# MA 151: Project #2

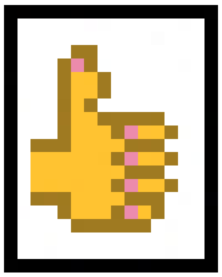
due Friday November 17

## Submissions

This project will be completed in groups of size 1-3, and submitted in the usual way on our class website. One member of each group should submit the code, and all group members' names should be listed in comments at the beginning of the file.

## Overview

This is the second episode of a 3-part project which will focus on digital images. This time we will use custom types to store our images, and introduce color pixels. Here is an example image:



### About color in digital images

All colors can be expressed as combinations of the three "additive primary" colors red, blue, and green. For example yellow is a combination of red and green.[1]

Colors are usually represented by their "RGB values"- every color is specified by how much red, blue, and green it contains. Here are some examples:

| | Composition | RGB code |
|---|---|---|
| 🟥 | 100% red, 0% green, 0% blue | `RGB 255 0 0` |
| 🟩 | 0% red, 100% green, 0% blue | `RGB 0 255 0` |
| 🟦 | 0% red, 0% green, 100% blue | `RGB 0 0 255` |
| 🟨 | 50% red, 50% green, 0% blue | `RGB 127 127 0` |
| ⬛ | 0% red, 0% green, 0% blue | `RGB 0 0 0` |
| ⬜ | 100% red, 100% green, 100% blue | `RGB 255 255 255` |

The individual RGB levels are each stored as an `Int` in the range 0 to 255. This is called "8-bit color depth", since each color is between 0 and $2^8$. This gives us a "pallette" of $(2^8)^3 = 2^{24} = 16777216$ possible different colors to choose from.

[1] This will seem weird if you are used to "primary colors" as red, blue, and yellow, like when you are mixing paints. It turns out the physics of light generated by pigment mixing is different from directly combining light frequencies. Digital images are always produced by mixing light waves directly rather than mixing pigments, and the rules are different.

### Types in this project

Last time, each pixel had type `Bool`, which was good enough for black and white images. This time, we will use a type like this:

```
data Pixel = RGB Int Int Int deriving Show
```

Then a red pixel would be represented as `RGB 255 0 0`, just like in the table up there.

Last time, every image was represented by a triple of type `(Int, Int, [Bool])`. This time, an image will be stored in a type like this:

```
data Image = Im Int Int [Pixel] deriving Show
```

Just like last time, I have provided a `display` function which will display a color image, and I've given you the code for the `thumb` image, this time in color. The `display` function won't actually show you the colors, since the GHCi terminal doesn't do color very well. But you will at least be able to tell the difference between colors.

## Required functions

Write the following functions. Always include a type signature. Start with the file `proj1.hs` from our class website, which includes the `thumb` image and the `display` function.

- Change all your functions from Project 1 to use our new types. Assuming you did them correctly last time, this should be easy, though it might take you a while. You'll need to make small changes throughout all your definitions and type signatures.
- Change `hImage` from Project 1 so that the image looks like a green H on a black background.
- Write a polymorphic function `blocks` which takes an `Int` and a list, and groups the list into blocks of the given length. For example:

  `blocks 4 [1..22]` is `[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[17,18,19,20],[21,22]]`

  Notice that any "extra" elements are just put into their own block at the end. (This function has nothing directly to do with images, but you'll use it a lot below.)
- Write a function called `dropFirstCol` which works like `dropFirstRow`, but removes the first column. Do this using `map`, `blocks`, and `concat`. (Use blocks to make a list of all the rows, then use map to change the rows a little bit, then use concat to stick them all together again.)
- Write a similar function `dropLastCol`, and also `dropFirstCols` and `dropLastCols`.

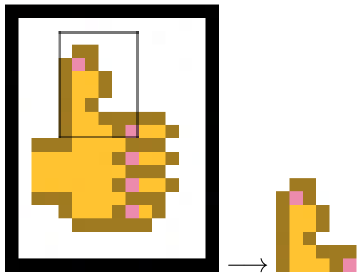
For the next few functions, use the higher order prelude function `zipWith`, which joins together two lists using some function. For example:

`zipWith (++) ["hi","mom","dad"] ["dude","sweet","weird"]` is `["hidude","momsweet","dadweird"]`
`zipWith (:)  [1,2,3] [[4,6,7],[0,0,0],[8,7]]` is `[[1,4,6,7],[2,0,0,0],[3,8,7]]`

- Write functions called `colStackFirst` and `colStackLast` and `colSandwich` which work like `rowStackFirst`, etc.
- Write a function called `hTile` which works like `vTile`, but tiles the image horizontally.
- Write a function called `tile` which does tiling both vertically and horizontally. So `tile 3 4 thumb` should look like the thumb image repeated 3 times horizontally and 4 times vertically. (So you'll see a total of 12 thumbs.)
- Write a function called `border` which takes a `Pixel` and an `Image`, and returns an image with a 1-pixel rectangular border around it, where the color of the border is given by the parameter pixel. So `border (RGB 0 0 255) thumb` should give the thumb image with a blue border around it.

- Write a function called `thickBorder` which works like `border` but lets you specify the border thickness. So `thickBorder 4 (RGB 0 0 255) thumb` gives the thumb image with a blue border 4 pixels thick.
- Write a function called `crop` which takes four `Int`s and an `Image`, and crops the image. The parameters work like this:

  In `crop 5 3 6 8 thumb`, the 5 and 3 define the upper left corner of the crop region (the top left corner is 1 and 1), and the 6 and 8 give the width and height of the cropped image. This looks like:



- Write a function called `rotateCW` which rotates an image 90 degrees clockwise. Also write similar functions `rotateCCW` (for counter-clockwise) and `rotate180`.