Lesson #3: Recursive Functions in Racket

What's It All About?

This lesson presents a number of recursive function definitions in Racket.

1. The first collection of functions simply displays patterns of stars.
2. The second collection pertains to flipping coins.
3. The third collection plays a bit with a couple of classic number sequences.
4. The fourth collection displays patterns of squares.
5. The fifth collection channels Frank Stella.

Rows, Rectangles, and Squares of Stars

Three recursive functions are written. These are about as simple as you can get when it comes to recursive functions. Please study them to the point where you can write them quickly from scratch, duplicating mine from memory, but without direct reference to them. Practice as much as you like in DrRacket! Then, find a friend and talk about the functions with your friend. Perhaps you can explain what they do and how they do it to your friend, at various levels of conception. Perhaps they can do the same to you.

If you are brand new to recursion in programming, perhaps Google can help you to find your way to some sources of wisdom that resonate with you.

Interactions Pane

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( row-of-stars 5 )
* * * * *
> ( row-of-stars 28 )
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
> ( row-of-stars 0 )

> ( rectangle-of-stars 5 3 )
* * *
* * *
* * *
* * *
* * *
> ( rectangle-of-stars 4 28 )
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
> ( square-of-stars 2 )
* *
* *
> ( square-of-stars 11 )
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
>
```

Definitions Pane

```
#lang racket

( define ( row-of-stars n )
  ( cond
    ( ( = n 0 )
      ( display "\n" )
    )
    ( ( > n 0 )
      ( display "* " )
      ( row-of-stars ( - n 1 ) )
    )
  )
)

( define ( rectangle-of-stars r c )
  ( cond
    ( ( = r 0 )
      ( display "" )
    )
    ( ( > r 0 )
      ( row-of-stars c )
      ( rectangle-of-stars ( - r 1 ) c )
    )
  )
)

( define ( square-of-stars n )
  ( rectangle-of-stars n n )
)
```

Flipping Coins

The functions written in this section of the lesson all pertain to the flipping of coins. In keeping with the theme of this lesson, simple little recursions are featured.

Simulating the flip of a coin

All of the recursions in this section of the lesson will make use of the following coin flipping function:

Demo:

```
> ( flip-coin )
't
> ( flip-coin )
't
> ( flip-coin )
't
> ( flip-coin )
'h
> ( flip-coin )
't
> ( flip-coin )
't
> ( flip-coin )
'h
>
```

Code:

```
( define ( flip-coin )
  ( define outcome ( random 2 ) )
  ( cond
    ( ( = outcome 0 ) 't )
    ( ( = outcome 1 ) 'h )
  )
)
```

Flipping the coin some number of times

Demo:

```
> ( flip 0 )
> ( flip 5 )
t h t h t
> ( flip 5 )
t t t t h
> ( flip 5 )
h t h t t
> ( flip 30 )
h t h t t t h t t h t t t t h t t t h t t h h h h t t t t h t
> ( flip 30 )
h h t t h t t t h t t h h h t t h h h t h t h h h t t h h t
> ( flip 30 )
h h h t t h h h h t t t t t t h h t h h h h h t t t h t h t
>
```

Code:

```
( define ( flip n )
  ( cond
    ( ( not ( zero? n ) )
      ( display ( flip-coin ) ) ( display " " )
      ( flip ( - n 1 ) )
    )
  )
)
```

Flip for a head

Demo:

```
> ( flip-for-h )
t t t h
> ( flip-for-h )
t h
> ( flip-for-h )
h
> ( flip-for-h )
t t t t h
> ( flip-for-h )
h
> ( flip-for-h )
t t t h
> ( flip-for-h )
t t h
> ( flip-for-h )
h
> ( flip-for-h )
h
> ( flip-for-h )
```

```
t h
>
```

Code:

```
( define ( flip-for-h )
  ( define outcome ( flip-coin ) )
  ( display outcome ) ( display " " )
  ( cond
    ( ( not ( eq? outcome 'h ) )
      ( flip-for-h )
    )
  )
)
```

Flip for consecutive heads

Code:

```
( define ( flip-for-hh )
  ( flip-for-h )
  ( define outcome ( flip-coin ) )
  ( display outcome ) ( display " " )
  ( cond
    ( ( not ( eq? outcome 'h ) )
      ( flip-for-hh )
    )
  )
)
```

Demo:

```
> ( flip-for-hh )
t h h
> ( flip-for-hh )
t h t h t h t h h
> ( flip-for-hh )
h t h h
> ( flip-for-hh )
h t t h h
> ( flip-for-hh )
h t h t h t t h t h h
> ( flip-for-hh )
t t h h
> ( flip-for-hh )
t t h h
> ( flip-for-hh )
h h
```

```
> ( flip-for-hh )
t t h t h t t h t t h h
> ( flip-for-hh )
t h h
>
```

Classic Number Sequence Fun

Two classic number sequences are featured in this section of the lesson, the Fibonacci sequence and the Cottalz sequence.

The Fibonacci sequence

Two short programs will be written with respect to Fibonacci numbers. The first will simply compute the `nth` element of the sequence. The second will display the first `n` elements of the sequence.

Introduction/Enrichment

A quick look at the Wikipedia page will provide all the introduction to the Fibonacci sequence that you will need for this lesson. You might take a longer look at your convenience for a bit of enrichment.

`https://en.wikipedia.org/wiki/Fibonacci_number`

Demo:

```
> ( fibonacci-number 1 )
1
> ( fibonacci-number 2 )
1
> ( fibonacci-number 3 )
2
> ( fibonacci-number 4 )
3
> ( fibonacci-number 5 )
5
> ( fibonacci-number 10 )
55
> ( fibonacci-sequence 5 )
1 1 2 3 5
> ( fibonacci-sequence 10 )
1 1 2 3 5 8 13 21 34 55
> ( fibonacci-sequence 20 )
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
>
```

Code:

```
( define ( fibonacci-number n )
  ( cond
    ( ( = n 1 ) 1 )
    ( ( = n 2 ) 1 )
    ( ( > n 2 )
      ( + ( fibonacci-number ( - n 1 ) ) ( fibonacci-number ( - n 2 ) ) )
    )
  )
)

( define ( fibonacci-sequence n )
  ( cond
    ( ( > n 0 )
      ( fibonacci-sequence ( - n 1 ) )
      ( display ( fibonacci-number n ) ) ( display " " )
    )
  )
)
```

The Collatz sequence

Just one short program will be written with respect to Collatz numbers. It will compute the Collatz sequence corresponding to a given starting value.

Introduction/Enrichment

A quick look at the Wikipedia page will provide all the introduction to the Collatz conjecture, and the corresponding Collatz sequence, that you will need for this lesson. You might take a longer look at your convenience for a bit of enrichment.

`https://en.wikipedia.org/wiki/Collatz_conjecture`

Demo:

```
> ( collatz-sequence 10 )
10 5 16 8 4 2 1
> ( collatz-sequence 100 )
100 50 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
> ( collatz-sequence 55 )
55 166 83 250 125 376 188 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206
103 310 155 466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502 251
754 377 1132 566 283 850 425 1276 638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288
3644 1822 911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433 1300 650 325
976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
>
```

Code:

```
( define ( collatz-sequence n )
  ( display n ) ( display " " )
  ( cond
    ( ( = n 1 )
      ( display "\n" )
    )
    ( ( even? n )
      ( collatz-sequence ( / n 2 ) )
    )
    ( ( odd? n )
      ( collatz-sequence ( + ( * 3 n ) 1 ) )
    )
  )
)
```

Grids of Tiles

The program featured in this section renders grids of tiles. Eight tile geneators are composed, only one of which is deterministic (that for the "standard tile".) A little utility program is written to display the type types. A program is written to render a row of tiles. Using the row rendering program, a program is written to render a rectangle of tiles. Finally, using the rectangle renderer, a program is written to render a square of tiles.

Preliminary notes:

1. All of the tiles are of just one square size. It is an easy matter to add flexibility to vary the size of the tiles that appear in rows, rectangles, and squares.
2. A preview of higher order functions is presented in in this section. This will take the form of passing a functional argument as a parameter to a function.
3. Several demos will be presented before the featured code. The idea is the whet your appetite for reading some code.
4. The code listing will be taken directly from a file, as is, in to provide you with an example of a complete program.

Demo 1: Tiles

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( types-of-tile )
standard-tile random-color-tile random-red-tile random-green-tile
random-red-tile random-yellow-tile random-cyan-tile random-magenta-tile
> ( standard-tile )
```

```
> ( random-color-tile )
```

```
> ( beside ( random-color-tile ) ( random-color-tile ) )
```

```
> ( beside ( random-red-tile ) ( random-red-tile ) )
```

```
> ( beside ( random-green-tile ) ( random-green-tile ) )
```

```
> ( beside ( random-blue-tile ) ( random-blue-tile ) )
```

```
> ( beside ( random-yellow-tile ) ( random-yellow-tile ) )
```

```
> ( beside ( random-magenta-tile ) ( random-magenta-tile ) )
```

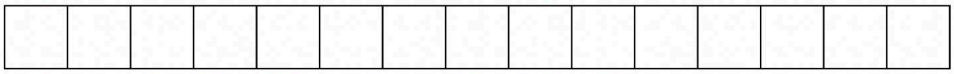
```
> ( beside ( random-cyan-tile ) ( random-cyan-tile ) )
```

```
>
```

Demo 2: Row of Tiles

Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.

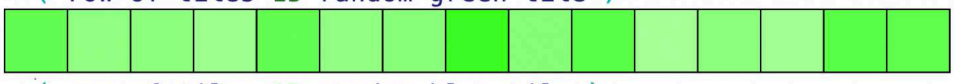
```
> ( types-of-tile )
standard-tile random-color-tile random-red-tile random-green-tile
random-red-tile random-yellow-tile random-cyan-tile random-magenta-tile
> ( row-of-tiles 15 standard-tile )
```



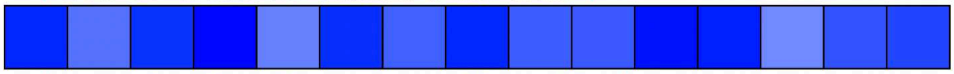
```
> ( row-of-tiles 15 random-color-tile )
```

```
> ( row-of-tiles 15 random-red-tile )
```

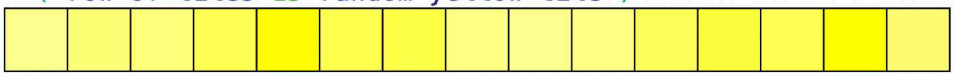
```
> ( row-of-tiles 15 random-green-tile )
```



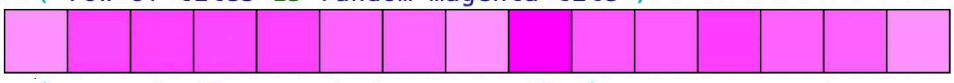
```
> ( row-of-tiles 15 random-blue-tile )
```



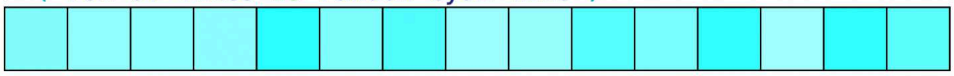
```
> ( row-of-tiles 15 random-yellow-tile )
```



```
> ( row-of-tiles 15 random-magenta-tile )
```



```
> ( row-of-tiles 15 random-cyan-tile )
```



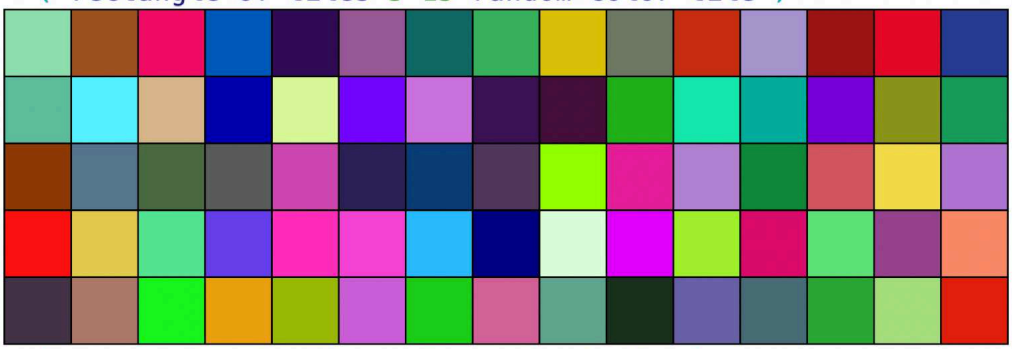
```
> ( row-of-tiles 1 standard-tile )
```

```
> ( row-of-tiles 0 random-color-tile )

>
```

Demo 3: Rectangle of Tiles

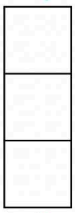
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.

```
> ( rectangle-of-tiles 5 15 random-color-tile )
```

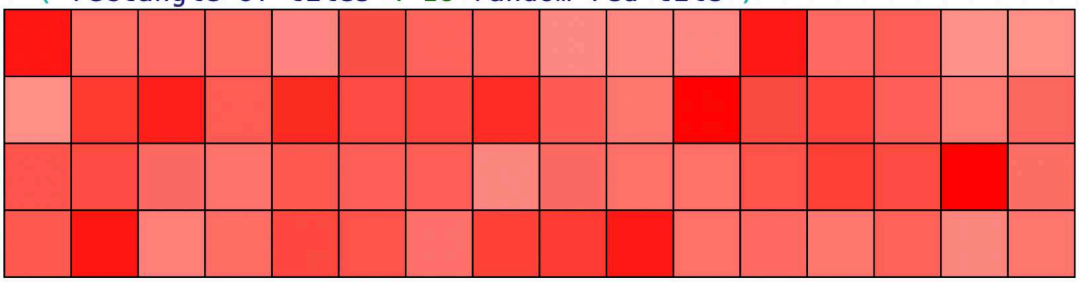


```
> ( rectangle-of-tiles 1 3 standard-tile )
```

```
> ( rectangle-of-tiles 3 1 standard-tile )
```



```
> ( rectangle-of-tiles 4 16 random-red-tile )
```

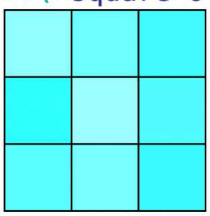


```
>
```

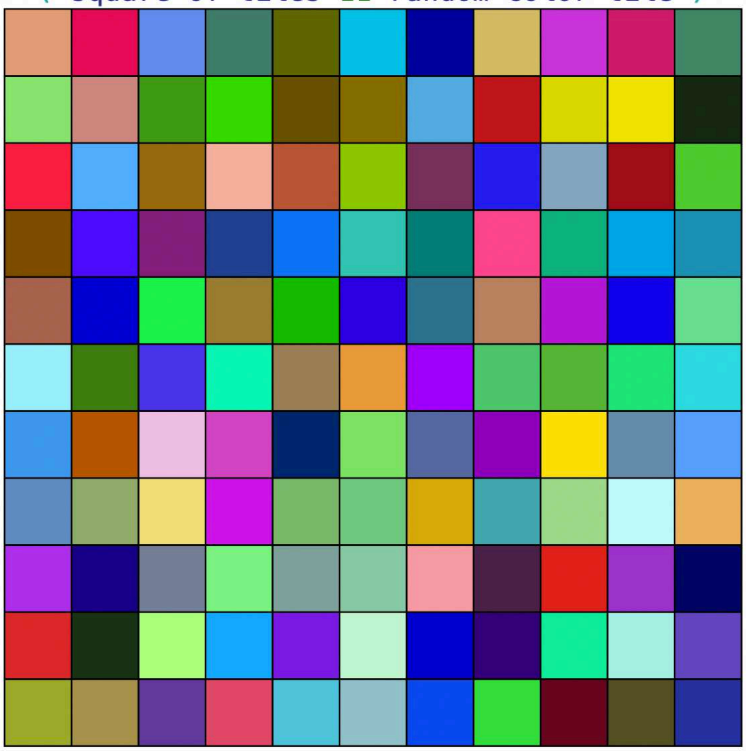
Demo 4: Square of Tiles

Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.

```
> ( square-of-tiles 3 random-cyan-tile )
```



```
> ( square-of-tiles 11 random-color-tile )
```



```
>
```

The Program: Grid.rtk

```
#lang racket

; -----------------------------------------------------------------------
; -----------------------------------------------------------------------
; Program that renders grids of tiles. A bit of randomness is tossed
; in, just for the fun of it.

; -----------------------------------------------------------------------
; Just one library to import, the image library from htdp, 2nd edition

(require 2htdp/image)

; -----------------------------------------------------------------------
; The display the available types of tiles

( define ( types-of-tile )
  ( display "standard-tile " )
  ( display "random-color-tile " )
  ( display "random-red-tile " )
  ( display "random-green-tile " )
  ( display "random-red-tile " )
  ( display "random-yellow-tile " )
  ( display "random-cyan-tile " )
  ( display "random-magenta-tile\n" )
)

; -----------------------------------------------------------------------
; Generate a row of tiles of specified length and tile type

( define ( row-of-tiles n tile )
  ( cond
    ( ( = n 0 )
      empty-image
    )
    ( ( > n 0 )
      ( beside ( row-of-tiles ( - n 1 ) tile ) ( tile ) )
    )
  )
)

; -----------------------------------------------------------------------
; Generate a rectangle of tiles of specified row count, column count,
; and tile type

( define ( rectangle-of-tiles r c tile )
  ( cond
    ( ( = r 0 )
      empty-image
    )
    ( ( > r 0 )
      ( above
```

```
        ( rectangle-of-tiles ( - r 1 ) c  tile ) ( row-of-tiles c tile ) )
    )
  )
)

; ---------------------------------------------------------------------
; Generate a square of tiles of specified side length and tile type

( define ( square-of-tiles n tile )
  ( rectangle-of-tiles n n tile )
)

; ---------------------------------------------------------------------
; Implment the tile generators

( define ( standard-tile ) ( square 40 "outline" "black" ) )

( define ( random-color-tile )
  ( overlay
    ( square 40 "outline" "black" )
    ( square 40 "solid" ( random-color ) )
  )
)

( define ( random-blue-tile )
  ( overlay
    ( square 40 "outline" "black" )
    ( square 40 "solid" ( random-blue-color ) )
  )
)

( define ( random-red-tile )
  ( overlay
    ( square 40 "outline" "black" )
    ( square 40 "solid" ( random-red-color ) )
  )
)

( define ( random-green-tile )
  ( overlay
    ( square 40 "outline" "black" )
    ( square 40 "solid" ( random-green-color ) )
  )
)

( define ( random-yellow-tile )
  ( overlay
    ( square 40 "outline" "black" )
    ( square 40 "solid" ( random-yellow-color ) )
  )
)

( define ( random-magenta-tile )
  ( overlay
```

```
    ( square 40 "outline" "black" )
    ( square 40 "solid" ( random-magenta-color ) )
  )
)

( define ( random-cyan-tile )
  ( overlay
    ( square 40 "outline" "black" )
    ( square 40 "solid" ( random-cyan-color ) )
  )
)

; Helper functions for rendering the tiles

( define ( random-color )
  ( color
    ( rgb-value ) ( rgb-value ) ( rgb-value )
  )
)

( define ( random-blue-color ) ( color 0 0 255 ( dark-rgb-value ) ) )

( define ( random-red-color ) ( color 255 0 0 ( dark-rgb-value ) ) )

( define ( random-green-color ) ( color 0 255 0 ( dark-rgb-value ) ) )

( define ( random-yellow-color ) ( color 255 255 0 ( dark-rgb-value ) ) )

( define ( random-magenta-color ) ( color 255 0 255 ( dark-rgb-value ) ) )

( define ( random-cyan-color ) ( color 0 255 255 ( dark-rgb-value ) ) )

( define ( rgb-value ) ( random 256 ) )

( define ( dark-rgb-value ) ( + ( random 128 ) 128 ) )
```

2htdp Image Documentation

There is a great deal of accessible functionality in the `2htdp/image` library. Some of it will come into play in your second Racket programming assignment. And some will be featured in the final section of this lesson. Consequently, this seems like a good time to get further acquainted with the functionality in a couple of libraries associated with the "How To Design Programs" project. Take a browse!

The Image Guide

`https://docs.racket-lang.org/teachpack/2htdpimage-guide.html`

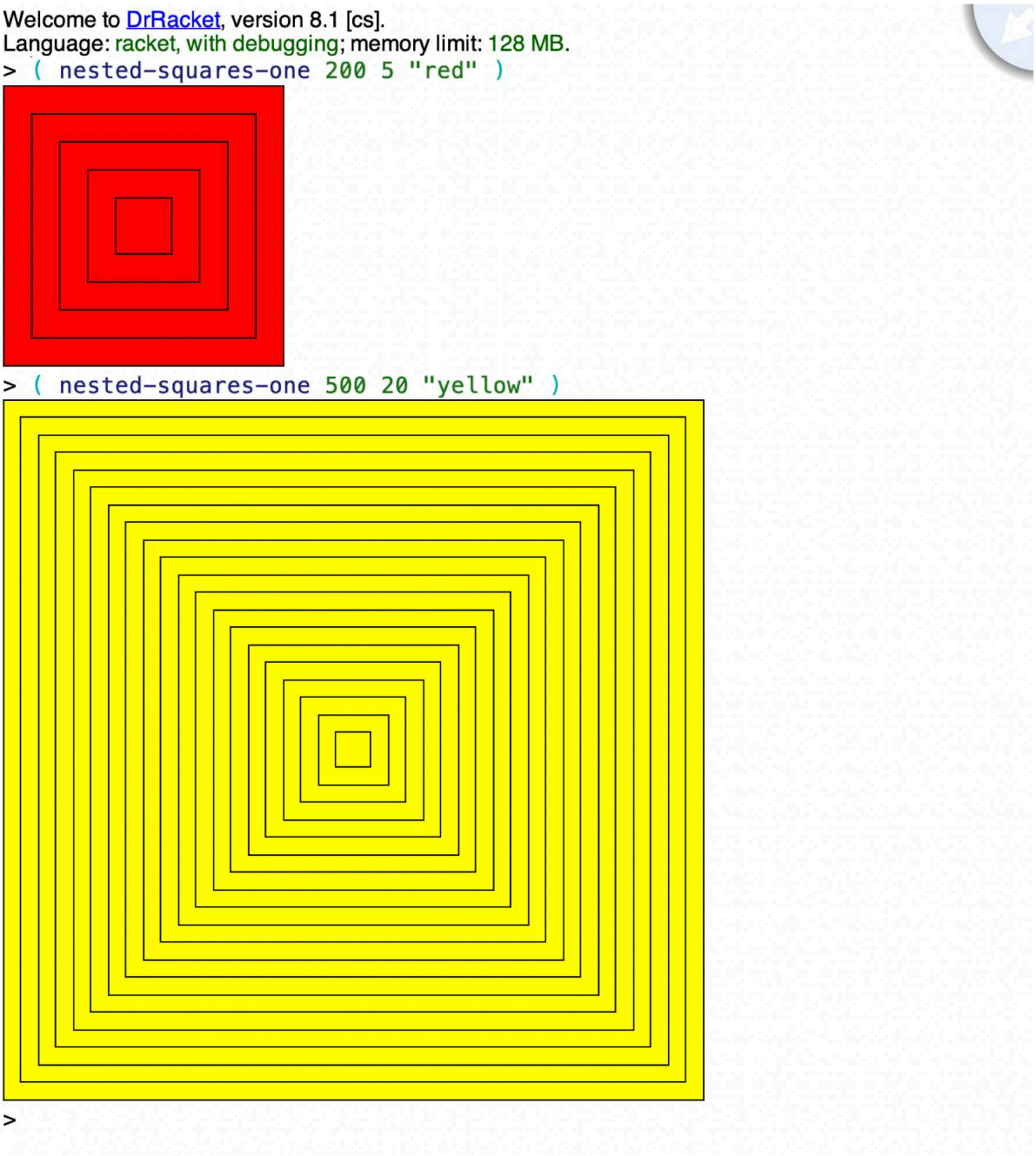
Images

`https://docs.racket-lang.org/teachpack/2htdpimage.html`

Channeling Frank Stella

You might like to Google Frank Stella, and take a look at pictures of some of his art. The two sorts of concentric squares featured in this portion of the lesson are representative of his style.

Demo – Monochromatic Stella Squares

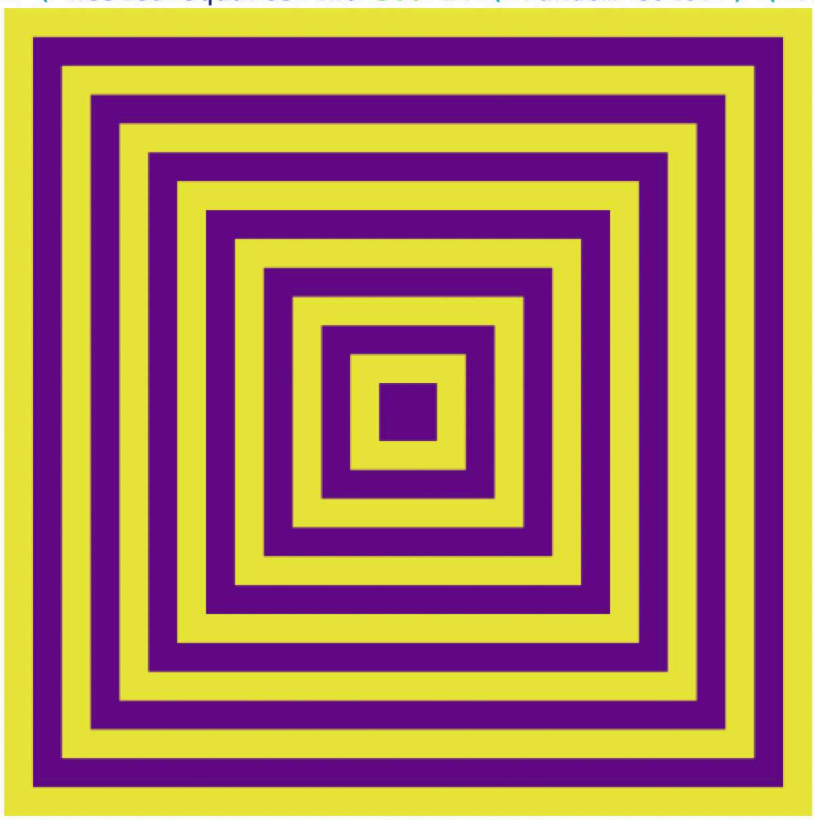


Code – Monochromatic Stella Squares

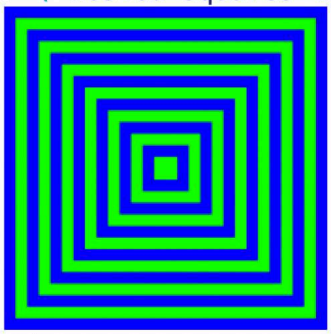
```
#lang racket

( require 2htdp/image )

( define ( nested-squares-one side count color )
  ( define unit ( / side count ) )
  ( paint-nested-squares-one 1 count unit color )
)

( define ( paint-nested-squares-one from to unit color)
  ( define side-length ( * from unit ) )
  ( cond
    ( ( = from to )
      ( framed-square side-length color )
    )
    ( ( < from to )
      ( overlay
        ( framed-square side-length color )
        ( paint-nested-squares-one ( + from 1 ) to unit color )
      )
    )
  )
)

( define ( framed-square side-length color )
  ( overlay
    ( square side-length "outline" "black" )
    ( square side-length "solid" color )
  )
)
```

Demo – Two Tone Stella Squares

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( nested-squares-two 500 14 ( random-color ) ( random-color ) )
```



```
> ( nested-squares-two 200 14 "blue" "green" )
```



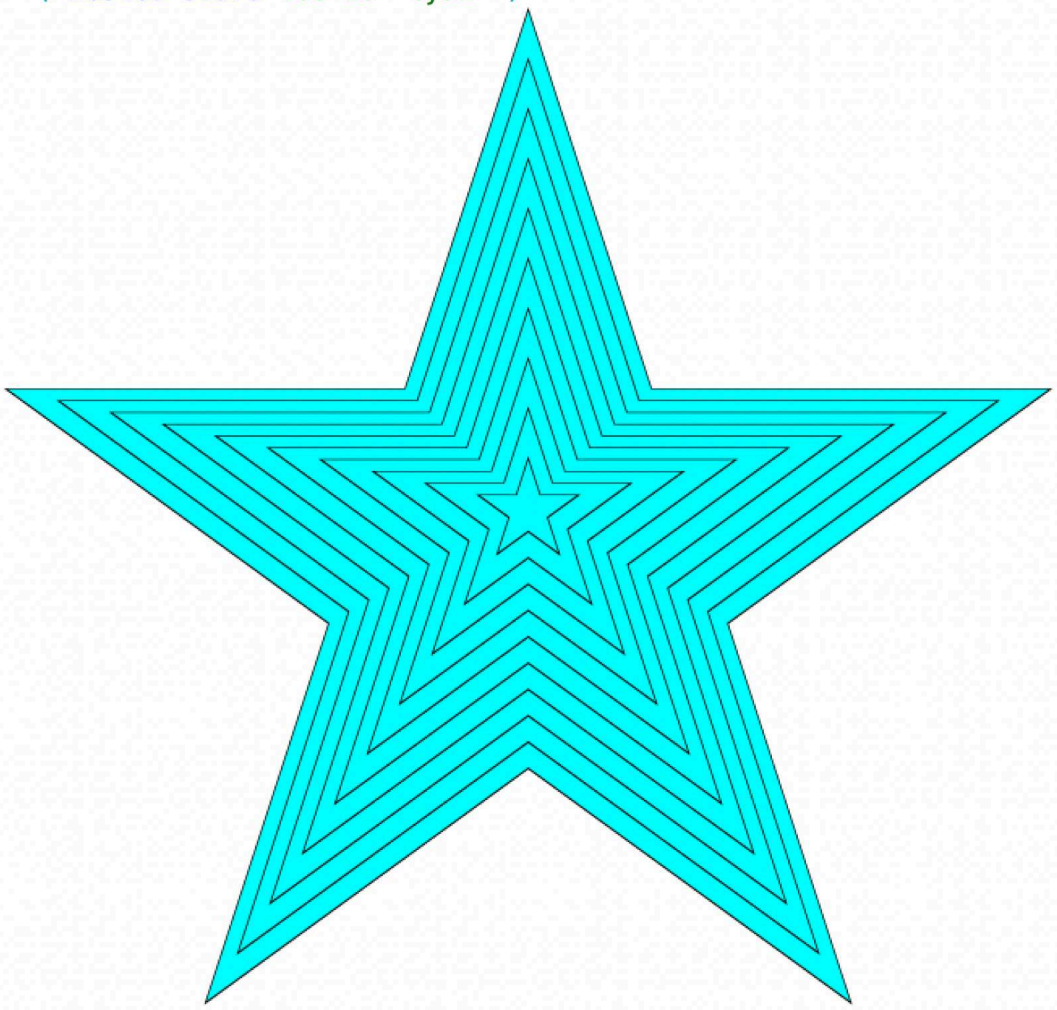
```
>
```

Code – Two Tone Stella Squares

```
#lang racket

( require 2htdp/image )

( define ( nested-squares-two side count color1 color2 )
  ( define delta ( / side count ) )
  ( paint-nested-squares-two 1 count delta color1 color2 )
)

( define ( paint-nested-squares-two from to delta color1 color2 )
  ( define side-length ( * from delta ) )
  ( cond
    ( ( = from to )
      ( if ( even? from )
        ( square side-length "solid" color1 )
        ( square side-length "solid" color2 )
      )
    )
    ( ( < from to )
      ( if ( even? from )
        ( overlay
          ( square side-length "solid" color1 )
          ( paint-nested-squares-two ( + from 1 ) to delta color1 color2 )
        )
        ( overlay
          ( square side-length "solid" color2 )
          ( paint-nested-squares-two ( + from 1 ) to delta color1 color2 )
        )
      )
    )
  )
)


( define ( random-color ) ( color ( rgb-value ) ( rgb-value ) ( rgb-value ) ) )
( define ( rgb-value ) ( random 256 ) )
```

Demo – Monochromatic Stella Stars

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( nested-stars 400 10 "cyan" )
```



```
> ( nested-stars 75 3 "yellow" )
```

```
> |
```

Code – Monochromatic Stella Stars

```
#lang racket

( require 2htdp/image )

( define ( nested-stars side count color )
  ( define unit ( / side count ) )
  ( paint-nested-stars 1 count unit color )
)

( define ( paint-nested-stars from to unit color)
  ( define side-length ( * from unit ) )
  ( cond
    ( ( = from to )
      ( framed-star side-length color )
    )
    ( ( < from to )
      ( overlay
        ( framed-star side-length color )
        ( paint-nested-stars ( + from 1 ) to unit color )
      )
    )
  )
)

( define ( framed-star side-length color )
  ( overlay
    ( star ( - side-length 3 ) "solid" color )
    ( star side-length "solid" "black" )
  )
)
```