],[3,4],[5,6]]`	`3`
`width`	`width [[1,2],[3,4],[5,6]]`	`2`

blackBox	blackbox 50 100	
padTop	padTop 100 sonic	
padBottom	padBottom 100 sonic	
padLeft	padLeft 100 sonic	
padRight	padRight 100 sonic	

padH	padH 100 sonic	
padV	padV 100 sonic	
cellShade	cellShade sonic	
grayScale	grayScale sonic