

## Laboratory: Conditionals

**Summary:** In this laboratory, you will explore two of Scheme's primary conditional control operations, `if` and `cond`.

### Contents:

- Preparation
- Exercises
  - Exercise 1: Comparing Brightness, Revisited
  - Exercise 2: Shades of Grey
  - Exercise 3: Drawing Stripes
  - Exercise 4: Some Shapes
  - Exercise 5: Drawing Lines and Regions
- For Those With Extra Time
  - Extra 1: Distance-Based Computations
  - Extra 2: Checkerboards
  - Extra 3: Shades of Grey, Revisited
  - Extra 4: Drawing Triangles
- Explorations
  - Exploration 1: Two-Color Computed Images
  - Exploration 2: Multi-color Checkerboards
- Notes

### Reference:

If statements typically have the form

```
(if test consequent alternative)
```

Scheme first evaluates the test. If the value of the test is false (`#f`), it evaluates the alternative and returns the value. If the value of the test is truthy (that is, anything that is not false), it evaluates the consequent and returns its value.

Cond statements typically have the form

```
(cond
  (test0 exp0)
  (test1 exp1)
  ...
  (testn expn)
  (else alternative))
```

Scheme evaluates each test in turn until one is truish. It then evaluates the corresponding expression and returns its value. If none of the tests is truish, then it evaluates the alternative and returns its value.

## Preparation

- a. Load a moderate-sized image (no more than about 250x250) and name the loaded image `picture`.
- b. Create a 200x200 image and name it `canvas`.
- c. Create and show a new 3x3 image and name it `grid`.
- d. Zoom in on `grid`.
- e. Fill in the pixels of `grid` as follows. Note that you may want to open this lab in a Web browser and cut and paste.

```
(image.set-pixel! grid 0 0 (rgb.new 192 0 0))
(image.set-pixel! grid 1 0 (rgb.new 192 0 192))
(image.set-pixel! grid 2 0 (rgb.new 0 0 192))
(image.set-pixel! grid 0 1 (rgb.new 192 192 0))
(image.set-pixel! grid 1 1 (rgb.new 0 192 192))
(image.set-pixel! grid 2 1 (rgb.new 255 255 255))
(image.set-pixel! grid 0 2 (rgb.new 0 192 0))
(image.set-pixel! grid 1 2 (rgb.new 0 0 0))
(image.set-pixel! grid 2 2 (rgb.new 192 192 192))
```

## Exercises

### Exercise 1: Comparing Brightness, Revisited

In many recent exercises, we've considered a number of activities that required us to work with the brightness of colors. We typically compute brightness using an expression which can be encapsulated in a procedure as follows:

```
;;; Procedure:
;;;   brightness
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Computes the brightness of color on a 0 (dark) to 100 (light) scale.
;;; Produces:
;;;   b, an integer
;;; Preconditions:
;;;   color is a valid RGB color. That is, each component is between
;;;     0 and 255, inclusive.
;;; Postconditions:
;;;   If color1 is likely to be perceived as lighter than color2,
;;;     then (brightness color1) > (brightness color2).
(define brightness
  (lambda (color)
```

```
(round (* 100 (/ (+ (* .30 (rgb.red color))
                  (* .59 (rgb.green color))
                  (* .11 (rgb.blue color)))
          255))))
```

a. Using brightness, write a procedure, (`brighter color1 color2`), that returns the brighter of `color1` and `color2`.

b. What do you expect the following to accomplish?

```
(define fave (rgb.new 192 0 192))
(image.show (image.map (lambda (color) (brighter fave color)) grid))
```

c. Check your result experimentally.

d. What do you expect the following to accomplish?

```
(define fave (rgb.new 192 192 0))
(image.show (image.map (lambda (color) (brighter fave color)) picture))
```

e. Check your result experimentally.

## Exercise 2: Shades of Grey

Consider the following four colors:

```
(define grey0 (rgb.new 0 0 0))
(define grey1 (rgb.new 96 96 96))
(define grey2 (rgb.new 192 192 192))
(define grey3 (rgb.new 255 255 255))
```

a. Using if statements (and not cond statements), write a procedure, (`rgb.4grey color`), that returns

- `grey0`, if the brightness of `color` is less than 25;
- `grey1`, if the brightness of `color` is at least 25 but less than 50;
- `grey2`, if the brightness of `color` is at least 50 but less than 75; or
- `grey3`, if the brightness of `color` is at least 75.

b. Check the effects of mapping your `rgb.4grey` onto your picture.

c. Rewrite `rgb.4grey` so that it uses `cond` rather than `if`.

## Exercise 3: Drawing Stripes

At times, it may be useful to create images with stripes, vertical or horizontal. How do we do so? Consider the problem of drawing vertical stripes of width 10 in alternating colors. For columns 0 through 9, we draw in the primary color. For columns 10 through 19, we draw in the alternate color. For columns 20 through 29, we draw in the primary color. For columns 30 through 39, we draw in the alternate color. We continue alternating until we reach the end of the image.

How do we express this computationally? Well, you'll note that we have a repeating pattern every twenty columns. Whenever we have repeating patterns, we think of modulo. So, we can start by computing the modulo of the column and 20. Now, we are left with numbers in the range 0 to 19. The values 0 to 9, inclusive, are in the primary color. The values 10 to 19, inclusive, are in the alternate color.

a. Write a predicate, `(primary-column? column)`, that returns `#t` if the given column should be colored in the primary color and `#f` if the given column should be colored in the alternate color. For example,

```
> (primary-column? 5)
#t
> (primary-column? 87)
#t
> (primary-column? 93)
#f
> (primary-column? 32)
#f
```

b. Write a procedure, `(column-color column)` that returns the color black for primary columns and the color red for alternate columns.

c. Create a series of stripes with

```
(region.compute-pixels! canvas 0 0 199 199
  (lambda (pos) (column-color (position.col pos))))
```

d. Rewrite `column-color` to choose between three colors (say, red, black, and white) and then create a new series of stripes. In this rewrite, you will no longer be able to use `primary-column?`. (You may want to think about why you can't use it.)

e. Change the call to `region.compute-pixels!` so that it creates striped rows, rather than striped columns.

## Exercise 4: Some Shapes

As you may recall from the reading, there are a variety of ways we can use conditionals to draw pictures, most typically by drawing in one color if some condition holds and another color if a condition does not hold. Suppose we've prepared to draw by defining a few colors (and the `square` procedure) as follows.

```
(define c0 (rgb.new 255 255 255))
(define c1 (rgb.new 255 0 255))
(define c2 (rgb.new 255 0 0))
(define c3 (rgb.new 0 255 255))
(define square (lambda (x) (* x x)))
```

a. What image do you expect the following code to create?

```
(region.compute-pixels! canvas 0 0 199 199
  (lambda (pos)
    (if (< (position.row pos) (- 100 (position.col pos))) c1 c0)))
```

b. Check your answer experimentally.

c. What image do you expect the following code to create?

```
(region.compute-pixels! canvas 0 0 199 199
  (lambda (pos)
    (if (< (position.row pos) (+ -75 (position.col pos))) c2 c0)))
```

d. Check your answer experimentally.

e. What image do you expect the following code to create?

```
(region.compute-pixels! canvas 0 0 199 199
  (lambda (pos)
    (if (>= (square 50)
        (+ (square (- (position.col pos) 20))
            (square (- (position.row pos) 60))))
        c3 c0)))
```

f. Check your answer experimentally.

g. In the reading, we noted that each of the procedures given in the exercise above eliminates any traces of the previous drawing. Try redefining `c0` as transparent and then running each of the instructions in sequence. What happens?

## Exercise 5: Drawing Lines and Regions

How did we come up with the formulae for the images in the previous problem? For the triangles, it was mostly a matter of working out lines, since we can use lines to border triangular region. So, let's explore that issue.

You should remember that the general formula of a non-vertical line is  $y = mx + b$ . We can therefore draw something that's a bit like a line with `region.compute-pixels!` and a translation of that formula into Scheme. For example, to draw a diagonal line with a slope of 1 in the upper-left-hand corner, we might write

```
(region.compute-pixels!
  canvas 0 0 199 199
  (lambda (pos)
    (if (= (position.row pos) (- 100 (position.col pos))) c1 c0)))
```

Why is the slope 1, rather than -1? Because the image coordinate system is upside-down.

If we replace the `=` with `<` or `>=`, we can color large regions of the image.

a. Using this technique, draw a black line from the upper-left corner of `canvas` to the lower-right corner of `canvas`.

```
(region.compute-pixels!
 canvas 0 0 199 199
 (lambda (pos)
  (if (= (position.row pos) (+ ____ (* ____ (position.col pos))))
      color.black transparent)))
```

b. Using this technique, draw a green line from the upper-right corner of canvas to the lower-left corner of canvas.

```
(region.compute-pixels!
 canvas 0 0 199 199
 (lambda (pos)
  (if (= (position.row pos) (+ ____ (* ____ (position.col pos))))
      color.green transparent)))
```

c. Using this technique, draw a red line from midway down the left side of canvas to the upper-right corner of canvas. (You may have to try a few different formulae to get it right.)

```
(region.compute-pixels!
 canvas 0 0 199 199
 (lambda (pos)
  (if (= (position.row pos) (+ ____ (* ____ (position.col pos))))
      color.red transparent)))
```

d. Using this technique, draw a blue line from midway across the bottom side of canvas to the upper-right corner of canvas. (You may have to try a few different formulae to get it right).

```
(region.compute-pixels!
 canvas 0 0 199 199
 (lambda (pos)
  (if (= (position.row pos) (+ ____ (* ____ (position.col pos))))
      color.blue transparent)))
```

e. You may note that some of the lines you've draw look "incomplete". That is, not all the pixels in the line are connected. The problem, of course, stems from our attempt to represent a continuous line using a discrete set of pixels. We will look at some solutions to this problem later in the semester. For now, you may find that it's easier to explore some slopes by filling in regions, and not just lines. Pick one of your formulae from c or d, and try the following.

```
(region.compute-pixels!
 canvas 0 0 199 199
 (lambda (pos)
  (if (<= (position.row pos) (+ ____ (* ____ (position.col pos))))
      color.brown transparent)))
```

```
(region.compute-pixels!
 canvas 0 0 199 199
 (lambda (pos)
  (if (>= (position.row pos) (+ ____ (* ____ (position.col pos))))
      color.tan transparent)))
```

## For Those With Extra Time

### Extra 1: Distance-Based Computations

When working with positions, we often concern ourselves with the distance between pairs of positions. There are two common ways to define this distance. We can use a standard Euclidean distance, the length of the shortest line between the two points.

```
(define square (lambda (x) (* x x)))
(define euclidean-distance
  (lambda (pos1 pos2)
    (sqrt (+ (square (- (position.col pos1) (position.col pos2)))
             (square (- (position.row pos1) (position.row pos2)))))))
```

We can also define distance in terms of the sum of the horizontal and vertical distances of the two positions. Because this distance mimics the way a taxi might drive in a city laid out on a grid (well, as long as you're not playing *Crazy Taxi*), we call this distance the "taxicab distance".

```
(define taxicab-distance
  (lambda (pos1 pos2)
    (+ (abs (- (position.col pos1) (position.col pos2)))
        (abs (- (position.row pos1) (position.row pos2))))))
```

Now, let's use those distance procedures to create some interesting drawings.

a. Write a procedure, (`pattern-e pos`) that returns

- the color magenta (255/0/255) if *pos* has a Euclidean distance of less than 30 from both (20,20) and (50,50);
- the color red if *pos* has a Euclidean distance of less than 30 from (20,20);
- the color blue if *pos* has a Euclidean distance of less than 30 from (50,50); or
- the color white, if none of the previous conditions hold.

b. Create a new image by applying this procedure at every position in `canvas`.

c. Write a procedure, (`pattern-t pos`), similar to `pattern-e`, except that it uses taxicab distance rather than Euclidean distance.

d. Create a new image by applying this procedure at every position in `canvas`.

### Extra 2: Checkerboards

Note that making a checkerboard-like image is much like making a set of stripes. However, instead of doing a whole column at a time, one must look at the columns and the rows.

- For pixels that are in a primary row and a primary column, use the primary color.
- For pixels that are in a primary row and an alternate column, use the alternate color.
- For pixels that are in an alternate row and a primary column, use the alternate color.
- For pixels that are in an alternate row and an alternate column, use the primary color.

- a. Write a procedure, (`checkerboard-color pos`) that, given a position, returns either black (the primary) or red (the secondary) according to the rules above. In your answer, strive to be as concise as possible.
- b. Use your procedure, along with `region.compute-pixels!`, to draw a simple checkerboard.

### Extra 3: Shades of Grey, Revisited

As you may recall from the reading on Boolean values and predicate procedures, it is possible to use `and` and `or` to achieve conditional behavior. The subsequent reading on conditionals provides some additional information on the use of `and` and `or`.

Using those techniques, rewrite `rgb.4grey` so that it uses neither `if` nor `cond`. That is, you'll need to find ways to express the same outcome using `and` and `or`.

### Extra 4: Drawing Triangles

In the earlier exploration of drawing shapes, we found that it was relatively easy to draw a triangle in the corner of an image by figuring out the slope of a line and restricting the pixels to be above the line.

To draw a more complex triangle, it may be necessary to figure out the formulae of three lines, one for each side of the triangle. We then build the triangle by drawing the desired color only for the pixels that are on the appropriate side of each side.

- a. Write instructions to draw an isosceles triangle with a horizontal base of length 20 and a height 20 in the color green. Put the base of the triangle at row 120 and the point of the triangle in column 70.
- b. Write instructions to draw a right triangle whose base is 30, and whose height is 60 in a color of your choice. Put the base of the triangle along row 130 and the left edge

## Explorations

### Exploration 1: Two-Color Computed Images

Using the technique of “when this expression holds, draw in one color, otherwise draw in another color”, see what interesting two-color images you can create.

### Exploration 2: Multi-color Checkerboards

If you completed extra problem 2, you've learned a technique for drawing two-color checkerboards. Extend that technique to draw checkerboards in more colors.

## Notes

---

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.