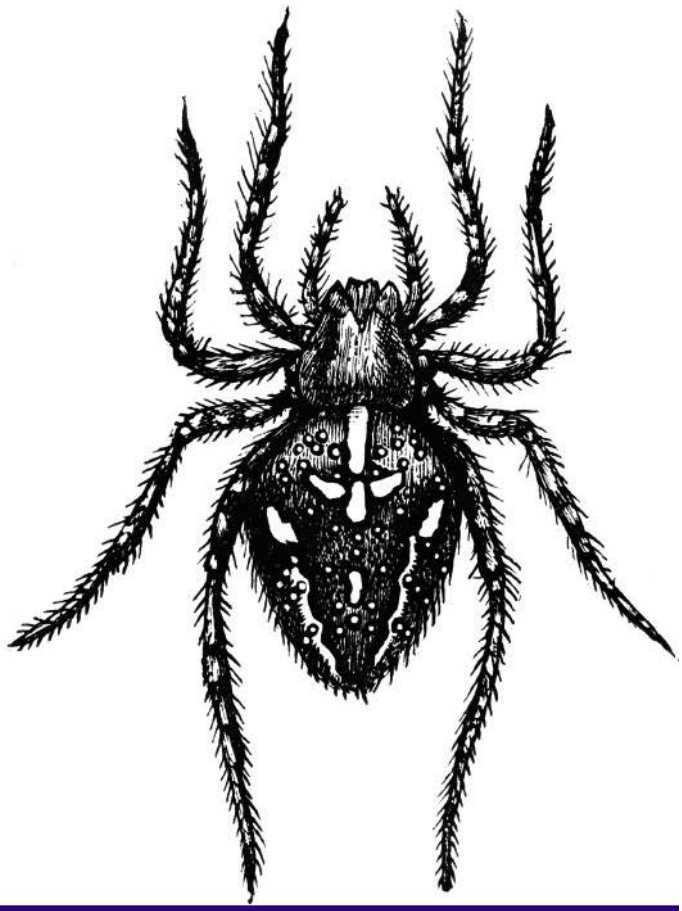
Racket Programming Assignment #4



Recursive List Processing

and Higher Order Functions

By Miguel Cruz

Abstract

This assignment features 10 tasks that goes over simple recursive list processing functions which some feature association lists, simple higher-order functions, and colorful renderings of diverse phenomena.

Problem 1 - Count

Specification

Define a **recursive** function called `count`, consistent with the given pseudocode, according to the following specification:

1. The first parameter is presumed to be a Lisp object.
2. The second parameter is presumed to be a list of Lisp objects .
3. The value of the function will be the number of occurrences of the object in the list.

Demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (count 'b '( a a b a b c a b c d))
3
> (count 5 '(1 5 2 5 3 5 4 5 ))
4
> (count 'cherry '(apple peach blueberry))
0
>
```

Source Code

```
1  #lang racket
2
3  ;---------
4  ; Programming Assignment 4: Recursive List Processing and Higher Order Functions
5
6  ;; Task 1: Count
7
8  (define (count obj l)
9    (cond
10     ( (empty? l) 0)
11     ( (eq? obj (car l) )
12       (+ 1 (count obj (cdr l)) ) )
13     (else
14      (count obj  (cdr l)) )
15     )
16   )
17
18
19
20
```

Problem 2 - list->set

Specification

Define a **recursive** function called `list->set`, consistent with the given pseudocode, according to the following specification:

1. The sole parameter is presumed to be a list.
2. The value of the function will be a list consisting of just one occurrence of each element in the given list.

Demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (list->set '(a b c b c d c d e))
'(a b c d e)
> (list->set '(1 2 3 2 3 4 3 4 5 4 5 6))
'(1 2 3 4 5 6)
> (list->set '(apple banana apple banana cherry))
'(apple banana cherry)
>
```

Source Code

```
1  #lang racket
2
3  ;---------
4  ; Programming Assignment 4: Recursive List Processing and Higher Order Functions
5
6  ;; Task 2: List-> Set
7
8  (define (list->set l)
9    (cond
10     ((empty? l) l)
11     ((member (car l) ( cdr l))
12      (list->set (cdr l))
13      )
14     (else
15      (cons (car l) (list->set (cdr l)))
16      )
17     )
18   )
19
```

Problem 3 - Association List Generator

An **association list**, or **a-list**, is a a list of cons cells. Often the cons cells contain an atomic car and an atomic cdr, but this is not necessarily the case.

Specification

Define a **recursive** function called `a-list` according to the following specification:

1. The first parameter is presumed to be a list of objects.
2. The second parameter is presumed to be a list of objects of the same length as the value of the first parameter.
3. The value of the function will be a list of pairs obtained by "cons-ing" successive elements of the two lists.

Demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (a-list '(one two three four five) '(un deux trois quatre
cinq))
'((one . un) (two . deux) (three . trois) (four . quatre) (five . cinq))
> (a-list '() '())
'()
> (a-list '(this) '(that))
'((this . that))
> (a-list '(one two three) '( (1) (2 2) (3 3 3)))
'((one 1) (two 2 2) (three 3 3 3))
>
```

Source Code

```racket
#lang racket

;---------
; Programming Assignment 4: Recursive List Processing and Higher Order Functions

;; Task 3: Association List Generator


(define (a-list list_a list_b)
  (cond
    ((empty? list_a) '())
    (else
     (cons (cons (car list_a) (car list_b)) (a-list (cdr list_a)
                                                    (cdr list_b)))
     )
    )
  )
```

Problem 4 - Assoc

Specification

Define a **recursive** function called `assoc` according to the following specification:

1. The first parameter is presumed to be a lisp object.
2. The second parameter is presumed to be an association list.
3. The value of the function will be the first pair in the given association list for which the car of the pair equals the value of the first parameter, or '() if there is no such element.

Demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (define al1
(a-list '(one two three four) '(un deux trois quatre))
)
> (define al2
(a-list '(one two three) '( (1) (2 2) (3 3 3)))
)
> al1
'((one . un) (two . deux) (three . trois) (four . quatre))
> (assoc 'two al1)
'(two . deux)
> (assoc 'five al1)
'()
> al2
'((one 1) (two 2 2) (three 3 3 3))
> (assoc 'three al2)
'(three 3 3 3)
> (assoc 'four al2)
'()
>
```

Source Code

```
1  #lang racket
2
3  ;---------
4  ; Programming Assignment 4: Recursive List Processing and Higher Order Functions
5
6  ;; Task 4: Assoc
7
8
9
10 (define (a-list list_a list_b)
11   (cond
12     ((empty? list_a) '())
13     (else
14      (cons (cons (car list_a) (car list_b)) (a-list (cdr list_a)
15                                                     (cdr list_b)))
```

```
      )
    )
  )

(define (assoc obj list)
  (cond
    ((empty? list) '())
    ((eq? (car (car list)) obj)
     (car list)
     )
    (else
     (assoc obj (cdr list))
     )
    )
  )
```

Problem 5 - Frequency Table

The Nature of this Task

This task is more of a reading/study/analysis task than a function composition task. Here is the script:

1. I will provide you with a demo for (1) a frequency table generating function, and (2) a very simple frequency table visualization program.
2. I will provide you with the code for these two programs, and the supporting code that they reference.
3. You will be asked to type in the code, and mimic the demo.
4. You will be asked to answer several questions about the code.
5. You will be asked to incorporate the code, the demo, and your answers to the questions into your solution document.

Demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (define ft1 (ft '(10 10 10 10 1 1 1 1 9 9 9 2 2 2 8 8 3 3 4 5
6 7)))
> ft1
'((1 . 4) (2 . 3) (3 . 2) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 2) (9 . 3) (10 . 4))
> (ft-visualizer ft1)
1:   ****
2:   ***
3:   **
4:   *
5:   *
6:   *
7:   *
8:   **
9:   ***
10:  ****
```

```
> (define ft2 (ft '( 1 10 2 9 3 8 4 4 7 7 6 6 6 5 5 5 )))
> ft2
'((1 . 1) (2 . 1) (3 . 1) (4 . 2) (5 . 3) (6 . 3) (7 . 2) (8 . 1) (9 . 1) (10 . 1))
> (ft-visualizer ft2)
1:   *
2:   *
3:   *
4:   **
5:   ***
6:   ***
7:   **
8:   *
9:   *
10:  *
>
```

```
( define ( ft the-list )
  ( define the-set ( list->set the-list ) )
  ( define the-counts
    ( map ( lambda (x) ( count x the-list ) ) the-set )
  )
  ( define association-list ( a-list the-set the-counts ) )
  ( sort association-list < #:key car )
)

( define ( ft-visualizer ft )
  ( map pair-visualizer ft )
  ( display "" )
)

( define ( pair-visualizer pair )
  ( define label
    ( string-append ( number->string ( car pair ) ) ":" )
  )
  ( define fixed-size-label ( add-blanks label ( - 5 ( string-length label ) ) ) )
  ( display fixed-size-label )
  ( display
    ( foldr

     string-append
     ""
     ( make-list ( cdr pair ) "*" )
   )
 )
  ( display "\n" )
)

( define ( add-blanks s n )
  ( cond
    ( ( = n 0 ) s )
    ( else ( add-blanks ( string-append s " " ) ( - n 1 ) ) )
  )
)
```

Questions (for the most part)

1. List the names of the functions used within the `ft` function that you were asked to write in this programming assignment.
    a. **Count, a-list, and list->set are used in the ft function as support functions to make it work.**
2. Within the `ft` function, what function is provided to the higher order function `map`? Since you cannot name this function, please write down the complete definition of this function.
    a. **The map functionally within the ft function is expressed through a lambda function. The function creates a list of the number of occurrences of each element within the list.**
3. How many parameters must the functional argument to the application of `map` in the `ft` function take?
    a. **The map function must contain two parameters.**
4. What would be the challenge involved in writing a named function to take the place of the lambda function within the `ft` function. Do your best to articulate this challenge in just one sentence.
    a. **The main challenged involved when it comes to writing a named function as opposed to a lambda function within the ft function is because higher order functions cannot have parameters of themselves, which is required in this case.**
5. Within the `ft` function, what function is provided to the higher order function `sort`? Since you can name this function, please simply write down its name.
    a. **car**
6. What is a "keyword argument"?
    a. **The definition of a "keyword argument" in the context of lisp is that it's a argument that is marked as a keyword as opposed to being recognized by its position in the argument line.**
7. Within the `ft-visualizer` function, what function is provided to the higher order function `map`? Since you can name this function, please simply write down its name.
    a. **pair-visualizer**
8. Why was the challenge involved in using a named function in the application of `map` in the `ft` function absent in the application of `map` in the `ft-visualization` function?
    a. **The reason why the challenge was absent in the application of map in the ft-visualization function was because map needs additional args which implies the use of a lambda function. Map and ft-visualization works only within the constraints of the number of parameters the map function provides.**
9. Within the `pair-visualizer` function, what function is provided to the higher order function `foldr`? Since you can name this function, please simply write down its name.
    a. **string-append**
10. Does the `add-blanks` function make use of any higher order functions?
    a. **The add-blanks function does make use of higher order functions such as string-append**
11. Why do you think the `display` function, with the empty string as its argument, was called in the `ft-visualizer` function?
    a. **A list of #<void> would occur if any empty strings were going through it, ft-visualizer would call upon pair-visualizer, but if there's no pairs to visualize then the void result is expected.**
12. What data structure is being used to represent a frequency table in this implementation? Please be as precise as you can be in articulating your answer, preferring abstraction to detail in your precision of expression.
    a. **The frequency table represents the association list, the car of the list represents the number in the list and the cdr representing frequency of the number within the list.**
13. Is the `make-list` function used in the `pair-visualizer` function a primitive function in Racket?
    a. **Yes**
14. What do you think is the most interesting aspect of the given frequency table generating code?
    a. **The application of using maps makes me relate it to other languages where maps are used, but the implementation of maps in lisp is pretty interesting.**
15. Please ask a meaningful question about some aspect of the accompanying code. Do your best to make it a question that you think a reasonable number of your classmates will find interesting.
    a. **Does this has any practical use or is it only for pedagogy purposes?**

Problem 6 - Generate list

Specification

Define a **recursive** function called `generate-list` according to the following specification:

1. The first parameter is a nonnegative integer.
2. The second parameter is a parameterless function that returns a lisp object.
3. The function returns a list of length equal to the value of the first parameter containing objects created by calls to the function represented by the second parameter.

Demo 1..3

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (generate-list 10 roll-die)
'(5 3 3 4 6 1 2 4 3 3)
> (generate-list 20 roll-die)
'(3 6 3 6 6 5 3 3 3 5 6 5 3 5 5 3 4 5 3 3)
> (generate-list 12 (lambda () (list-ref '(red yellow blue )
(random 3))))
'(red blue blue red blue yellow red red yellow blue blue red)
> (define dots (generate-list 3 dot))
> dots
```

```
(list                )
> (foldr overlay empty-image dots)
```

```
> (sort-dots dots)
```

```
(list                )
> (foldr overlay empty-image (sort-dots dots))
```

```
> (define a (generate-list 5 big-dot))
> (foldr overlay empty-image (sort-dots a))
```

```
> (define b (generate-list 10 big-dot))
> (foldr overlay empty-image (sort-dots b))
```

```
>
```

Source Code

```
1  #lang racket
2  (require 2htdp/image)
3  ;---------
4  ; Programming Assignment 4: Recursive List Processing and Higher Order Functions
5
6  ;; Task 6: Generate List
7
8  (define (generate-list num fn)
9    (cond
10     ((= 1 num) (make-list 1 (fn)))
11     (else
12      (append (make-list 1 (fn)) (generate-list (- num 1) fn))
13      )
14     )
15   )
16
17 (define (big-dot)
18   (circle (* 2 (random 41)) "solid" (random-color))
19   )
20
21 (define (dot)
22   (circle ( + 10 ( random 41 ) ) "solid" (random-color) )
23   )
24 (define (random-color)
25   (color (random 256)(random 256)(random 256))
26   )
27 (define (roll-die) (+ (random 6) 1))
28 (define (sort-dots loc)
29   (sort loc #:key image-width <)
30   )
```

Problem 7 - The Diamond

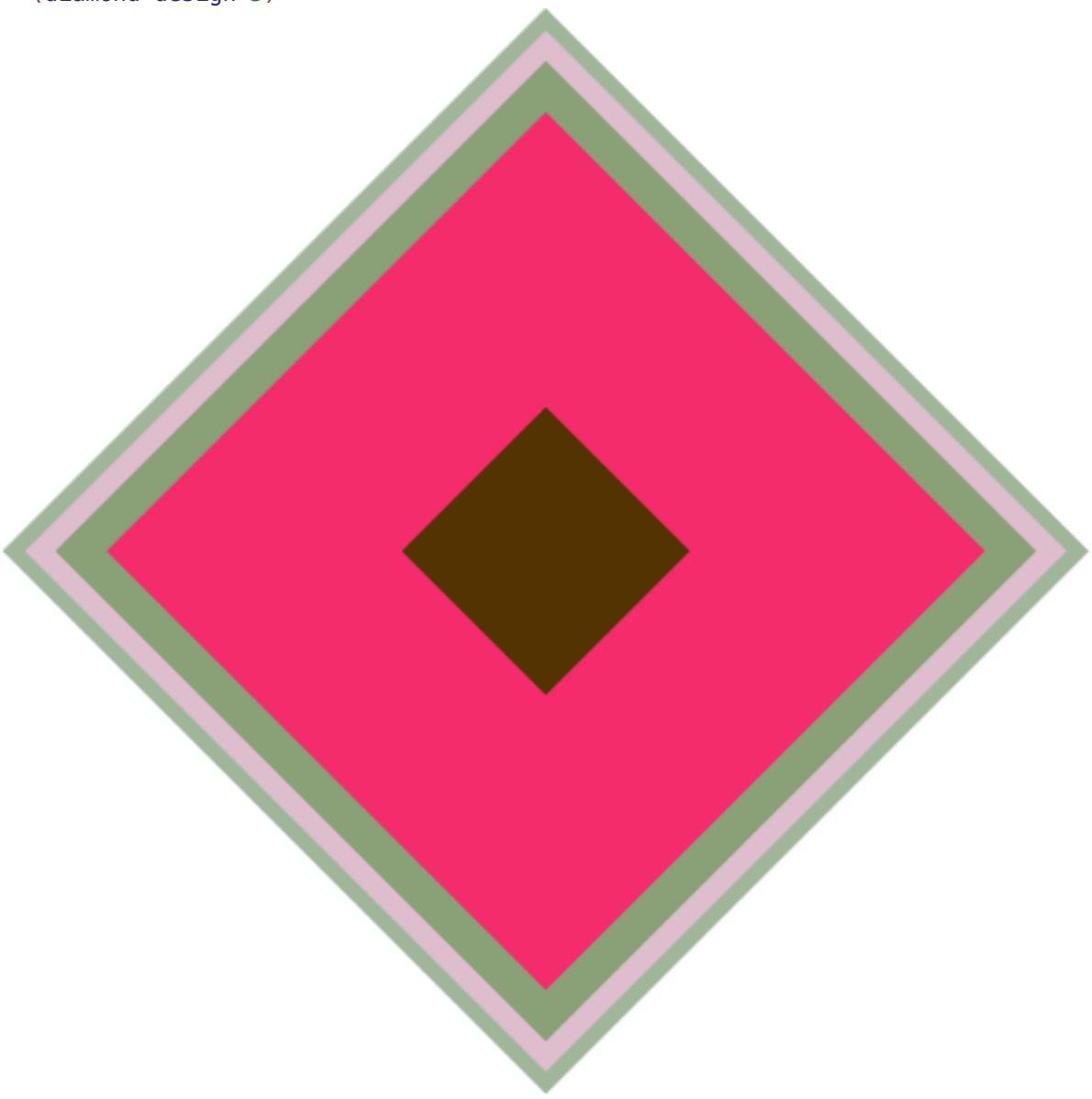
Specification

Using what you learned from Problem 6 as a hint, define a function called `diamond` that is consistent with the following specification:

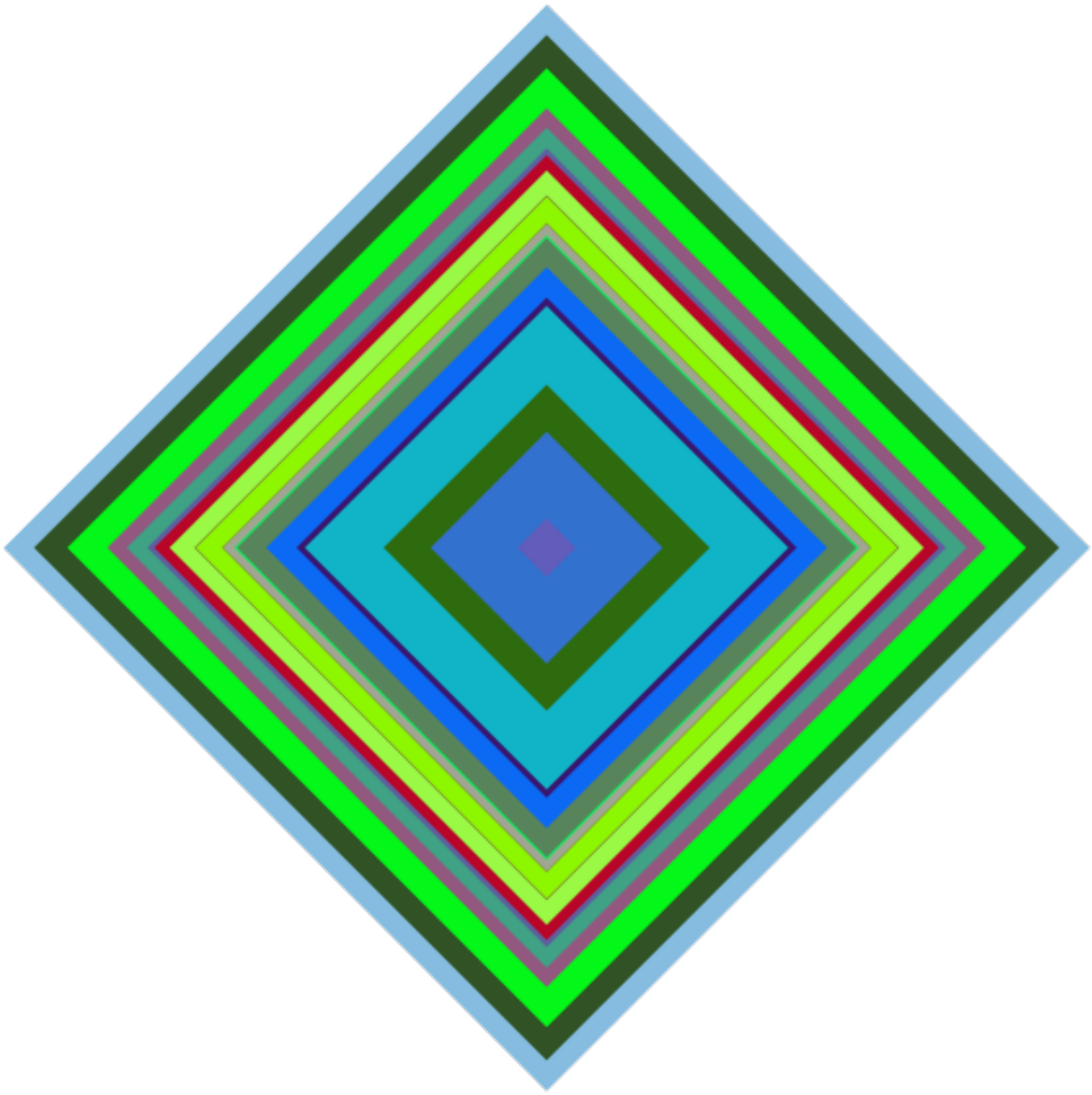
1. The sole parameter is a number indicating how many diamonds will be featured in the design.
2. The function returns an image which consists of the number of diamonds specified by the parameter, where each diamond is randomly colored and has a side length between 20 and 400.

Demo 1 & 2

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (diamond-design 5)
```



```
> (diamond-design 20)
```



Source Code

```
#lang racket
(require 2htdp/image)
;---------
; Programming Assignment 4: Recursive List Processing and Higher Order Functions

;; Task 7: The Diamond


;; Helper Functions
;;-----------------------------------------------------------------
;; From Task 6
(define (generate-list num fn)
  (cond
    ((= 1 num) (make-list 1 (fn)))
    (else
     (append (make-list 1 (fn)) (generate-list (- num 1) fn))
     )
    )
  )

(define (sort-diamonds d)
  (sort d #:key image-width <)
  )

(define (random-color)
  (color (random 256)(random 256)(random 256))
  )

(define (draw-diamond)
  (rotate 45
          (square (+ 21 (random 380)) "solid" (random-color))
          )
  )

(define (diamond-design n)
  (foldr overlay empty-image (sort-diamonds (generate-list n
                                                           draw-diamond)))
  )
```

Problem 8 - Chromesthetic renderings

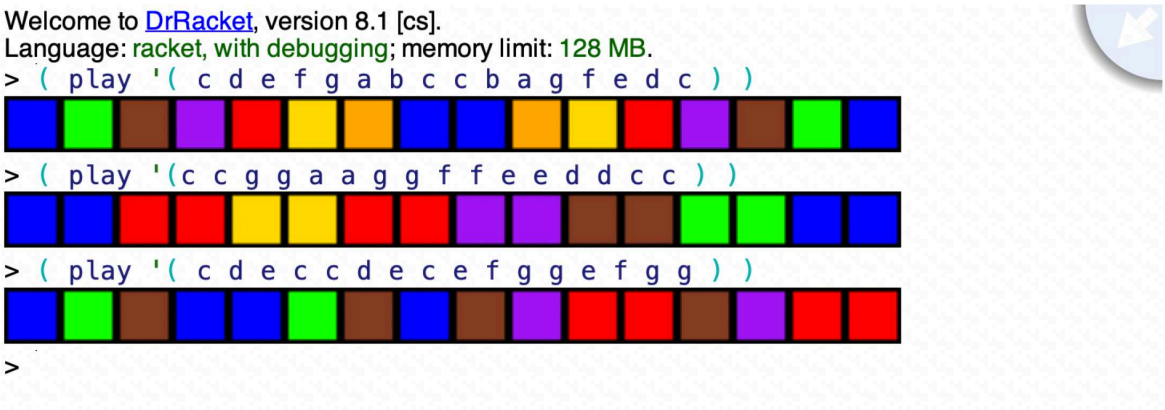
Specification

Define a function called `play` according to the following specification:

1. The sole parameter is a list of pitch names drawn from the set {c, d, e, f, g, a, b}.
2. The result is an image consisting of a sequence of colored squares in black frames, with the colors determined by the following mapping: c→blue; d→green; e→brown; f→purple; g→red; a→gold; b→orange.

Constraint: Your function definition must use `map` twice and `foldr` one time.

Demo



Source Code

```
#lang racket
(require 2htdp/image)
;---------
; Programming Assignment 4: Recursive List Processing and Higher Order Functions

;; Task 8: Chromesthetic Renderings

;; Helper Functions
;;-----------------------------------------------------------------
;; From Task 3
(define (a-list list_a list_b)
  (cond
    ((empty? list_a) '())
    (else
     (cons (cons (car list_a) (car list_b)) (a-list (cdr list_a)
                                                    (cdr list_b)))
     )
    )
  )

;; From Task 4
(define (assoc obj list)
  (cond
    ((empty? list) '())
    ((eq? (car (car list)) obj)
     (car list)
     )
    (else
     (assoc obj (cdr list))
     )
    )
  )


;Given auxiliary definitions
;--------------------------
(define pitch-classes '(c d e f g a b))
(define color-names '(blue green brown purple red yellow
                           orange))
(define (box color)
  (overlay
   (square 30 "solid" color)
   (square 35 "solid" "black")
   )
  )
(define boxes
  (list
   (box "blue")
   (box "green")
   (box "brown")
   (box "purple")
   (box "red")
   (box "gold")
   (box "orange")
   )
  )
(define pc-a-list (a-list pitch-classes color-names))
(define cb-a-list ( a-list color-names boxes ) )
```

```
59 (define (pc->color pc)
60   (cdr (assoc pc pc-a-list))
61   )
62 (define (color->box color)
63   (cdr (assoc color cb-a-list))
64   )
65 ;Function that reads in a series of letter representing notes
66 ;and displays corresponding colored boxes
67 ;-------------------------------------------------------------
68 (define (play lst)
69   (define colors (map pc->color lst))
70   (define squares (map color->box colors))
71   (foldr beside empty-image squares)
72   )
```

Problem 9 - Transformation of a Recursive Sampler

1. Please study the "recursive flip for offset" function, which computes an "offset" from zero that is determined by flipping a coin a specified number of times. The **offset** is essentially the number of heads minus the number of tails, for a given number of flips. Then, study the demo.
2. Define an equivalent function which does not use recursion, but which uses three higher order functions, `generate-list`, `map`, and `foldr`.

Demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (flip-for-offset 100)
10
> (flip-for-offset 100)
2
> (flip-for-offset 100)
-14
> (flip-for-offset 100)
-12
> (flip-for-offset 100)
10
> (demo-for-flip-for-offset)
-18: **
-14: **
-10: *****
-8:  *****
-6:  ********
-4:  **********
-2:  **********
0:   ********
2:   *******
4:   **********
6:   *********
8:   ****
10:  ****
12:  ******
16:  *
18:  *
```

```
> (demo-for-flip-for-offset)
-18: *
-16: *
-14: **
-12: ***
-10: *******
-8:  *********
-6:  ****
-4:  ******
-2:  ********
0:   ************
2:   ********
4:   **************
6:   ************
8:   ****
10:  ***
12:  **
14:  *
16:  **
>
```

Source Code

```
#lang racket
;---------
; Programming Assignment 4: Recursive List Processing and Higher Order Functions

;; Task 9: Transformation of a Recursive Sampler

;; Helper Functions
;;-----------------------------------------------------------------

;; From Task 1
(define (count obj l)
  (cond
    ( (empty? l) 0)
    ( (eq? obj (car l) )
      (+ 1 (count obj (cdr l)) ) )
    (else
     (count obj  (cdr l)) )
    )
  )

;; From Task 2

(define (list->set l)
  (cond
    ((empty? l) l)
    ((member (car l) ( cdr l))
     (list->set (cdr l))
     )
    (else
     (cons (car l) (list->set (cdr l)))
     )
    )
  )

;; From Task 4
(define (a-list list_a list_b)
  (cond
    ((empty? list_a) '())
    (else
     (cons (cons (car list_a) (car list_b)) (a-list (cdr list_a)
                                                    (cdr list_b)))
     )
    )
  )


;; From Task 5
;-----------------------------------
(define (add-blanks s n)
  (cond
    (( = n 0) s)
    (else (add-blanks (string-append s " ") (- n 1) ) )
    )
  )
(define (ft-visualizer ft)
  (map pair-visualizer ft)
  (display "")
  )
```

```
(define (pair-visualizer pair)
  (define label
    (string-append (number->string (car pair) ) ":")
    )
  (define fixed-size-label (add-blanks label ( - 5 ( string-length label) ) ) )

  (display fixed-size-label)
  (display
   (foldr
    string-append
    ""
    (make-list (cdr pair ) "*")
    )
   )
  (display "\n")
  )
(define (ft the-list)
  (define the-set (list->set the-list))
  (define the-counts
    (map (lambda (x) (count x the-list)) the-set)
    )
  (define association-list (a-list the-set the-counts))
  (sort association-list < #:key car)
  )

;; From Task 6
(define (generate-list num fn)
  (cond
    ((= 1 num) (make-list 1 (fn)))
    (else
     (append (make-list 1 (fn)) (generate-list (- num 1) fn))
     )
    )
  )
;-------------------------------------------------------------

;Uses higher-order functions to calculate offset of n coin flips
;-------------------------------------------------------------------
(define (flip-for-offset n)
  (define options '(t h))
  (define flips (generate-list n flip-coin))
  (define totals (map (lambda (x) (count x flips)) options))
  (foldr - (car totals) (cdr totals))
  )
;Creates a frequency table of the offset
;---------------------------------
(define (demo-for-flip-for-offset)
  (define offsets
    (generate-list
     100
     (lambda () (flip-for-offset 50))
     )
    )
  (ft-visualizer (ft offsets))
  )
;Provided helper function that flips a coin
;-----------------------------------------
(define (flip-coin)
```

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

```

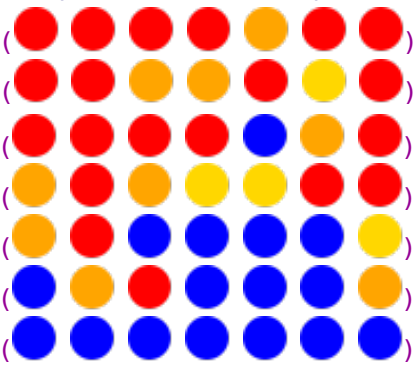
Problem 10 - Blood Preassure Trend Visualizer

Demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (sample 120)
128
> (sample 80)
68
> (data-for-one-day 110)
'(125 91)
> (data-for-one-day 110)
'(129 79)
> (data-for-one-day 110)
'(123 97)
> (data-for-one-day 90)
'(113 73)
> (data-for-one-day 90)
'(99 65)
> (data-for-one-day 90)
'(117 85)
> (data-for-one-week 110)
'((129 91) (139 85) (133 97) (127 89) (133 89) (131 83) (139 83))
> (data-for-one-week 100)
'((120 84) (116 76) (120 80) (100 80) (128 80) (114 76) (128 74))
> (data-for-one-week 90)
'((115 69) (91 67) (99 71) (111 69) (101 75) (123 83) (117 67))
> (define getting-worse '(95 98 100 102 105))
> (define getting-better '(105 102 100 98 95))
> (generate-data getting-worse)
'(((116 76) (104 86) (120 70) (118 72) (116 78) (108 66) (118 72))
  ((117 75) (129 61) (121 79) (119 83) (121 73) (119 77) (115 73))
  ((124 84) (102 80) (102 72) (122 90) (118 86) (112 76) (96 80))
  ((127 77) (121 83) (119 77) (117 77) (135 85) (117 83) (127 93))
  ((129 89) (117 79) (135 75) (121 89) (123 77) (141 81) (119 89)))
> (generate-data getting-better)
'(((131 89) (129 83) (135 87) (139 97) (131 93) (123 87) (129 95))
  ((121 77) (119 71) (115 87) (115 95) (115 81) (123 81) (119 79))
  ((118 78) (120 86) (126 82) (120 74) (120 84) (126 80) (114 90))
  ((115 85) (127 63) (111 67) (117 79) (117 77) (111 79) (109 83))
  ((116 66) (108 74) (126 72) (114 68) (114 84) (124 78) (112 74)))
> (define bad-week (data-for-one-week 110))
> define good-week (data-for-one-week 90))
```

```
> (define good-week (data-for-one-week 90))
> bad-week
'((129 97) (125 75) (135 97) (121 89) (135 97) (139 99) (131 93))
> (display (one-week-visualization bad-week))
```

```
> good-week
'((113 69) (109 83) (119 77) (107 73) (111 67) (101 83) (119 81))
> (display (one-week-visualization good-week))
```

```
> (define bp-data (generate-data '(110 105 102 100 98 95 90)))
> (bp-visualization bp-data)
```

```
>
```

Source Code

```
#lang racket
(require 2htdp/image)
;---------
; Programming Assignment 4: Recursive List Processing and Higher Order Functions

;; Task 10: Blood Preassure Trend Visualizer

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

(define (count t l)
  (cond
    ((empty? l) 0)
    ((eq? t (car l))
     (+ 1 (count t (cdr l))))
    (else
     (count t (cdr l)))
    )
  )

(define (flip-for-offset n)
  (define options '(t h))
  (define flips (generate-list n flip-coin))
  (define totals (map (lambda (x) (count x flips)) options))
  (foldr - (car totals) (cdr totals))
  )

(define (generate-list num fn)
  (cond
    ((= 1 num) (make-list 1 (fn)))
    (else
     (append (make-list 1 (fn)) (generate-list (- num 1) fn))
     )
    )
  )

(define (sample cardio-index)
  (+ cardio-index (flip-for-offset (quotient cardio-index 2)))
  )

(define (data-for-one-day middle-base)
  (list
   (sample (+ middle-base 20))
   (sample (- middle-base 20))
   )
  )

(define (data-for-one-week middle-base)
  (generate-list
```

```
   7
   (lambda () (data-for-one-day middle-base))
   )
  )

(define (generate-data base-sequence)
  (map data-for-one-week base-sequence)
  )

(define (one-day-visualization lst)
  (cond
    ((and (<= 120 (car lst))(<= 80 (cadr lst)))
     (circle 10 "solid" "red")
     )
    ((and (<= 120 (car lst))(> 80 (cadr lst)))
     (circle 10 "solid" "gold")
     )
    ((and (> 120 (car lst))(<= 80 (cadr lst)))
     (circle 10 "solid" "orange")
     )
    (else
     (circle 10 "solid" "blue")
     )
    )
  )

(define (one-week-visualization lst)
  (define dots (map one-day-visualization lst))
  dots
  )

(define (bp-visualization lst)
  (define dots (map one-week-visualization lst))
  (display-dots dots)
  )

(define (display-dots lst)
  (cond
    ((empty? lst)
     (display empty-image)
     )
    (else
     (display (car lst))
     (display "\n")
     (display-dots (cdr lst))
     )
    )
  )


```