Fundamentals of Computer Science I: Media Computing (CS151.02 2007F)

# Transforming Images

**Summary:** We explore how to expand the power of color transformations, first by applying them to images rather than to individual pixels then by combining them into new transformations.

**Contents:**


- From Transforming Pixels to Transforming Images
- Composing Transformations
- Reviewing Key Concepts


## From Transforming Pixels to Transforming Images

If you think back to the end of the previous reading on transforming RGB images, you may recall that we were starting to write filters that transformed not just individual colors but whole images. Here is one set of commands that transforms a four-by-three image called `canvas`.

```
(image.transform-pixel! canvas 0 0 rgb.complement)
(image.transform-pixel! canvas 0 1 rgb.complement)
(image.transform-pixel! canvas 0 2 rgb.complement)
(image.transform-pixel! canvas 1 0 rgb.complement)
(image.transform-pixel! canvas 1 1 rgb.complement)
(image.transform-pixel! canvas 1 2 rgb.complement)
(image.transform-pixel! canvas 2 0 rgb.complement)
(image.transform-pixel! canvas 2 1 rgb.complement)
(image.transform-pixel! canvas 2 2 rgb.complement)
(image.transform-pixel! canvas 3 0 rgb.complement)
(image.transform-pixel! canvas 3 1 rgb.complement)
(image.transform-pixel! canvas 3 2 rgb.complement)
```

That's an awful lot of typing. (Even though it seems that we've been typing a lot over the past few days, we do want to type less, and will look for techniques for doing so.) More importantly, this technique does not adapt itself easily to images of different sizes or to typical-sized images. (In a 200x200 image, which is still fairly small, there are 40,000 positions. No one wants to type that many lines of code, particularly when it is that repetitive.)

So, how do we write filters? DrFu includes a pair of helpful procedures, `(image.map` *`transformation`* `image)` and `(image.map!` *`transformation`* `image)`. The first builds a new image by setting each pixel in the new image to the result of applying the given transformation to every pixel in another image. The second changes an existing image by applying the transformation "in place". You will explore the differences between the two in the lab.

In Scheme, "map" typically means something akin to "do to each portion of this thing". So, `image.map` means "do to each pixel". When we learn about lists, Scheme's primary mechanism for grouping other kinds of values, we'll find that there's a version of map that does something with each element of the list. There's even an `rgb.map` that lets us do something to each component of an RGB color.

You'll note that the order of the parameters in `image.map` is a bit different than you'd expect. In almost every operation so far, the first parameter is the primary object being used or affected. Here, the first parameter is the color transformation being applied to each pixel. Why the difference? Because the tradition with `map` is to make the function the first parameters. As we'll see again and again, there are multiple customs for how one writes or designs procedures, and sometimes these customs come into conflict.

Using `image.map` (or `image.map!`) and the color transformations we learned in the previous reading, we can now transform images in a few basic ways: we can lighten images, darken images, complement images (and perhaps even compliment the resulting images), and so on and so forth.

# Composing Transformations

We are almost done on our quest to learn how to write filters. We've learned some new functions provided by DrFu, such as the transformations. We've learned about one new technique, refactoring, which involves writing new functions that encapsulate common code. However, we have not yet learned how to write these refactored procedures. We've seen that Scheme permits procedures to take other procedures as parameters, and that this permission supports refactoring. Finally, we've learned a new kind of operation, map, which lets us do something to every component of a compound value. All this put together lets us write some simple filters. For example, we can write one line of Scheme code to make a a picture bluer.

```
> (image.map! rgb.bluer picture)
```

But what if we want more interesting filters, ones that can't be described with just a single built-in transformation? One thing that we can do is to combine transformations. There are two ways to transform an image using more than one transformation: You can do each transformation in sequence, or you can use function composition, an old mathematical trick, to first combine the transformations. Consider, for example, the problem of lightening an image and then increasing the red component. We can certainly write the following:

```
> (image.map! rgb.lighter picture)
> (image.map! rgb.redder picture)
```

However, what we really want to do is make each pixel in the picture lighter and then redder. In mathematics, when you want to build a function that does the work of two functions, you *compose* those functions. In DrFu, the `compose` operator lets you combine multiple functions into one. Here's a new function that makes colors lighter and then redder.

```
> (define lr (compose rgb.redder rgb.lighter))
> (define sample (cname->rgb "blue violet"))
> (rgb->string (lr sample))
"183/111/175"
```

Using function composition, we can therefore rewrite the earlier transformation as

```
> (image.map! lr picture)
```

or, as simply,

```
> (image.map! (compose rgb.redder rgb.lighter) picture)
```

What's the big difference between this instruction and the series of two instructions? In effect, we've changed the way you sequence operations. That is, rather than having to write multiple instructions, in sequence, to get something done, we could instead insert information about the sequencing into a single instruction. By using composition, along with nesting, we can then express our algorithms more concisely and often more clearly.

# Reviewing Key Concepts

In most of the initial readings for this course, you were learning some basic operations that you could use in writing algorithms. In this reading, while you learned about new operations, they were a different kind of operation. `image.map`, `image.map!`, and `compose` all provide a way to add *control* to your program. The two maps do something to each component of the image, the composition sequences operations within a single Scheme instruction.

---