HW 4: Making a Racket

due Tuesday, October 9 at 11:59pm

Overview

In this assignment, we'll play with the Racket programming language.The main goals of this work are to:

- Become familiar with DrRacket, the Integrated Development Environment (IDE) for Racket.
- Gently practice writing Racket code, in preparation for a larger Racket program next week.
- Make nice pictures.

Style and testing are important!

It is important to write code that is clear and that another person can easily understand. Be sure to write a "header comment" before each function, as well as comments for any part of the code that might not be immediately clear. You should also write tests (first!), where appropriate. You don't need to write tests for the first part of the assignment (it's hard to write tests to determine whether a picture is correct); but you must write tests for the second part of the assignment.

Note: We'll see examples of functions, header comments, and tests during Thursday's class.

Because we've just started learning Racket, we'll be lenient on the style grade this week: any reasonable attempt at clarity will receive credit. When grading, we'll also provide feedback on how to improve style, so that you get a sense of what "good style" means in Racket, for future assignments.

Materials

When you're ready to start, you'll need Racket, which you can download or use in the LAC or CS labs.

Racket tutorial

Open the Racket "Quick introduction" site, which is a fun guide through a small piece of the language that uses images.

- Do **steps 1–6**, running the code as you go.
- **Put all of your code in a file called `pictures.rkt`** (include the function definitions that the tutorial asks you to write in the definitions area).

- In the tutorial, click on the links, so you can practice reading the Racket documentation
    - For example, check out the function documentation for:
        - circle
        - hc-append

Drawing more pictures

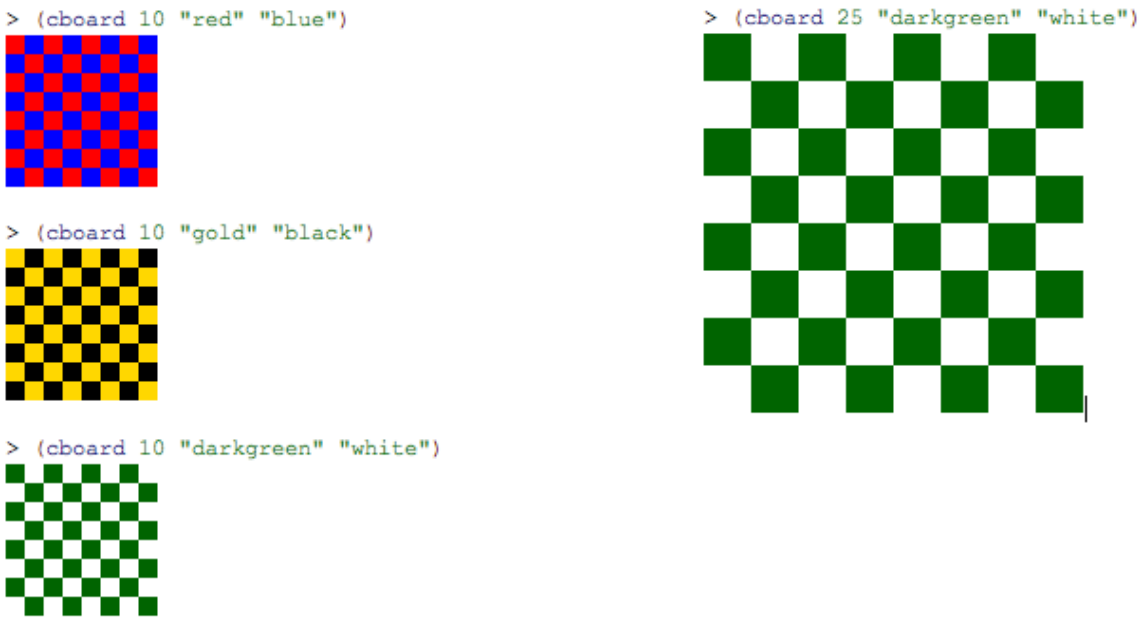
In this part of the assignment, you'll build on the tutorial to draw more pictures. Put your code in the same file, i.e., **`pictures.rkt`**.

cboard

Using the provided `checkerboard` function as a guide, create a function named `cboard` that takes three inputs: `n`, the number of pixels of the component square of the resulting checkerboard, `color1`, a string representing the color of the upper left square, and `color2`, a string representing the color of the lower left square. The output should be an 8x8 checkerboard, using only squares of the designated size, with the appropriate colors.

Be sure to have a header comment for your function! Remember that your grade will be based upon providing clear comments.

Here is a screenshot of four examples:



hcomb

Write another function, named `hcomb`, whose inputs are the same as for

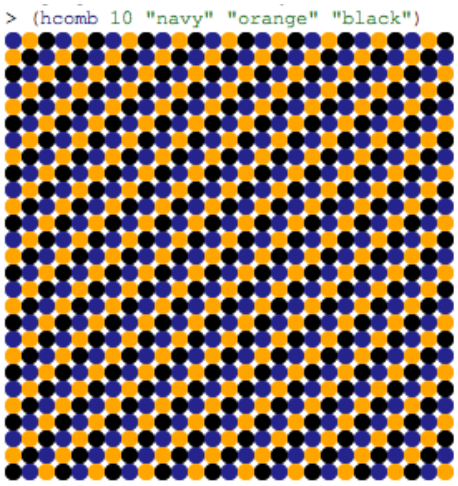
`cboard`, except that there is an additional `color`, called `color3`, that the function accepts as its fourth argument. The `hcomb` function should create a nine-by-nine grid of three-by-three patches of filled-in circles of the appropriate colors and the appropriate size. One strategy is to start with a copy of one of the existing functions and change things slowly to add the functionality you want (this is sometimes called iterative development).

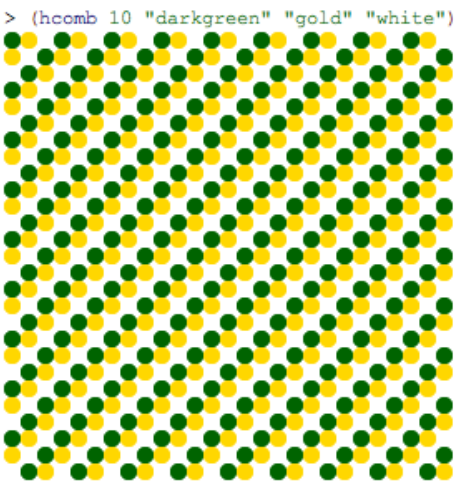
Again, add a header comment for your function. Your grade will be based in part upon your providing clear comments.

Note: to create filled-in circles, use `filled-ellipse`, e.g.,

```
(colorize (filled-ellipse 10 10) “blue”)
```

Here are two examples:





Submit your artwork, too.

When you finish, please take a PNG screenshot (instructions: www.take-a-screenshot.org/) that captures in the same shot at least one call to `cboard` and one call to `hcomb`. **Submit your screenshot as a file named `pictures.png`.**

What to turn in

Submit both of your files—`pictures.rkt` and `pictures.png`—to Gradescope assignment called “HW4: pictures.rkt and pictures.png”. There is no autograder for these files; we’ll manually grade them.

Collatz

In this part of the assignment, we'll practice writing recursive functions in Racket. **Place your tests, comments, and code in the file `collatz.rkt`.**

collatz

Write a Racket function named `collatz`, which takes one positive integer argument. The Racket function should compute the following mathematical function:

$$\text{collatz}(N) = \begin{cases} 3N+1 & \text{if N is odd} \\ N/2 & \text{if N is even} \end{cases}$$

Warning: Racket's built-in integer-division function is `quotient`, not the `/` symbol. If you try `(/ 3 2)` you will get the rational number three-halves, instead of the floating-point number 1.5 or the integer 1. But `(quotient 3 2)` evaluates to 1, as expected with integer division. The "mod" function in Racket is `modulo`, so that `(modulo 12 7)` evaluates to 5.

---

collatz-count

Write a **recursive** Racket function named `collatz-count`, which takes one positive integer argument. The function should return the smallest number of times that `collatz` (your previous function) must be called, when given an input of N, to arrive at a value of 1.

For example, consider the expression `(collatz 3)`. Let's use the symbol `==>` to mean "evaluates to." Then

```
(collatz (collatz (collatz (collatz (collatz (collatz (collatz 3)))))) ==> 1
```

so, `(collatz-count 3) ==> 7`, because there are seven applications of collatz above.

Here are a few Racket tests, which you can use in your code:

```
; provided tests
(check-equal? (collatz-count 26) 10)
(check-equal? (collatz-count 27) 111)
```

These are relatively complicated test cases. **Before** writing code for `collatz-count`, you should write at least two simple test cases, which will be helpful for debugging. Include these tests in your file.

Important information for testing

Be sure to add the line

```
(require rackunit)
```

at the top of your `collatz.rkt` file. This enables the `check-equal?` tests to run. This step is also necessary for your file to be accepted by the autograder.

Also, please make sure that you include the following two lines at the top of your .rkt file, right below the line`(require rackunit)`that you added earlier:

```
(provide collatz)
(provide collatz-count)
```

This allows the autograder to have access to your functions when it is running the tests. If you don't add these lines, the autograder won't run.

What to turn in

Submit your file `collatz.rkt` to Gradescope assignment called "HW4: collatz.rkt". Be sure your file contains the `require` and `provide` lines described above. There is an autograder for these files that will some run tests and show the results. We will manually grade the code's style and tests.

Where to go from here

If you're interested in these topics, here are a few ways to explore them deeper. These items aren't part of the assignment and aren't meant to interfere with your other work. They're just here for fun, and you can refer back to them later, whenever you have some free time and want to think more about the themes of the assignment.

If you're interested in reading more about functional programming, you may enjoy these articles:

- Excellent quora post by Professor Jean Yang (with a shoutout to Racket)
- *Why Functional Programming Matters* by John Hughes (the article from class)