Racket Programming Assignment #2: Racket Functions and Recursion

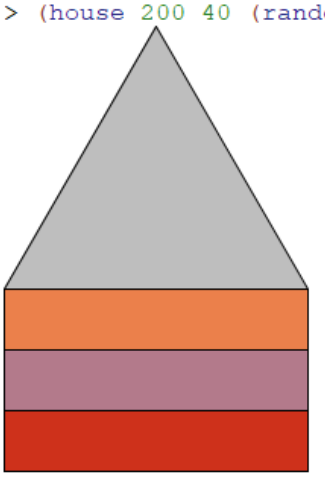
Learning Abstract

This Assignment will focus on Recursion and use of the 2htdp/image library in Racket. Recursion is a fast way of solving multi-level problems, without using iterative programming and saving a lot of time. It is letting the function do the work for you! The 2htdp/image library gives me access to several different pre-built functions to create the shapes you'll see below. To practice recursive functions and creating images, I will be completing 7 different tasks. These will include visual permutations in a set of 3, rolling dice for certain results using recursion, recursive number sequences, and recursive image generations. Once I have completed the first 6 tasks, my final task will be to use what I've learned to make my own creation. Overall, I will be familiarizing myself more with the Racket language and writing recursive functions for both images and non-image-outputting functions.

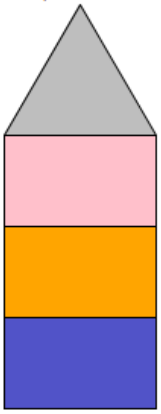
Task 1: Colorful Permutations of Tract Houses

1. **A snapshot of a demo for the house function that is just like that provided, except for the colors.**

```
> (house 200 40 (random-color) (random-color) (random-color))
```

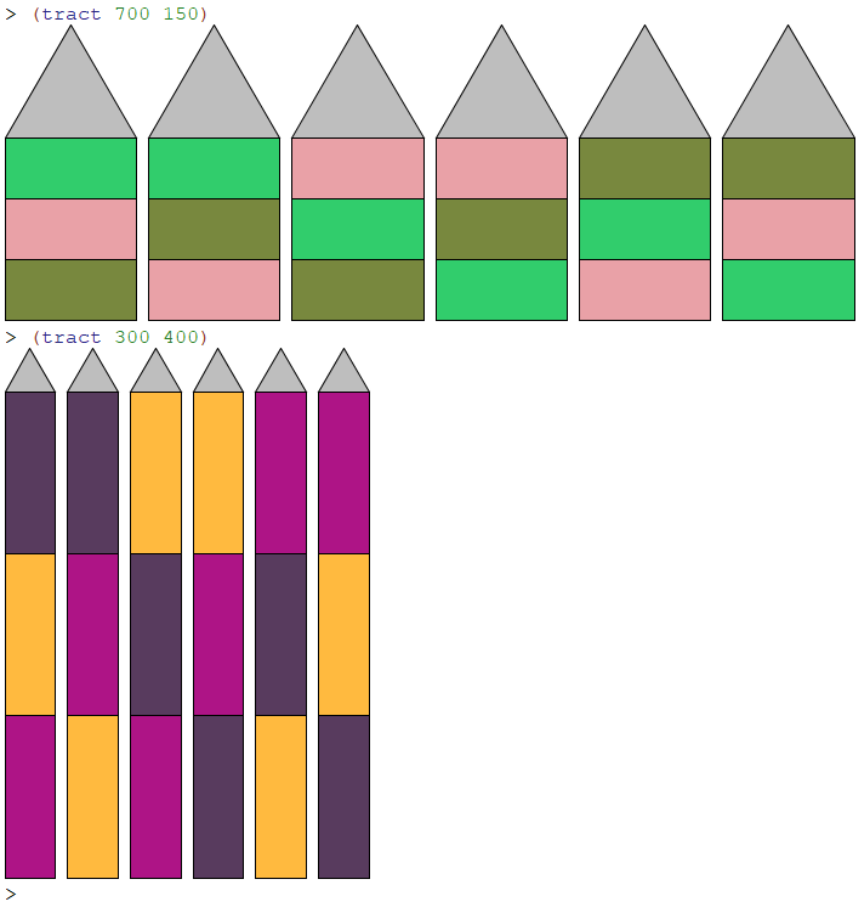


```
> (house 100 60 (random-color) "orange" "pink")
```



```
>
```

2. A snapshot of a demo for the tract function that is just like that provided, except for the colors.



3. The code for the house function, for the tract function, and for all of the code that you wrote in support of these two functions.

```
#lang racket
(require 2htdp/image)
(define (random-color) (color (rgb-value) (rgb-value) (rgb-value)))
(define (rgb-value) (random 256))
(define (house width height color1 color2 color3)
  (define rectangleoutline (rectangle width height 'outline "black"))
  (define triangleoutline (triangle width "outline" "black"))
  (define rectangle1 (rectangle width height "solid" color1))
  (define rectangle2 (rectangle width height "solid" color2))
  (define rectangle3 (rectangle width height "solid" color3))
  (define triangle1 (triangle width "solid" "gray"))
  (define the-house (above (underlay triangle1 triangleoutline) (underlay rectangle3 rectangleoutline)
  (underlay rectangle2 rectangleoutline) (underlay rectangle1 rectangleoutline)))
  the-house
)

(define (tract width height)
  (define hw (/ (- width 50) 6))
  (define hh (/ height 3))
  (define color1 (random-color))
  (define color2 (random-color))
  (define color3 (random-color))
  (define h1 (house hw hh color1 color2 color3))
  (define h2 (house hw hh color2 color1 color3))
  (define h3 (house hw hh color1 color3 color2))
  (define h4 (house hw hh color3 color1 color2))
  (define h5 (house hw hh color2 color3 color1))
  (define h6 (house hw hh color3 color2 color1))
  (define gap (square 10 "solid" "white"))
  (define the-tract(beside/align "top" h6 gap h5 gap h4 gap h3 gap h2 gap h1))
```

Task 2: Dice

1. **A demo in which roll-die is run 5 times, roll-for-1 is run 5 times, roll-for-11 is run 5 times, roll-for-odd-even-odd is run 5 times, and roll-two-dice-for-a-lucky-pair is run 10 times.**

```
> (roll-die)
2
> (roll-die)
3
> (roll-die)
3
> (roll-die)
4
> (roll-die)
5
> (roll-for-1)
2 4 4 2 6 6 1
> (roll-for-1)
1
> (roll-for-1)
5 3 4 1
> (roll-for-1)
6 6 2 3 3 4 1
> (roll-for-1)
3 5 1
> (roll-for-11)
1 2 6 2 3 4 1 5 4 2 6 4 1 2 1 4 5 3 1 1
> (roll-for-11)
6 6 2 1 3 3 6 5 3 3 5 6 3 3 6 4 1 5 2 1 6 3 2 3 1 4 1 3 1 2 6 1 4 2 1 3 5 6 1 4 4 6 5 2 1 4 ↵
5 4 2 4 4 1 5 3 6 3 4 4 3 5 3 4 6 6 6 1 5 4 4 5 6 5 3 6 6 2 3 6 4 2 5 4 3 4 6 2 3 3 5 1 2 5 ↵
4 2 2 4 5 4 4 6 6 6 3 5 5 3 3 4 6 2 5 3 3 5 5 3 6 2 5 6 5 6 6 1 6 6 5 2 2 6 6 3 6 4 6 6 6 3 ↵
5 2 5 5 3 1 5 6 4 6 1 2 2 3 1 3 3 3 4 6 6 3 6 3 4 2 4 5 6 1 1
> (roll-for-11)
5 5 5 2 2 6 2 1 4 3 1 4 2 1 5 3 5 1 4 5 5 4 3 5 1 5 2 6 3 5 1 5 4 1 3 2 1 4 2 6 3 5 6 3 3 5 ↵
2 2 6 6 1 3 2 6 4 5 1 3 5 6 1 6 4 1 6 6 4 2 5 2 2 3 4 6 4 2 2 1 4 5 3 3 4 6 5 6 3 5 4 6 5 1 ↵
1
> (roll-for-11)
5 6 3 3 3 5 3 5 5 6 2 1 5 5 5 2 4 4 5 6 6 1 4 5 6 5 2 2 4 4 4 4 6 1 1
> (roll-for-11)
1 2 5 1 6 4 1 3 3 2 2 2 1 1
> (roll-for-odd-even-odd)
2 5 5 2 1 5 4 5 2 1
> (roll-for-odd-even-odd)
6 2 6 4 1 1 6 4 5 4 4 3 5 2 5 5 2 5 2 1
> (roll-for-odd-even-odd)
5 5 3 5 5 3 1 2 1
> (roll-for-odd-even-odd)
6 2 6 6 1 3 2 2 3 1 4 1 5 2 6 2 6 4 5 4 6 1 4 5
> (roll-for-odd-even-odd)
3 5 2 5 5 1 1 3 3 5 4 1
```

```
> (roll-two-dice-for-a-lucky-pair)
(3 5)(6 1)
> (roll-two-dice-for-a-lucky-pair)
(4 6)(5 1)(1 5)(3 4)
> (roll-two-dice-for-a-lucky-pair)
(3 2)(3 2)(3 6)(2 1)(3 2)(2 1)(1 6)
> (roll-two-dice-for-a-lucky-pair)
(1 1)
> (roll-two-dice-for-a-lucky-pair)
(1 3)(3 4)
> (roll-two-dice-for-a-lucky-pair)
(3 4)
> (roll-two-dice-for-a-lucky-pair)
(3 1)(2 1)(2 4)(3 5)(4 5)(3 3)
> (roll-two-dice-for-a-lucky-pair)
(2 1)(1 1)
> (roll-two-dice-for-a-lucky-pair)
(1 3)(4 2)(4 5)(3 5)(4 5)(5 6)
> (roll-two-dice-for-a-lucky-pair)
```

2. Your code for the five featured functions, and for any other functions that you write in support of the featured functions.

```
(define (random-num) (+ (random 6) 1))
(define (roll-die)(random-num))
(define (roll-for-1)
  (define outcome (roll-die))
  (display outcome)(display " ")
  (cond
    ((not (eq? outcome '1))
     (roll-for-1)
     )
    )
  )
(define (roll-for-11)
  (roll-for-1)
  (define outcome (roll-die))
  (display outcome)(display " ")
  (cond
    ((not (eq? outcome '1))
     (roll-for-11)
     )
    )
  )
(define (roll-for-odd)
  (define outcome (roll-die))
  (display outcome)(display " ")
  (cond
    ((and(not (eq? outcome '1))(not (eq? outcome '3))(not (eq? outcome '5)))
     (roll-for-odd))
   )
)
(define (roll-for-even)
  (define outcome (roll-die))
  (display outcome)(display " ")
  (cond
    ((and(not (eq? outcome '2))(not (eq? outcome '4))(not (eq? outcome '6)))
     (roll-for-even))
   )
)
(define (roll-for-odd-even)
  (roll-for-odd)
  (define outcome (roll-die))
  (display outcome)(display " ")
  (cond
    ((and(not (eq? outcome '2))(not (eq? outcome '4))(not (eq? outcome '6)))
     (roll-for-odd-even))
   )
)

(define (roll-for-odd-even-odd)
  (roll-for-odd-even)
  (define outcome (roll-die))
  (display outcome)(display " ")
  (cond
        ((and(not (eq? outcome '1))(not (eq? outcome '3))(not (eq? outcome '5)))
    (roll-for-odd-even-odd))
   )
)
(define (roll-two-dice-for-a-lucky-pair)
  (define outcome1(roll-die))
  (define outcome2(roll-die))
  (display "(")(display outcome1)(display " ") (display outcome2) (display ")")
  (define sum (+ outcome1 outcome2))
  (cond
    ((and(not (eq? outcome1 outcome2))(not (eq? sum '7))(not (eq? sum '11)))
     (roll-two-dice-for-a-lucky-pair))
   )
)
```

Task 3: Number Sequences

1. **Three demos: Your recreation of the "preliminary demo". Your recreation of the "Triangular" demo. Your recreation of the "Sigma" demo.**

```
> (square 5)
25
> (square 10)
100
> (sequence square 15)
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225
> (cube 2)
8
> (cube 3)
27
> (sequence cube 15)
1 8 27 64 125 216 343 512 729 1000 1331 1728 2197 2744 3375
> (triangular 1)
1
> (triangular 2)
3
> (triangular 3)
6
> (triangular 4)
10
> (triangular 5)
15
> (sequence triangular 20)
1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210
> (sigma 1)
1
> (sigma 2)
3
> (sigma 3)
4
> (sigma 4)
7
> (sigma 5)
6
> (sequence sigma 20)
1 3 4 7 6 12 8 15 13 18 12 28 14 24 24 31 18 39 20 42
>
```

2. **A listing of the code that was involved in generating the three demos (my preliminary code, your code for triangular numbers and the sigma (sum of divisors) numbers**

```
(define (square n)
  (* n n)
  )

(define (cube n)
  (* n n n)
  )

( define ( sequence name n )
   ( cond
      ((= n 1)
       ( display ( name 1 ) ) ( display " " )
       )
   ( else
     ( sequence name ( - n 1 ) )
     ( display ( name n ) ) ( display " " )
     )
   )
)

(define (triangular n)
  (if (= n 1) n
      (+ n (triangular(- n 1)
           )
       )
   )
)

(define (factors n f)
  (cond ((> f n) 0)
        ((= 0 (remainder n f))
         (+ f (factors n (+ f 1))))
        (else
         (factors n (+ f 1)))))

(define (sigma n)
  (factors n 1))
```

Tas 4: Hirst Dots

1. **A demo that displays a 10x10 grid of Hirst dots, and a 4x4 grid of Hirst dots.**

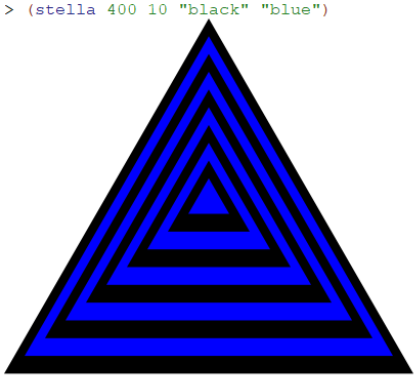
2. **Your code for the hirst-dots function, and for any functions that you write in support of it.**
   In order to get proper spacing, I used beside in the dot function to create a blank dot of radius 10. In order to get proper spacing between rows, I determined there was already a 4-pixel spacing created by the program, therefore I used above and placed a blank circle of 8 radius above each row to achieve the desired spacing of 20.

```
(define (dot n) (above
        (circle 8 "solid" "white") (beside (circle 10 "solid" "white")(circle n "solid" (random-color)))))

(define (row-of-dots n)
  (cond ((= n 0)
         (display "\n")
        )
  ((> n 0)
   (display (dot 15)) (row-of-dots (sub1 n)))))

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

(define (hirst-dots n)
  (rectangle-of-dots n n))
```

Task 5: Channeling Frank Stella

1. **A demo that displays at least two different images in the family of images that your program produces.**

```
> (stella 400 10 "black" "blue")
```



```
> (stella 250 5 "white" "red")
```

```
>
```

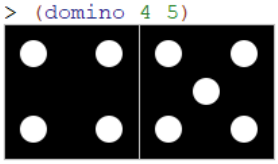
2. **Your code for your "Stella variation" function, and for any functions that you write in support of it.**

```
( define ( stella side count color1 color2 )
   ( define s ( / side count ) )
   ( paint 1 count delta color1 color2 )
)

( define ( paint from to delta color1 color2 )
   ( define side-length ( * from delta ) )
   ( cond
      ( ( = from to )
        ( if ( even? from )
             ( triangle side-length "solid" color1 )
             ( triangle side-length "solid" color2 )
            )
      )
      ( ( < from to )
```

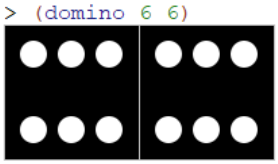
Task 6: Dominoes

1. **Your recreation of the final domino rendering demo.**

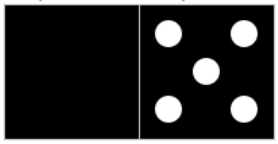
```
> (domino 4 5)
```



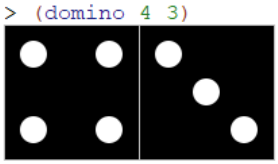
```
> (domino 6 6)
```



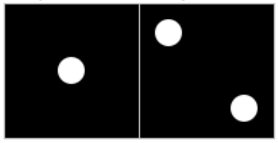
```
> (domino 0 5)
```



```
> (domino 4 3)
```



```
> (domino 1 2)
```



```
>
```

2. **The code for your complete domino rendering program (the code that you were given to work with plus that which you contributed.)**

```
#lang racket
;-------------------------------------------------------------------------------
; Requirements
;
; - Just the image library from Version 2 of "How to Design Programs"
;
( require 2htdp/image )
;-------------------------------------------------------------------------------
; Problem parameters
;
; - Variables to denote the side of a tile and the dimensions of a pip
;
( define side-of-tile 100 )
( define diameter-of-pip ( * side-of-tile 0.2 ) )
( define radius-of-pip ( / diameter-of-pip 2 ) )
;-------------------------------------------------------------------------------
; Numbers used for offsetting pips from the center of a tile
;
; - d and nd are used as offsets in the overlay/offset function applications
;
( define d ( * diameter-of-pip 1.4 ) )
( define nd ( * -1 d ) )
;-------------------------------------------------------------------------------
; The blank tile and the pip generator
;
; - Bind one variable to a blank tile and another to a pip
;
( define blank-tile ( square side-of-tile "solid" "black" ) )
( define ( pip ) ( circle radius-of-pip "solid" "white" ) )
;-------------------------------------------------------------------------------
; The basic tiles
;
; - Bind one variable to each of the basic tiles
;
( define basic-tile1 ( overlay ( pip ) blank-tile ) )
( define basic-tile2
( overlay/offset ( pip ) d d
( overlay/offset ( pip ) nd nd blank-tile)
)
)
( define basic-tile3 ( overlay ( pip ) basic-tile2 ) )
( define basic-tile4
   (overlay/offset (pip) nd d
   (overlay/offset (pip) d nd basic-tile2)))
( define basic-tile5 (overlay (pip) basic-tile4))
( define basic-tile6
   (overlay/offset (pip) 0 d
   (overlay/offset (pip) 0 nd basic-tile4 )))
;-------------------------------------------------------------------------------
; The framed framed tiles
;
; - Bind one variable to each of the six framed tiles
;
( define frame ( square side-of-tile "outline" "gray" ) )
( define tile0 ( overlay frame blank-tile ) )
( define tile1 ( overlay frame basic-tile1 ) )
( define tile2 ( overlay frame basic-tile2 ) )
( define tile3 ( overlay frame basic-tile3 ) )
( define tile4 ( overlay frame basic-tile4 ) )
( define tile5 ( overlay frame basic-tile5 ) )
( define tile6 ( overlay frame basic-tile6 ) )
;-------------------------------------------------------------------------------
; Domino generator
;
; - Funtion to generate a domino
;
( define ( domino a b )
( beside ( tile a ) ( tile b ) )
)
( define ( tile x )
   ( cond
( ( = x 0 ) tile0 )
( ( = x 1 ) tile1 )
( ( = x 2 ) tile2 )
( ( = x 3 ) tile3 )
( ( = x 4 ) tile4 )
( ( = x 5 ) tile5 )
( ( = x 6 ) tile6 )
)
)
```

Task 7: Creation

1. **An image of a demo that features you creation.**

```
> (my-creation 200 "yellow" "black")
```

```
> (my-creation 150 (random-color) (random-color))
```

```
>
```

2. **The code for your creation.**

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


(define (my-creation n color1 color2)
  (define lx (* n -.25))
  (define x (* n .25))
  (define y (* n .375))
  (define my (* n -.4))
  (define eye-width (/ n 6))
  (define eye-height(* n .7))
  (define main-circle (circle n "solid" color1))
  (define outline (circle n "outline" color2))
  (define eye (ellipse eye-width eye-height "solid" color2))
  (define mouth (scene+curve (rectangle (* n 1.5) (/ n 2) "solid" color1)
              (* .0666 (* n 1.5)) 0 (* -.2 (* n 1.5)) .4
              (* .9333 (* n 1.5)) 0 (* .2 (* n 1.5)) .4
              color2))
  (define smileyface(overlay/offset mouth 0 my
                  (overlay/offset eye x y
                  (overlay/offset eye lx y (underlay main-circle outline)))))
  smileyface)
```