Exercise 1: Themes and variations

Out: Friday, October 9
Due: Friday, October 16 **noon**

Note: We strongly recommend that you start this assignment early, or at least read through the whole thing soon. You don't want to find out the night before it's due that you don't understand one of the questions.

Introduction

In this exercise we will experiment with creating serial images using iterators such as **iterated-overlay**, **iterated-beside**, etc. These are not built-in procedures, but we have provided definitions for both in the companion Racket file for this assignment. You should start by opening the Exercise 1.rkt file, which will start Racket, and load the associated file that defines the iterators you will be using.

Testing

We have provided a set of **unit tests**, bits of code and the correct answers they should generate, at the end of the file. So when you run your code, it will run the tests and generate a report of what tests succeeded and failed. The report will pop up as a window on your screen.

Questions and getting help

Remember that we have **office hours** all week (20 hours of them!), so if you have any issues, it's easy to get help.

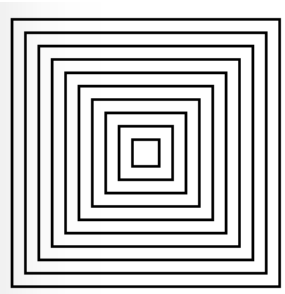
You can also post questions on Piazza (see the link in the class syllabus). However, when posting questions, it's very important that you **never post code to Piazza**. That is, if you are asking for help, don't just put your code in a message and post it as a public message. Send it as a private message to the instructors, or leave your code out of the posting.

Part 1: Writing expressions

Write expressions to do each of the following. Note that you need to exactly duplicate the pictures.

Remember that you're just writing expressions to make particular pictures, not procedures to make some general class of pictures. That will come in part 2.

Question 1



Use iterated-overlay to write an expression to make a set of 10 concentric squares: 10x10, 20x20, 30x30, etc., up to 100x100. Your picture should look like the one above. **Hint**: we recommend you count the number of squares in your picture, since it's easy to accidentally make a 0x0 square.

If you like, you can type your expressions at the **prompt** (the > at the bottom of the screen) and try them out that way. Note that before you do that, you'll need to **run the file once** to make sure that the require commands at the top of it get run. Otherwise, you won't have access to the graphics procedures. When you do, don't be scare that it pops up a testing report (or that it says that all the tests failed). You haven't typed any code yet! Just close the test report window and start hacking.

If you don't get the answer you like when typing at the prompt, you can type **control-up** (i.e. hold control down and press the upward arrow key to paste your last command back into the prompt so you can edit it. Change it as you like and try it again. Repeat until you get what you're looking for.

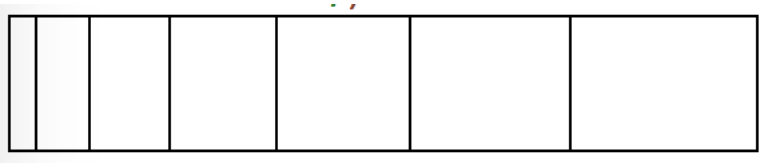
When you have an answer you're satisfied with, paste it into the expression we've provided inside the file:

```
(define question1-answer "fill this in")
```

Just replace the string "fill this in" with your answer. This will allow the testing code to find your answer and check it.

Alternatively, **instead of typing it into the prompt**, you can **type it directly into the define** from the beginning and **run the whole file to test it out**. That will generate a testing report and you can check it to see whether the test passed for this question.

Question 2

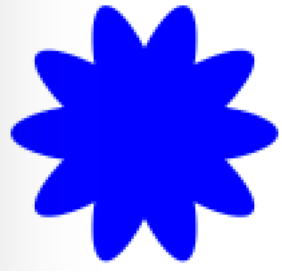


Now use iterated-beside write an expression to make a series of growing squares of sizes 10x50, 20x50, 30x50, etc., up to 100x70.

Again, fill your answer into the expression:

(define question2-answer **"fill this in"**)

Question 3

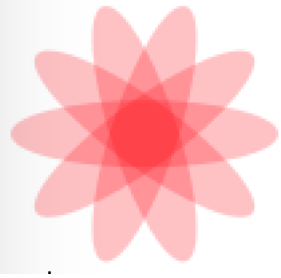


Now use iterated-overlay to make a flower pattern out of 100x25 blue ellipses like the one above. Use the color "blue". Your image should look just like the one above.

Again, fill your answer into the expression:

(define question3-answer **"fill this in"**)

Question 4



One cool thing about computer imagery is that you can specify a level of transparency (or alternatively opacity) for a color called an *alpha value*. When you draw two colors with alpha values on top of one another, the system does mixes the colors using a technique called *alpha bending*. You can specify an alpha value for a color by passing a 4th argument, the alpha value, to the color procedure. Alpha values range from 0, meaning the color is completely transparent and so drawing in it has no effect, to 255 meaning the color is completely opaque and it entirely covers up any ink beneath it. Modify your code from question 3 to use the color:

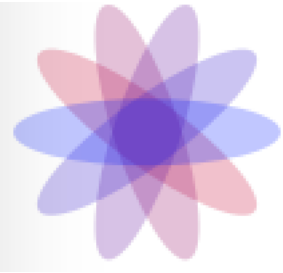
(color 255 0 0 64)

i.e. a pure but very transparent red. You should get the picture above.

Again, fill your answer into the expression:

(define question4-answer **"fill this in"**)

Question 5



Now do a version of where each ellipse is drawn in a different color, but they all have an alpha value of 64. Draw the first ellipse in (color 0 0 255 64), i.e. transparent blue. The second ellipse should have 50 fewer units of blue, but 50 more units of red, shifting it toward purple. So it should be (color 50 0 502 64). The next ellipse should have another 50 fewer units of blue and another 50 more units of red, e.g. (color 100 0 155 64). Continue that for all five ellipses. You should get the picture above.

Again, fill your answer into the expression:

(define question5-answer **"fill this in"**)

Part 2

Now write procedures that do similar things. Be sure to test your procedures to make sure that they work.

Note that in order to allow the unit tests to run, we added placeholders for your procedures to the file, e.g. for the swatch procedure you write for question 6, you will find the code:

```
(define swatch
  (λ (color1 color2 count)
    "fill this in"))
```

As before, just fill in the necessary part of the code.

Question 6: Paint swatches

Write a procedure, **swatch**, that takes two colors and a number as arguments, and makes a series of 50x50 squares where the leftmost square is the first color, the rightmost square is the

second color, and the intermediate squares interpolate between the two colors. The number of squares produced should be determined by the third argument (the number).

For example, the call should produce the image above:

```
(swatch (color 255 0 0)
        (color 0 0 255)
        5)
```

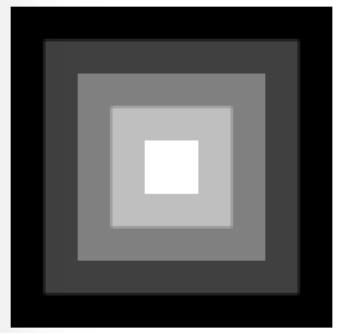
Another good test is:

```
(swatch (color 255 255 255)
        (color 0 0 0)
        2)
```

which should produce a perfectly white square next to a perfectly black one:

You can interpolate between two colors using the **interpolate-colors** procedure provides in the companion file for this assignment. If you say (interpolate-colors *color1 color2 weight*), where *weight* is a number between 0 and 1, it will return a new color that is *weight* parts *color1* and 1-*weight* parts *color2*. So as you vary weight from 0 to 1, the result will smoothly vary from *color1* (*weight*=0) to *color2* (*weight*=1).

Question 7: Controlling the range of iterated-overlay



Iterated-overlay is a little inconvenient because if you ask for 3 pictures, it always calls your generator procedure with the numbers 0, 1, and 2. But what if you wanted it to call the procedure with the numbers 0, 120, and 240, e.g. because you're using them as angles for rotations, and you want each number to be 1/3 of a revolution (120 degrees) more than the previous one?

Write two procedures, **iterated-overlay-inclusive**, and **iterated-overlay-exclusive**, that take four arguments; the first two are the picture generating procedure and count of pictures, like with the original version of iterated-overlay, and the other two are the starting and ending values of argument for the generator. So if you say (iterated-overlay-inclusive *generator* 5 0 10) or (iterated-overlay-exclusive *generator* 5 0 10), it means you want to call generator 5 times with arguments from 0 to 10.

Iterated-overlay-**inclusive** should include the end value among the values it passes to the generator. So if you say (iterated-overlay-inclusive *generator* 5 0 10), it should call generator with the arguments 0, 2.5, 5, 7.5, and 10. More generally, if the start value is s, the end value e, and the count is c, then on the ith call (i counting from 0) to the *generator*, it should pass the *generator* the argument:

$$s + i\frac{e-s}{c-1}$$

However, iterated-overlay-**exclusive** should omit the end value from the calls. If you say (iterated-overlay-exclusive *generator* 5 0 10), it should call generator with the arguments 0, 2, 4, 6, and 8. More generally, the ith time it calls the generator, it should pass it the argument:

$$s + i\frac{e-s}{c}$$

You can write both procedures by defining them in terms of the normal version of iterated-overlay. That is, your code should look something like:

```
(define iterated-overlay-inclusive
   (λ (generator count start end)
     (iterated-overlay (λ (picture-number)
                          …fill this in…)
                       count)))
```

For example this code:

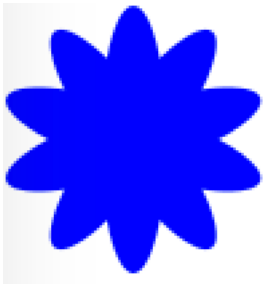
```
(iterated-overlay-inclusive (λ (weight)
                               (square (+ 20
                                          (* weight 100))
                                       "solid"
                                       (interpolate-colors (color 255 255 255)
                                                           (color 0 0 0)
                                                           weight)))
                            5
                            0
                            1)
```

Should produce the image above. The last three arguments of the code say it should call the generator 5 times with arguments between 0 and 1, inclusive. So that means it gets called with *weight* being 0, 0.25, 0.5, 0.75, and 1.

On the other hand, the code:

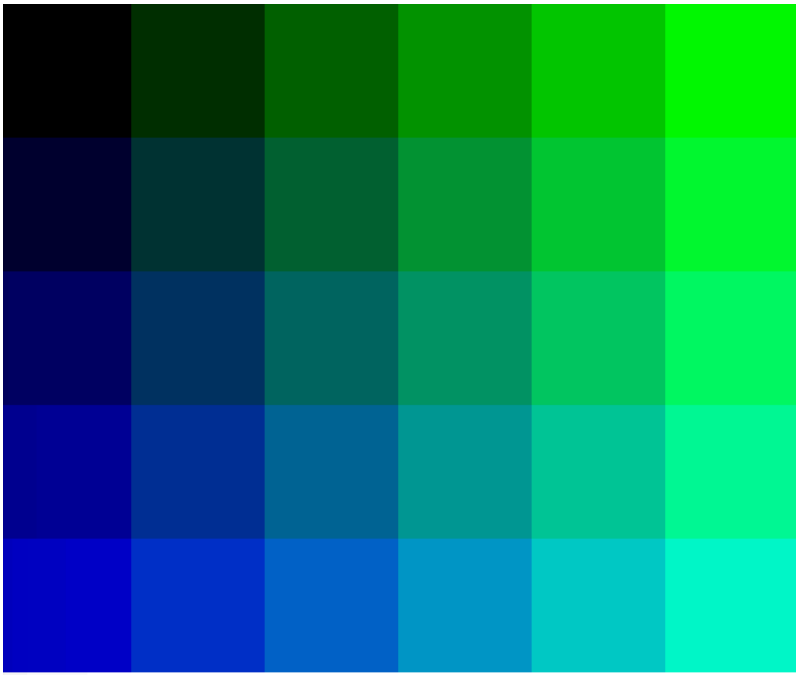
```
(iterated-overlay-exclusive (λ (angle)
                              (rotate angle
                                      (ellipse 25 100 "solid" "blue")))
                            5
                            0
                            180)
```

should produce the figure:



Here, it's calling the generator 5 times with arguments from 0 to 180 *exclusive*. Since these are angles, and since rotating an ellipse 180 degrees is the same as rotating it 0 degrees, we don't actually want the argument to the generator (the angle to rotate by) to ever get to 180. So we want it to stop short of 180 degrees. So it calls the generator with the arguments, 0, 36, 72, 108, and 144, and that gives the figure above.

Question 8: Grids revisited



We wrote a procedure in class that made grids, but it only made grids of identical pictures. What if we want to vary the objects that make up the grid? Write a procedure, **grid-from-generator**, that takes a generator (a picture making procedure), a number of columns, and a number of rows, and produces a grid of pictures where each subpicture in the grid is computed by calling the generator with two arguments: the number of the column the picture appears in, and the number of the row it appears in. Both rows and columns are numbered from zero.

You can do this by nesting calls to iterated-beside inside calls to iterated-above (or vice-versa; either will work). So your code would look like this:

```
(define grid-from-generator
   (λ (generator columns rows)
      (iterated-above (λ (row-number)
                         (iterated-beside (λ (column-number)
                                             …fill this in…))
                                          columns))
                      rows)))
```

For example, if we run the code:

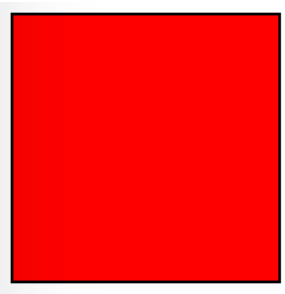
```
(grid-from-generator (λ (column row)
                       (square 50
                               "solid"
                               (color 0
                                      (* 50 column)
```

```
                                        (* 50 row))))
                                6
                                5)
```

should produce the figure above.

Don't worry about making your procedure work when the generator returns objects of different sizes. You may assume that the objects only vary in color.

Question 9



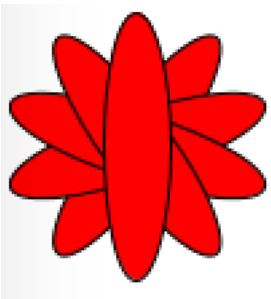
Write a procedure **outlined** that takes a picture-making procedure (either **rectangle** or **ellipse**), along with its width, height, and interior color, and returns the specified type, type, size, and color of shape, but outlined with a black line around it. For example, the call:

```
(outlined rectangle 100 100 "red")
```

should produce the picture above, while the call:

```
(iterated-overlay-exclusive (λ (angle)
                              (rotate angle
                                      (outlined ellipse 25 100 "red")))
                            5
                            0
                            180)
```

should produce the image:



Hint: your first argument really is a picture-making procedure (either rectangle or ellipse), so you can just call it. In fact, you can call it twice, once with “outline” as an argument and once with “solid” as the argument. Then just overlay them.

Turning it in

Before turning your assignment in, run the file one last time (e.g. type Command-R) to make sure that it runs properly and doesn’t generate any exceptions and all the tests pass. Assuming they do, press the **EECS-111 Handin** button to submit your assignment.

Note: if you run out of time before you get your assignment working fully, you can still submit the unfinished or broken version for partial credit.