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

Laboratory: Representing Images as Lists of Pixels

Summary: In this laboratory, you will explore not only the basic list operations, but also some applications of those operations in working with images.

Contents:

- Preparation
- Exercises
  - Exercise 1: Some Simple Lists
  - Exercise 2: Extracting Information from Lists
  - Exercise 3: Other List Operations
  - Exercise 4: It's So `cons`-fusing
  - Exercise 5: Making Images with `list-ref`
  - Exercise 6: Making Pixel Info
  - Exercise 7: Extracting Pixel Info
  - Exercise 8: Drawing Pixels
  - Exercise 8: Drawing Bigger Pixels
  - Exercise 9: Translating Pixels
- For Those With Extra Time
  - Explorations
- Explorations
  - Exploration 1: Extending Stripes
- Notes
  - Notes on Exercise 4: It's So `cons`-fusing

Preparation

Create a new 100x100 image and name it `canvas`.

Exercises

Exercise 1: Some Simple Lists

a. Call the `cons` procedure to build a list of the value `color.red`. The result of your call should be `(-16776961)`.

b. Call the `cons` procedure to build a list of the value 5 followed by the value of `color.red`. The result of your call should be `(5 -16776961)`. Note that you will need to call `cons` twice to build this list.

c. Call the `cons` procedure to build a list of the value 2, followed by the value 2, followed by the value of `color.red`. The result of your call should be `(2 5 -16776961)`. Note that you will need to call `cons` three times to build this list.

d. Build the same list as in step c, using `list` rather than `cons`.

Exercise 2: Extracting Information from Lists

Consider the following list:

```
(define numbers (list 1 2 3 4 5 6 7 8))
```

a. What do you expect the result of `(car numbers)` to be? Check your answer experimentally.

b. What do you expect the result of `(cdr numbers)` to be? Check your answer experimentally.

c. What do you expect the result of `(car (cdr numbers))` to be? Check your answer experimentally.

d. What do you expect the result of `(cdr (cdr numbers))` to be? Check your answer experimentally.

e. What do you expect the result of `(cdr (car numbers))` to be? Check your answer experimentally.

f. Write an expression that gets the sixth element of `numbers`. (That is, your expression should extract the 5.)

Exercise 3: Other List Operations

a. Create the list `("red" "orange" "yellow")` and name it `roy`. Create the list `("green" "blue")` and name it `gb`..

b. Determine what happens when reverse `roy` with `(reverse roy)`.

c. Determine what happens when you append the two lists together with `(append roy gb)`.

d. Determine what happens when you append the two lists together with `(append gb roy)`.

Exercise 4: It's So `cons`-fusing

As you may recall, the `cons` procedure takes two parameters, a value and a list. It builds a new list by prepending the value to another list. However, it is also possible to apply `cons` to two non-list values. (You should not do so at this point in your career, but some do accidentally, so we want you to see what happens.)

Consider the following:

```
(define one-two (cons 1 2))
(define won-too (list 1 2))
```

a. Execute those definitions and then ask for the values of `one-two` and `won-too`. Explain how the two are and are not similar.

b. What do you expect to have happen when you apply the `list?` predicate to each. Check your answer experimentally.

c. What do you expect to have happen when you call `reverse` on each? Check your answer experimentally.

d. What do you expect to have happen if you try to get the car and the cdr of `one-two` and `won-too`. Check your answer experimentally.

e. What do you expect to have happen if you append each of these lists to the list `(3 4)`, as in the following example?

```
(append one-two (list 3 4))
(append won-too (list 3 4))
```

Check your answer experimentally.

f. What do you expect to have happen if you append the list `(0 0)` to each of these two lists, as in the following example?

```
(append (list 0 0) one-two)
(append (list 0 0) won-too)
```

If you are confused by any of the results, please look at the notes on this problem.

Exercise 5: Making Images with `list-ref`

As you may recall, in your exploration of conditionals, you used conditionals to draw stripes. Another technique for drawing stripes involves using `list-ref` cleverly. Suppose we want five colors of stripes. We create a list of those five colors:

```
(define stripe-colors
  (list (rgb.new 255 0 0)
        (rgb.new 255 255 0)
        (rgb.new 0 255 0)
        (rbg.new 0 255 255)
        (rgb.new 0 0 255)))
```

a. Suppose we want our stripes to be just one column wide. Then we can simply use the column (modulo five) to select the color, as in

```
(region.compute-pixels!
 canvas 0 0 99 99
 (lambda (pos) (list-ref stripe-colors (modulo (position.col pos) 5))))
```

Confirm that this technique works correctly.

b. Extend that code to use seven colors instead of five. That is, you'll need to change the length of the list and use a different modulus.

c. We can make the columns wider by computing the quotient of the column and the desired width. Try expanding your columns to a width of 5 and then 10.

Exercise 6: Making Pixel Info

We claimed that the reading and the lab were about representing images as lists of pixels. However, up to this point, we've just worked with lists. Let's now build some procedures that work with pixels. We've decided to represent each pixel as a three element list, where the first element is the column, the second the row, and the third the color.

When creating a new way to represent data, we often start by writing the constructor.

Write a procedure, `(pixel.new col row color)` that creates a list of the form `(col row color)`.

Exercise 7: Extracting Pixel Info

Of course, we also want to extract information from the pixel.

a. Write a procedure, `(pixel.col pixel)`, that extracts the column from a pixel represented as a three-element list.

b. Write a procedure, `(pixel.row pixel)`, that extracts the row from a pixel represented as a three-element list.

c. Write a procedure, `(pixel.color pixel)`, that extracts the color from a pixel represented as a three-element list.

Exercise 8: Drawing Pixels

Of course, representing pixels this way is not sufficient to draw them. To draw them, we must also put them into an image. Here is a procedure that will do so:

```
(define pixel.draw
  (lambda (image pixel)
    (image.set-pixel! image
                      (pixel.col pixel) (pixel.row pixel)
                      (pixel.color pixel))))
```

Here are a few pixels, put into a list

```
(define pixels
  (list (pixel.new 0 0 (rgb.new 255 0 0))
        (pixel.new 1 0 (rgb.new 128 128 0))
        (pixel.new 2 0 (rgb.new 0 255 0))
        (pixel.new 0 1 (rgb.new 128 0 128))
        (pixel.new 1 1 (rgb.new 0 128 128))
        (pixel.new 0 2 (rgb.new 0 0 255))))
```

Using a sequence of calls to `pixel.draw` and `list-ref`, draw each pixel on canvas.

Exercise 8: Drawing Bigger Pixels

As you've noted many times, when we set just one pixel at a time, it's hard to see the changes. One opportunity we have is to draw "bigger" pixels (that is, we treat a "pixel" as a larger area in an image).

Suppose we decide that we want to represent each pixel in a list as a 10x10 grid of pixels in the image. We can use `region.compute-pixels!` to set the 10x10 area to a single color, as in

```
(region.compute-pixels!
 image
 (* col 10) (* row 10)
 (+ 9 (* col 10)) (+ 9 (* row 10) )
 (lambda (pos) color))
```

a. Write a procedure, (`pixel.decadraw` *`pixel image`*), that draws a 10x10 square for the pixel at the appropriate location in the image. For example, the pixel with column 5 and row 3 would be expanded into the region from (50,30) to (59,39).

b. Test your procedure on the list of pixels from the previous problem.

```
(pixel.decadraw (list-ref pixels 0) canvas)
(pixel.decadraw (list-ref pixels 1) canvas)
(pixel.decadraw (list-ref pixels 2) canvas)
(pixel.decadraw (list-ref pixels 3) canvas)
(pixel.decadraw (list-ref pixels 4) canvas)
(pixel.decadraw (list-ref pixels 5) canvas)
```

Exercise 9: Translating Pixels

One advantage of the three-element lists representation is that we can easily compute modified versions of pixels. For example, we might translate a pixel horizontally or vertically. (In the next lab, we'll see how to do so for a whole list of pixels.)

a. Write a procedure, (`pixel.htrans` *`pixel amt`*), that horizontally translates the given pixel by the specified amount. For example,

```
> (define pix (pixel.new 2 1 color.white))
> pix
(1 1 -1)
> (pixel.htrans pix 5)
(7 1 -1)
> (pixel.vtrans pix -1)
(1 1 -1)
```

b. Write a procedure, (`pixel.vtrans` *`pixel amt`*), the vertically translates the given pixel by the specified amount. (Positive amounts mean to translate down, negative amounts mean to translate up.)

c. Test your procedures by creating a sample pixel and then translating and drawing it with different offsets.

For Those With Extra Time

In exercise 8, you wrote a procedure that "enlarged" pixels by drawing each pixel as a 10x10 grid. Let's generalize that technique. Write a procedure, (`pixel.scaled-draw` *`pixel scale image`*), that draws a *scale*x*scale* square for pixels.

Explorations

Exploration 1: Extending Stripes

In exercise 5, you explored ways to use lists of colors and `list-ref` to draw stripes. We can generalize this technique to draw "interesting" images with a limited palette of colors. Here's what one might do:

- Build a list of colors of some length, say 5.
- Use some formula to compute an integer from the row and column. (For wider stripes, we simply multiplied the column by some number. But you could incorporate the row, and use techniques other than multiplication.)
- Reduce that integer to an appropriate index into the list using `modulo`.
- Use that value to select a color from the list.

Using that technique, write an expression that draws an interesting pattern. The code should look something like this:

```
(define colors (list ___))
(region.compute-pixels!
 canvas 0 0 99 99
 (lambda (pos) (list-ref colors (modulo (___) 5))))
```

Your creative challenges are to choose an appropriate group of colors and a formula that provides an interesting structure.

Notes

Notes on Exercise 4: It's So `cons`-fusing

As you might guess, `won-two` is the list `(1 2)`. As you might not have guessed, `one-two` is the value `(1 . 2)`. That value looks much like a list, but it has a period in the middle. The period is a signal to you that the value is *not* a list.

Since `one-two` is not a list, it is not possible to reverse it or to append it to another list.

However, like the typical implementation of `cons`, the typical implementation of `append` does not confirm that its second parameter is a list. And, like `cons`, when given a non-list as a second parameter, `append` returns a non-list. In this case, `append` returns `(0 0 1 . 2)`. Once again, the period indicates "hey, that's *not* a list".

Why does Scheme permit these non-lists? Because they are a generalization of lists (or at least of the techniques by which we process lists). As we'll see later in the semester, these non-lists can be quite useful.

Copyright © 2007 Janet Davis, Matthew Kluber, and Samuel A. Rebelsky. (Selected materials copyright by John David Stone and Henry Walker and used by permission.) This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/2.5/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.