Racket Programming Assignment #4: Recursive List Processing and Higher Order Functions

Learning Abstract

This problem set focuses on taking problems and solving them using simple recursion. While we have looked at recursion before this specifically works with recursive list processing. The second part of this assignment takes the recursive list processing and adds in higher order functions such as `map` and `foldr`.

Problem 1 - count Function

Program that counts occurrences of a given atom in a given list.

Demo:

```
1 >  ( count 'b '(a a b a b c a b c d) )
2 3
3 >  ( count 5 '(1 5 2 5 3 5 4 5) )
4 4
5 > ( count 'cherry '(apple peach blueberry) )
6 0
7 >
```

Source Code:

```
1 #lang racket
2
3 ( define ( count c list )
4    ( cond
5       (( empty? list )
6        0
7        )
8       (( equal? ( car list ) c )
9        ( + 1 ( count c ( cdr list ) ) )
10        )
11       ( else
12         ( count c ( cdr list ) )
13         )
14       )
15    )
```

Problem 2 - list->set Function

Program that takes a given list and reduces it to one occurrence of each unique atom.

Demo:

```
1 >  ( list->set '(a b c b c d c d e) )
2 '(a b c d e)
3 >  ( list->set '(1 2 3 2 3 4 3 4 5 4 5 6) )
4 '(1 2 3 4 5 6)
5 >  ( list->set '(apple banana apple banana cherry) )
6 '(apple banana cherry)
7 >
```

Source Code:

```
 1 #lang racket
2
3 ( define ( count c list )
4    ( cond
5       (( empty? list )
6        0
7        )
8       (( equal? ( car list ) c )
9        ( + 1 ( count c ( cdr list ) ) )
10        )
11       ( else
12         ( count c ( cdr list ) )
13         )
14       )
15    )
16
17 ( define ( list->set list )
18    ( cond
19       (( empty? list )
20        '()
21        )
22       ( ( not ( = 0 ( count ( car list ) ( cdr list ) ) ) )
23        ( list->set ( cdr list ) )
24        )
25       ( else
26         ( cons ( car list ) ( list->set ( cdr list ) ) )
27         )
28       )
29    )
```

Problem 3 - Association List Generator

Program that takes two lists of equal size and returns a list of dotted pairs.

Demo:

```
1 > ( a-list '(one two three four five) '(un deux trois quatre cinq) )
2 '((one . un) (two . deux) (three . trois) (four . quatre) (five . cinq))
3 > ( a-list '() '() )
4 '()
5 > ( a-list '( this ) '( that ) )
6 '((this . that))
7 > ( a-list '(one two three) '( (1) (2 2) ( 3 3 3 ) ) )
8 '((one 1) (two 2 2) (three 3 3 3))
9 >
```

Source Code:

```
1 #lang racket
2
3 ( define ( a-list list-1 list-2 )
4    ( cond
5       (( empty? list-1 )
6        '()
7        )
8       ( else
9         ( cons ( cons ( car list-1 ) ( car list-2 ) ) ( a-list ( cdr list-1 )
( cdr list-2 ) ) )
10         )
11       )
12    )
```

Problem 4 - Assoc

Program that returns a dotted pair from an association list if the first element of the pair matches the given atom.

Demo:

```
1 > ( define al1
2      ( a-list '(one two three four ) '(un deux trois quatre ) )
3      )
4 > ( define al2
5       ( a-list '(one two three) '( (1) (2 2) (3 3 3) ) )
6       )
7 > al1
8 '((one . un) (two . deux) (three . trois) (four . quatre))
9 > ( assoc 'two al1 )
10 '(two . deux)
11 > ( assoc 'five al1 )
```

```
12 '()
13 > al2
14 '((one 1) (two 2 2) (three 3 3 3))
15 > ( assoc 'three al2 )
16 '(three 3 3 3)
17 > ( assoc 'four al2 )
18 '()
19 >
```

Source Code:

```
1 #lang racket
2
3 ( define ( a-list list-1 list-2 )
4    ( cond
5       (( empty? list-1 )
6        '()
7        )
8       ( else
9         ( cons ( cons ( car list-1 ) ( car list-2 ) ) ( a-list ( cdr list-1 )
( cdr list-2 ) ) )
10         )
11       )
12    )
13
14 ( define ( assoc index assoc-list )
15    ( cond
16       (( empty? assoc-list )
17        '()
18        )
19       (( equal? index ( car ( car assoc-list ) ) )
20        ( car assoc-list )
21        )
22       ( else
23         ( assoc index ( cdr assoc-list ) )
24         )
25       )
26    )
```

Problem 5 - Frequency Table

Function and that creates a visual representation of a given list in terms of frequency. Could be used to represent collected data, ie how many 10's, 9's, etc.

Demo:

```
1 > ( 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 ) ) )
2 >  ft1
3 '((1 . 4) (2 . 3) (3 . 2) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 2) (9 . 3)
      (10 . 4))
4 >  ( ft-visualizer ft1 )
5 1:   ****
6 2:   ***
7 3:   **
8 4:   *
9 5:   *
10 6:  *
11 7:  *
12 8:  **
13 9:  ***
14 10: ****
15 >  ( define ft2 ( ft '( 1 10 2 9 3 8 4 4 7 7 6 6 6 5 5 5 ) ) )
16 > ft2
17 '((1 . 1) (2 . 1) (3 . 1) (4 . 2) (5 . 3) (6 . 3) (7 . 2) (8 . 1) (9 . 1)
       (10 . 1))
18 > ( ft-visualizer ft2 )
19 1:  *
20 2:  *
21 3:  *
22 4:  **
23 5:  ***
24 6:  ***
25 7:  **
26 8:  *
27 9:  *
28 10: *
29 >
```

Source Code:

```
1 ( define ( ft the-list )
2    ( define the-set ( list->set the-list ) )
3    ( define the-counts
4       ( map ( lambda (x) ( count x the-list ) ) the-set )
5       )
6    ( define association-list ( a-list the-set the-counts ) )
7    ( sort association-list < #:key car )
8    )
9
10 ( define ( ft-visualizer ft )
11    ( map pair-visualizer ft )
```

```
12    ( display "" )
13    )
14
15 ( define ( pair-visualizer pair )
16    ( define label
17       ( string-append ( number->string ( car pair ) ) ":" )
18       )
19    ( define fixed-size-label ( add-blanks label ( - 5 ( string-length label
) ) ) )
20    ( display fixed-size-label )
21    ( display
22      ( foldr
23        string-append
24        ""
25        ( make-list ( cdr pair ) "*" )
26        )
27      )
28    ( display "\n" )
29    )
30
31 ( define ( add-blanks s n )
32    ( cond
33       (( = n 0 )s)
34       ( else
35         ( add-blanks ( string-append s " " ) ( - n 1 ) )
36         )
37       )
38    )
```

Question:

1) List the names of the functions used within the ft function that you were asked to write in this programming assignment.
    a) `count`, `list->set`, and `a-list` all were used by these functions.
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) This lambda function is defined as: `( lambda (x) ( count x the-list ) )`
3) How many parameters must the functional argument to the application of map in the ft function take?
    a) One `x`
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-list` and `the-set` are likely different lengths making any type of recursion between both difficult.

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) `<` or to order the list in increasing order
6) What is a “keyword argument”?
    a) A specific word that begins with a `#`
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) It is not running through two lists at the same time and is also too complicated to be reasonably written as a lambda function.
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) No
11) Why do you think the display function, with the empty string as its argument, was called in the ft-visualizer function?
    a) Without the empty string the function attempts to print out a list of void elements
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) An association list
13) Is the make-list function used in the pair-visualizer function a primitive function in Racket?
    a) Yes it is imported directly from racket
14) What do you think is the most interesting aspect of the given frequency table generating code?
    a) Adding on the blank spaces at the end of each line
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) How could this program be created without using higher order functions?

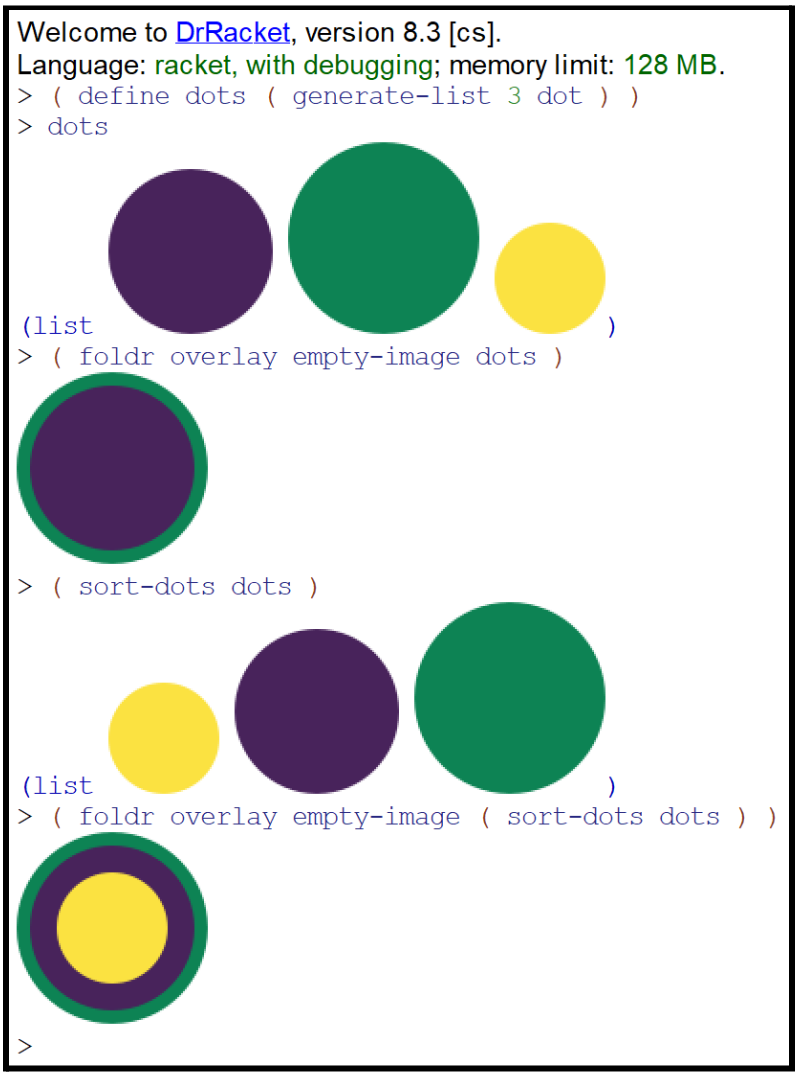
Problem 6- Generate List

Function that creates a list of n number lisp objects created by a parameterless function.

Demo 1:

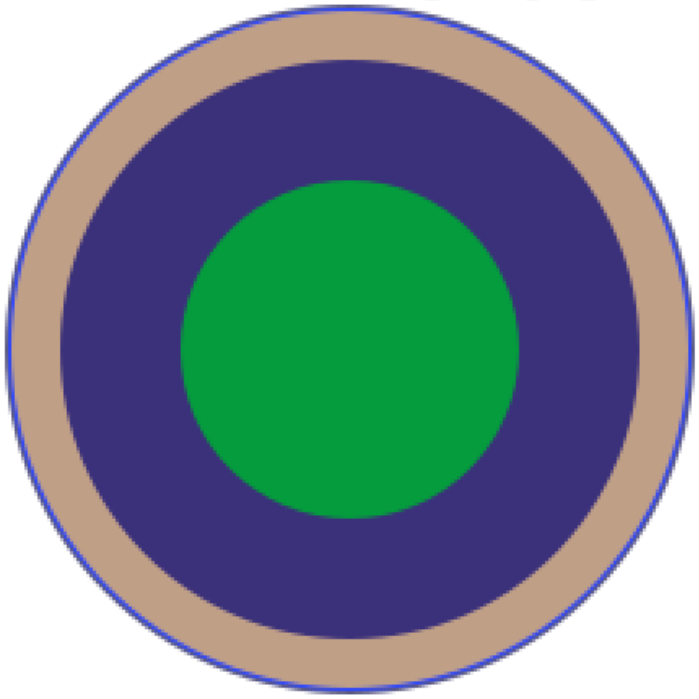
```
1 > ( generate-list 10 roll-die )
2 '(4 4 3 5 3 1 1 2 5 2)
3 > ( generate-list 20 roll-die )
4 '(3 3 2 3 4 1 5 1 6 4 5 1 6 2 4 5 4 5 2 2)
5 > ( generate-list 12
6                   ( lambda () ( list-ref '( red yellow blue ) ( random 3 ) ) )
7                    )
8 '(red blue yellow yellow red blue yellow yellow blue red red blue)
9 >
```

Demo 2:

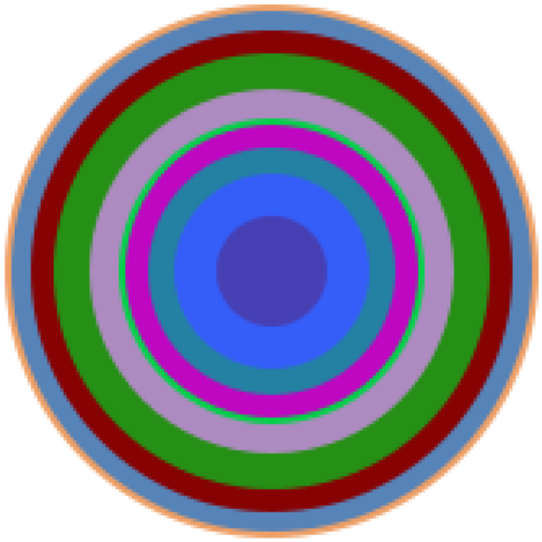


Demo 3:

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( 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 ) )
```



```
>
```

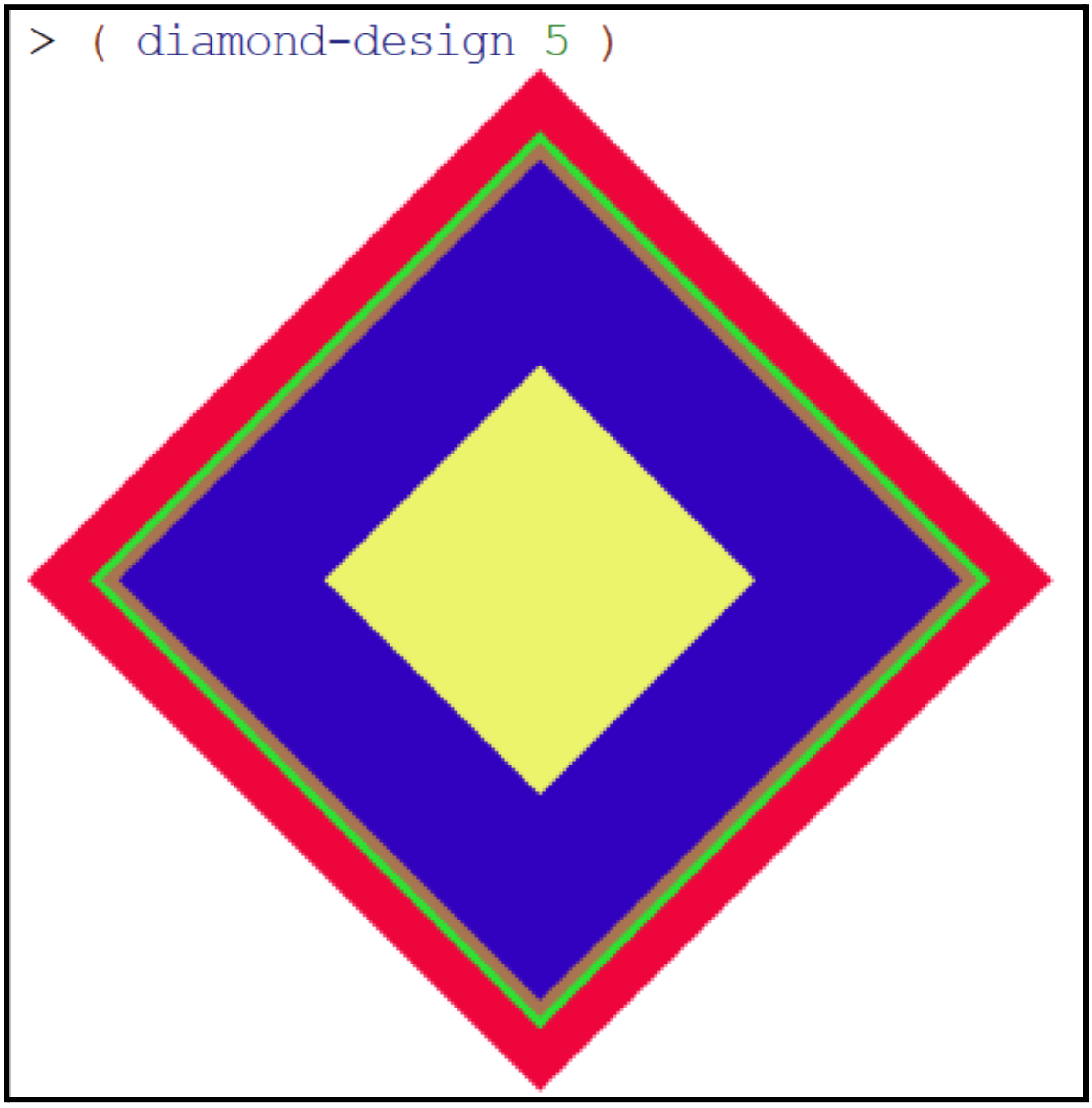
Source Code:

```
#lang racket
(require 2htdp/image)

;-----------
; aux code to help with demos

( define ( roll-die ) ( + ( random 6 ) 1 ) )

( define ( dot )
   ( circle ( + 10 ( random 41 ) ) "solid" ( random-color ) )
   )

( define ( big-dot )
   ( circle ( + 10 ( random 101 ) ) "solid" ( random-color ) )
   )

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

( define ( sort-dots loc )
   ( sort loc #:key image-width < )
   )

;-----
; generate-list function

( define ( generate-list n fxn )
   (cond
     (( = 0 n )
      '()
      )
     ( else
       ( cons ( fxn ) ( generate-list ( - n 1 ) fxn ) )
       )
     )
   )
```

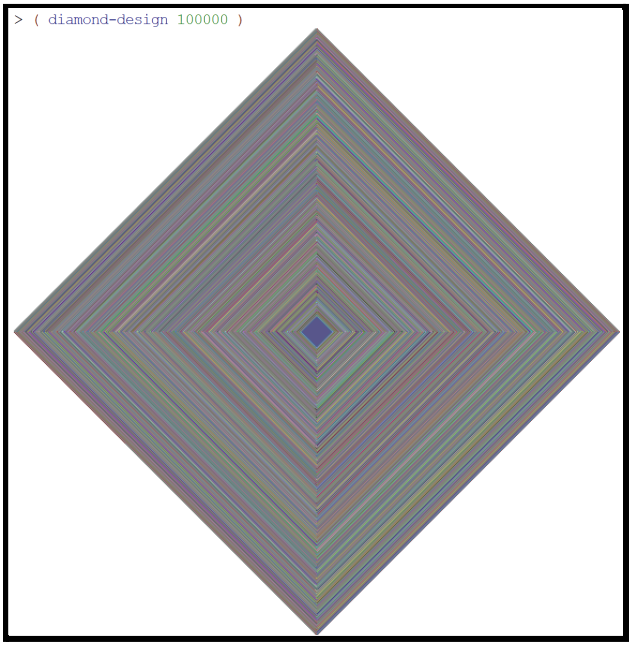
Problem 7 - The Diamond

Function that creates a picture of diamonds size n overlaid on another to form a Frank Stella esc image.

Demo:



I also pushed it pretty far



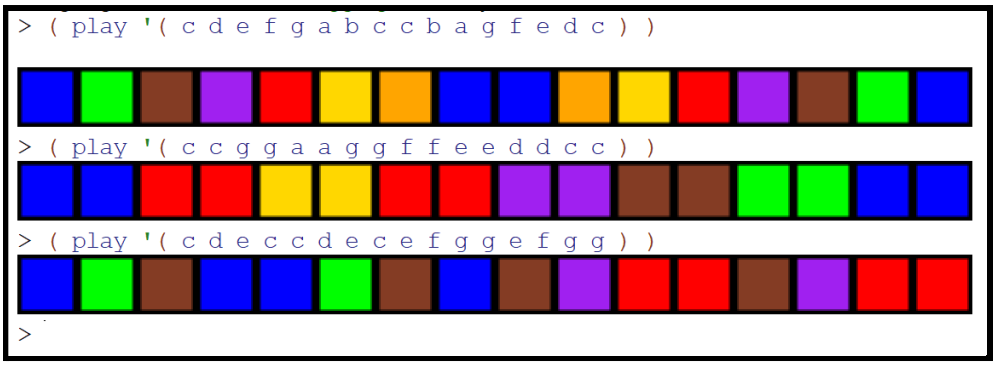
Source Code:

```
#lang racket
(require 2htdp/image)

;-----------
; aux code

( define ( diamond )
   ( rotate 45 ( square ( + 20 ( random 381 ) ) "solid" ( random-color ) ) )
   )

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

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

;-----
; generate-list function

( define ( generate-list n fxn )
   (cond
     (( = 0 n )
      '()
      )
     ( else
       ( cons ( fxn ) ( generate-list ( - n 1 ) fxn ) )
       )
     )
   )

;---------
( define ( diamond-design n )

   ; create list of diamonds
   ( define b ( generate-list n diamond ) )

   ; put list in order and overlay it on itself
   ( foldr overlay empty-image ( sort-diamonds b ) )
   )
```

Problem 8 - Chromesthetic Renderings

Function that creates a visual representation of letters presumed to be pitches. It takes one list as a parameter.

Demo:



Source Code:

```
(require 2htdp/image)

;--------
; a-lsit and assoc

( define ( a-list list-1 list-2 )
   ( cond
      (( empty? list-1 )
       '()
       )
      ( else
        ( cons ( cons ( car list-1 ) ( car list-2 ) ) ( a-list ( cdr list-1 ) ( cdr list-2 ) ) )
        )
      )
   )

( define ( assoc index assoc-list )
   ( cond
      (( empty? assoc-list )
       '()
       )
      (( equal? index ( car ( car assoc-list ) ) )
       ( car assoc-list )
       )
      ( else
        ( assoc index ( cdr assoc-list ) )
        )
      )
   )

;--------
```

```
; aux code

( 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 ) )

( define ( pc->color pc )
   ( cdr ( assoc pc pc-a-list ) )
   )

( define ( color->box color )
   ( cdr ( assoc color cb-a-list ) )
   )

;--------
; play function

;must use map twice and foldr once
( define ( play measure )
   ; make list of notes into list of colors
   ( define measure-as-colors ( map pc->color measure ) )
   ; make list of colors into list of boxes
   ( define measure-as-boxes ( map color->box measure-as-colors ) )
   ; foldr the list
   ( foldr beside empty-image measure-as-boxes )
   )
```

Problem 9 - Flip-for-offset

Function that creates a visual representation of flipping a coin to calculate for the offset. The offset being the number of heads minus the number of tails.

Demo:

```
1 > ( flip-for-offset 100 )
2 -6
3 > ( flip-for-offset 100 )
4 -6
5 > ( flip-for-offset 100 )
6 -4
7 > ( flip-for-offset 100 )
8 -2
9 > ( flip-for-offset 100 )
10 -6
11 > ( demo-for-flip-for-offset )
12 -16: **
13 -14: *
14 -10: ****
15 -8:  **********
16 -6:  *******
17 -4:  **************
18 -2:  **************
19 0:   ***********
20 2:   ***
21 4:   ***********
22 6:   *****
23 8:   ********
24 10:  ****
25 12:  ***
26 16:  *
27 20:  **
28 > ( demo-for-flip-for-offset )
29 -22: *
30 -18: *
31 -14: **
32 -12: *
33 -10: **********
34 -8:  *****
35 -6:  ****
36 -4:  ************
37 -2:  **********
38 0:   ******
39 2:   *******
40 4:   ***************
41 6:   *********
42 8:   ********
43 10:  ****
44 12:  **
```

```
45 14:  *
46 16:  *
47 18:  *
48 >
```

Source Code: This source code is messy and I did my best to help organize with comments, but the first part is auxiliary code followed by one new supporting function and finishing with the flip-for-offset code

```
1 #lang racket
2
3 ;---------------------------------------
4 ; aux code
5
6 ;----- count
7
8 ( define ( count c list )
9    ( cond
10       (( empty? list )
11        0
12        )
13       (( equal? ( car list ) c )
14        ( + 1 ( count c ( cdr list ) ) )
15        )
16       ( else
17         ( count c ( cdr list ) )
18         )
19       )
20    )
21
22 ( define ( list->set list )
23    ( cond
24       (( empty? list )
25        '()
26        )
27       ( ( not ( = 0 ( count ( car list ) ( cdr list ) ) ) )
28        ( list->set ( cdr list ) )
29        )
30       ( else
31         ( cons ( car list ) ( list->set ( cdr list ) ) )
32         )
33       )
34    )
35
36 ;----- generate-list
37 ( define ( generate-list n fxn )
38    (cond
39      (( = 0 n )
40       '()
41       )
42      ( else
```

```
43        ( cons ( fxn ) ( generate-list ( - n 1 ) fxn ) )
44        )
45      )
46    )
47
48 ;------a-list
49
50 ( define ( a-list list-1 list-2 )
51    ( cond
52       (( empty? list-1 )
53        '()
54        )
55       ( else
56         ( cons ( cons ( car list-1 ) ( car list-2 ) ) ( a-list ( cdr list-1
) ( cdr list-2 ) ) )
57         )
58       )
59    )
60
61 ( define ( assoc index assoc-list )
62    ( cond
63       (( empty? assoc-list )
64        '()
65        )
66       (( equal? index ( car ( car assoc-list ) ) )
67        ( car assoc-list )
68        )
69       ( else
70         ( assoc index ( cdr assoc-list ) )
71         )
72       )
73    )
74
75 ;------ ft-vis
76
77 ( define ( ft the-list )
78    ( define the-set ( list->set the-list ) )
79    ( define the-counts
80       ( map ( lambda (x) ( count x the-list ) ) the-set )
81       )
82    ( define association-list ( a-list the-set the-counts ) )
83    ( sort association-list < #:key car )
84    )
85
86 ( define ( ft-visualizer ft )
87    ( map pair-visualizer ft )
88    ( display "" )
89    )
90
```

```
91 ( define ( pair-visualizer pair )
92    ( define label
93       ( string-append ( number->string ( car pair ) ) ":" )
94       )
95    ( define fixed-size-label ( add-blanks label ( - 5 ( string-length label
) ) ) )
96    ( display fixed-size-label )
97    ( display
98      ( foldr
99        string-append
100        ""
101        ( make-list ( cdr pair ) "*" )
102        )
103      )
104    ( display "\n" )
105    )
106
107 ( define ( add-blanks s n )
108    ( cond
109       (( = n 0 )s)
110       ( else
111         ( add-blanks ( string-append s " " ) ( - n 1 ) )
112         )
113       )
114    )
115
116 ;--------
117 ; new aux functions
118
119 ; given
120 ( define ( flip-coin )
121    ( define outcome ( random 2 ) )
122    ( cond
123       ( ( = outcome 1 )
124         'h
125         )
126       ( ( = outcome 0 )
127         't
128         )
129       )
130    )
131
132 ;ht->number function that maps all 'h to 1 and all 't to -1
133 ( define ( ht->number c )
134    (cond
135      (( equal? 'h c )
136       1
137       )
138      ( else
```

```
        -1
        )
      )
    )

;--------------------
; non-recursive-flip-for-offset

; must use generate-list map and foldr

( define ( flip-for-offset n )
    ; use gen-lsit to make a long list
    ( define result-list ( generate-list n flip-coin ) )
    ; use map to make each h a 1 and each t a -1
    ( define result-list-numbers ( map ht->number result-list ) )
    ; use foldr to add
    ( foldr + 0 result-list-numbers )
    )

( define ( demo-for-flip-for-offset )
    ( define offsets
       ( generate-list
         100
         ( lambda () ( flip-for-offset 50 ) )
         )
       )
    ( ft-visualizer (ft offsets ) )
    )
```

Problem 10 - Blood Pressure Visualizer

Program that creates and visualizes a set of data related to blood pressure building off of all the functions created in this assignment

Demo:

```
1 > ( sample 120 )
2 120
3 > ( sample 80 )
4 82
5 >  ( data-for-one-day 110 )
6 '(135 81)
7 >  ( data-for-one-day 110 )
8 '(127 85)
9 >  ( data-for-one-day 110 )
10 '(121 83)
11 >  ( data-for-one-day 110 )
12 '(129 87)
13 >  ( data-for-one-day 110 )
14 '(121 95)
15 > ( data-for-one-week 110 )
16 '((135 91) (115 87) (117 87) (135 97) (141 99) (131 95) (125 91))
17 > ( data-for-one-week 110 )
18 '((141 79) (131 93) (137 87) (127 101) (137 89) (127 89) (127 91))
19 > ( data-for-one-week 110 )
20 '((139 95) (123 85) (137 73) (139 97) (127 87) (129 85) (127 83))
21 > ( define getting-worse '(95 98 100 102 105) )
22 >  ( define getting-better '(105 102 100 98 95) )
23 > ( generate-data getting-worse )
24 '(((114 82) (112 74) (116 76) (112 66) (112 76) (118 86) (116 74))
25   ((113 75) (117 71) (121 89) (123 83) (119 79) (121 65) (129 87))
26   ((122 84) (128 70) (110 78) (112 80) (120 78) (122 70) (114 76))
27   ((125 69) (121 85) (131 75) (123 67) (129 79) (121 83) (131 83))
28   ((123 79) (125 97) (139 75) (123 85) (125 85) (119 85) (133 77)))
29 >  ( generate-data getting-better )
30 '(((127 89) (135 87) (135 91) (119 75) (133 91) (131 97) (111 93))
31   ((113 87) (111 77) (129 79) (107 89) (131 85) (127 87) (127 83))
32   ((124 74) (126 72) (130 74) (124 72) (132 74) (112 74) (128 90))
33   ((117 91) (119 83) (117 71) (115 77) (131 81) (105 95) (119 85))
34   ((112 80) (118 76) (118 74) (108 78) (108 80) (112 76) (118 76)))
35 >
```

```
> ( one-day-visualization '(125 83 ) )
```


```
> ( one-day-visualization '( 125 78 ) )
```

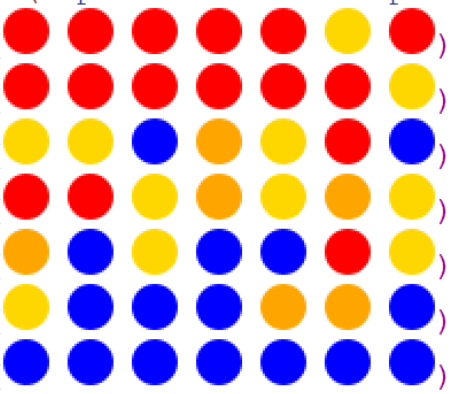

```
> ( one-day-visualization '( 116 87 ) )
```


```
> ( one-day-visualization '( 114 75 ) )
```


```
> |
```

```
> ( define bad-week ( data-for-one-week 110 ) )
> ( define good-week ( data-for-one-week 90 ) )
> bad-week
'((125 75) (133 99) (133 89) (131 79) (129 95) (131 99) (125 79))
> ( one-week-visualization bad-week )
```


```
> good-week
'((117 73) (107 75) (105 73) (115 77) (131 77) (115 67) (105 69))
> ( one-week-visualization good-week )
```


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


```
>
```

Source Code: This code relies on all code from the previous problem 9, but I felt including all 163 lines would be a mess and opted on leaving this note instead

```
1 #lang racket
2 (require 2htdp/image)
3
4 ( define ( sample cardio-index )
5    ( + cardio-index ( flip-for-offset ( quotient cardio-index 2)))
6    )
7
8 ( define ( data-for-one-day middle-base )
9    ( list
10      ( sample ( + middle-base 20 ) )
11      ( sample ( - middle-base 20 ) )
12      )
13    )
14
15 ( define ( data-for-one-week middle-base )
16    ( generate-list
17      7
18      ( lambda () ( data-for-one-day middle-base ) )
19      )
20    )
21
22 ( define ( generate-data base-sequence )
23    ( map data-for-one-week base-sequence )
24    )
25
26 ( define ( one-day-visualization list )
27    ( define n1 ( car list ) )
28    ( define n2 ( car ( cdr list ) ) )
29    (cond
30      (( and ( > n1 119 ) ( > n2 79 ) )
31       ( circle 10 "solid" "red" )
32       )
33      (( and ( > n1 119 ) ( < n2 80 ) )
34       ( circle 10 "solid" "gold" )
35       )
36      (( and ( < n1 120 ) ( > n2 79 ) )
37       ( circle 10 "solid" "orange" )
38       )
39      (( and ( < n1 120 ) ( < n2 80 ) )
40       ( circle 10 "solid" "blue" )
41       )
42      )
43    )
44
45 ( define ( one-week-visualization list )
46    ( display ( map one-day-visualization list ) ) ( display "\n" )
47    empty-image
```

```
48    )
49
50 ( define ( bp-visualization list )
51    ( define visual-list ( map one-week-visualization list ) )
52    ( map write visual-list )
53    empty-image
54    )
55
```