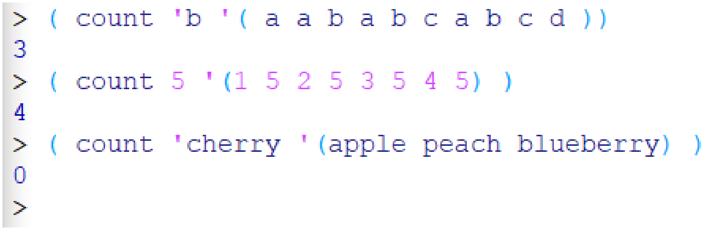
Sven Kappeler - Racket Assignment #4:

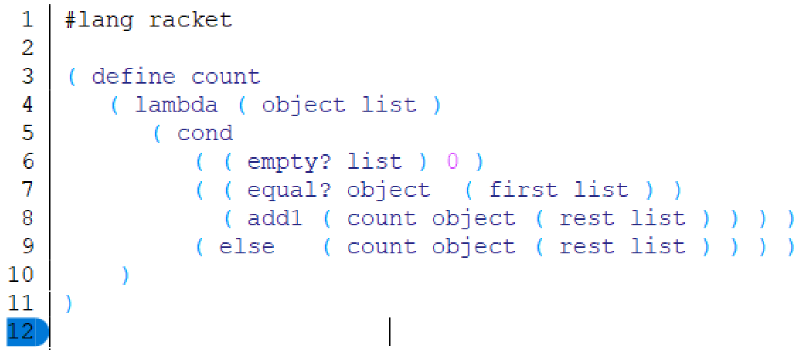
Learning Abstract

This assignment is a continuation of the Racket programming language. The purpose of this assignment is to get myself acquainted with using higher-order functions in Racket.

Task 1: Count

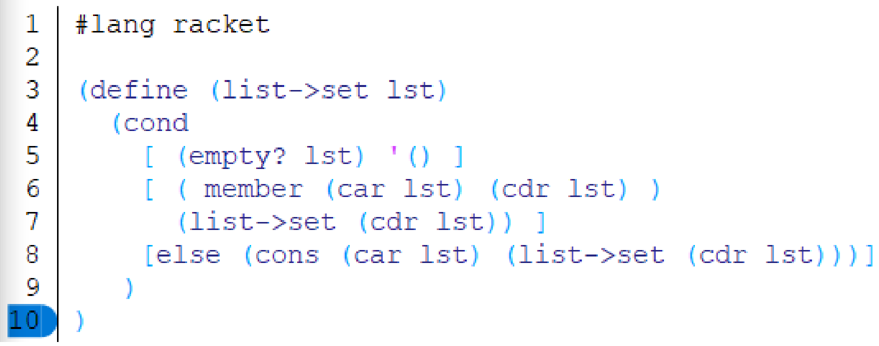


```
> ( count 'b '( a a b a b c a b c d ))
3
> ( count 5 '(1 5 2 5 3 5 4 5) )
4
> ( count 'cherry '(apple peach blueberry) )
0
>
```

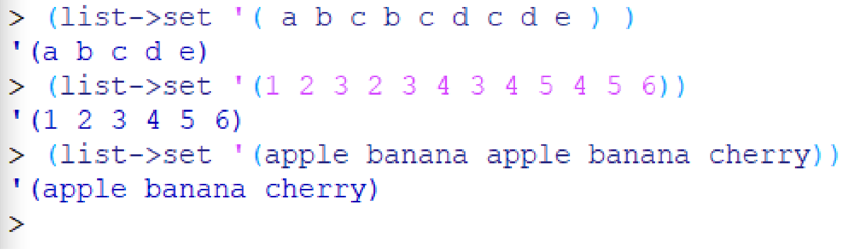


```
#lang racket

( define count
   ( lambda ( object list )
      ( cond
         ( ( empty? list ) 0 )
         ( ( equal? object  ( first list ) )
           ( add1 ( count object ( rest list ) ) ) )
         ( else   ( count object ( rest list ) ) ) )
     )
)

```

Task 2: List -> Set

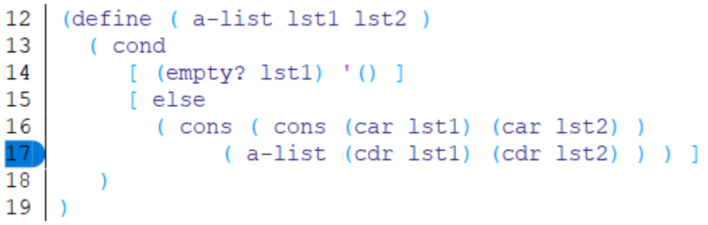


```
#lang racket

(define (list->set lst)
  (cond
    [ (empty? lst) '() ]
    [ ( member (car lst) (cdr lst) )
      (list->set (cdr lst)) ]
    [else (cons (car lst) (list->set (cdr lst)))]
   )
)
```

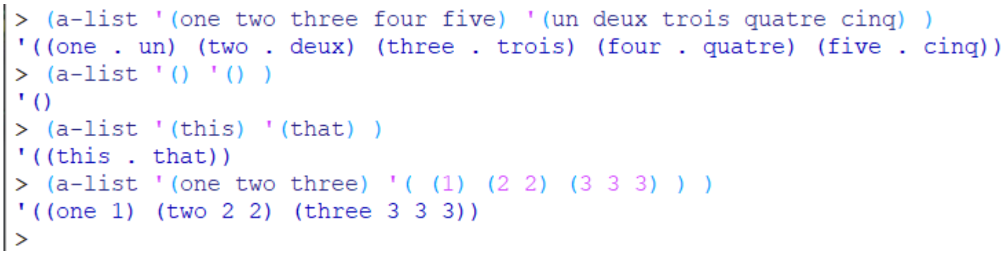


```
> (list->set '( a b c b c d c d e ) )
'(a b c d e)
> (list->set '(1 2 3 2 3 4 3 4 5 4 5 6))
'(1 2 3 4 5 6)
> (list->set '(apple banana apple banana cherry))
'(apple banana cherry)
>
```

Task 3: Association List Generator

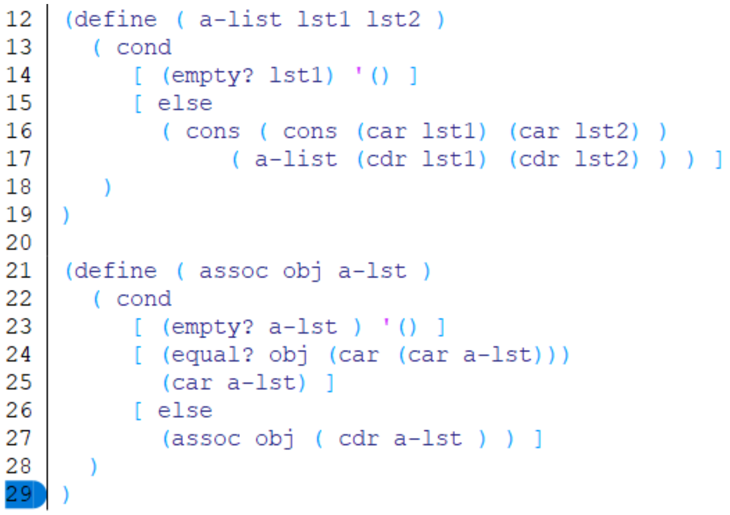


```
(define ( a-list lst1 lst2 )
  ( cond
     [ (empty? lst1) '() ]
     [ else
       ( cons ( cons (car lst1) (car lst2) )
            ( a-list (cdr lst1) (cdr lst2) ) ) ]
   )
)
```

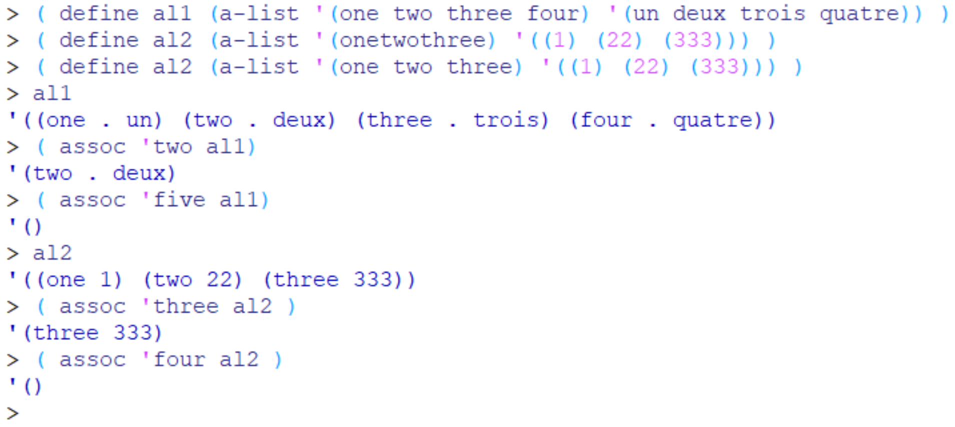


```
> (a-list '(one two three four five) '(un deux trois quatre cinq) )
'((one . un) (two . deux) (three . trois) (four . quatre) (five . cinq))
> (a-list '() '() )
'()
> (a-list '(this) '(that) )
'((this . that))
> (a-list '(one two three) '( (1) (2 2) (3 3 3) ) )
'((one 1) (two 2 2) (three 3 3 3))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

Task 4: Assoc

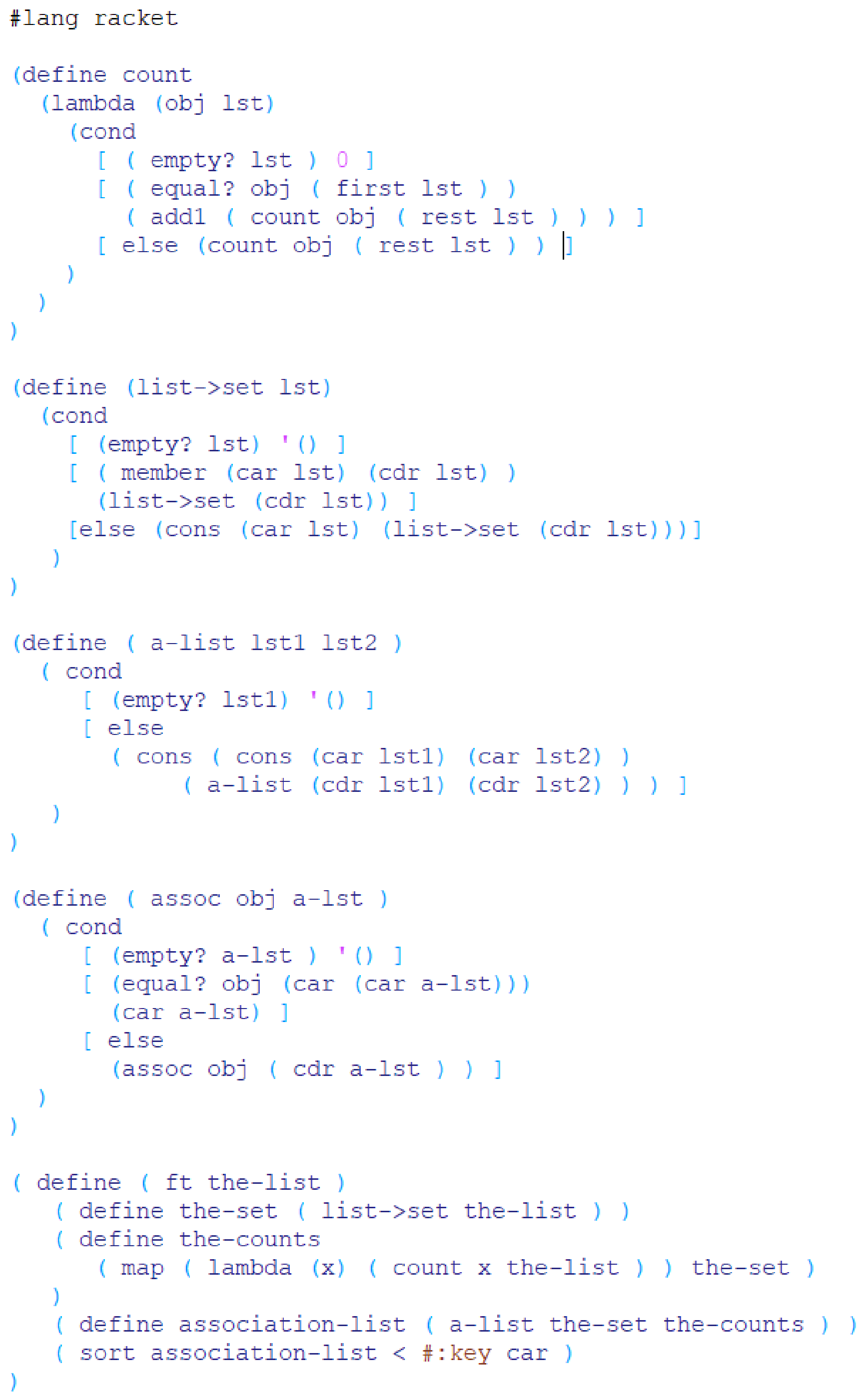


```
(define ( a-list lst1 lst2 )
  ( cond
     [ (empty? lst1) '() ]
     [ else
       ( cons ( cons (car lst1) (car lst2) )
            ( a-list (cdr lst1) (cdr lst2) ) ) ) ]
   )
)

(define ( assoc obj a-lst )
  ( cond
     [ (empty? a-lst ) '() ]
     [ (equal? obj (car (car a-lst)))
       (car a-lst) ]
     [ else
       (assoc obj ( cdr a-lst ) ) ]
  )
)
```



```
> ( define al1 (a-list '(one two three four) '(un deux trois quatre)) )
> ( define al2 (a-list '(onetwothree) '((1) (22) (333))) )
> ( define al2 (a-list '(one two three) '((1) (22) (333))) )
> al1
'((one . un) (two . deux) (three . trois) (four . quatre))
> ( assoc 'two al1)
'(two . deux)
> ( assoc 'five al1)
'()
> al2
'((one 1) (two 22) (three 333))
> ( assoc 'three al2 )
'(three 333)
> ( assoc 'four al2 )
'()
>
```

Task 5: Frequency Table



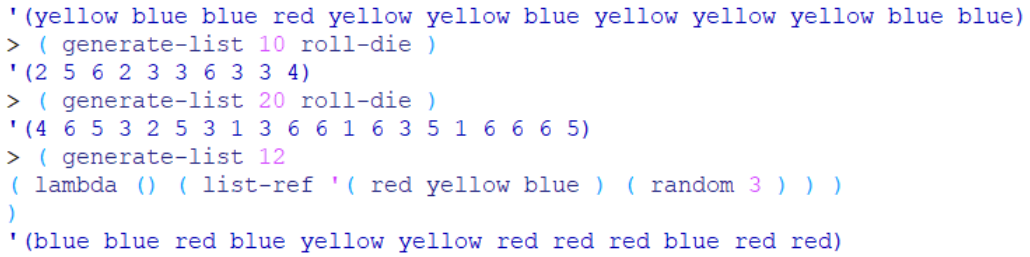
```
#lang racket

(define count
  (lambda (obj lst)
    (cond
      [ ( empty? lst ) 0 ]
      [ ( equal? obj ( first lst ) )
        ( add1 ( count obj ( rest lst ) ) ) ) ]
      [ else (count obj ( rest lst ) ) ]
    )
  )
)

(define (list->set lst)
  (cond
    [ (empty? lst) '() ]
    [ ( member (car lst) (cdr lst) )
      (list->set (cdr lst)) ]
    [else (cons (car lst) (list->set (cdr lst)))]
   )
)

(define ( a-list lst1 lst2 )
  ( cond
     [ (empty? lst1) '() ]
     [ else
       ( cons ( cons (car lst1) (car lst2) )
            ( a-list (cdr lst1) (cdr lst2) ) ) ]
   )
)

(define ( assoc obj a-lst )
  ( cond
     [ (empty? a-lst ) '() ]
     [ (equal? obj (car (car a-lst)))
       (car a-lst) ]
     [ else
       (assoc obj ( cdr a-lst ) ) ]
  )
)

( 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 ) ) )
   )
)
```

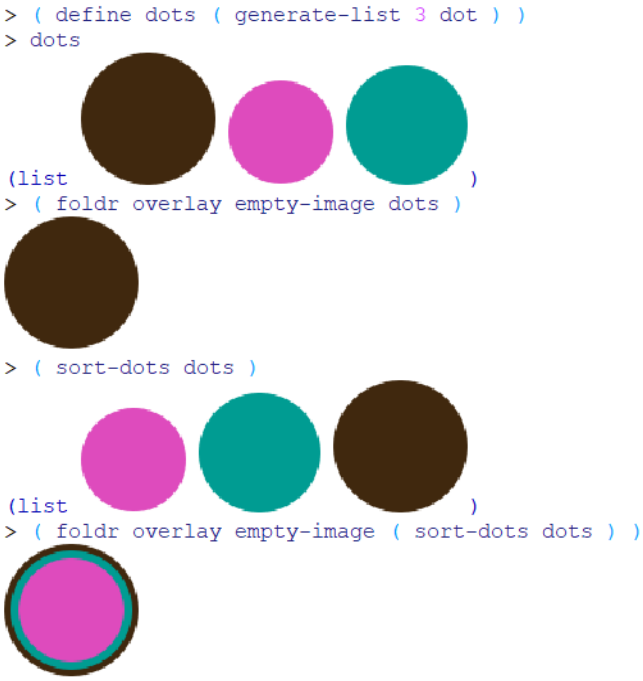
```
> ( define ft1 ( ft '(10 10 10 10 1 1 1 1 9 9 9 2 2 2 8 8 3 3 4 5 6 7 ) ) )
> ft1
'((1 . 4) (2 . 3) (3 . 2) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 2) (9 . 3) (10 . 4))
> ( ft-visualizer ft1)
1:   ****
2:   ***
3:   **
4:   *
5:   *
6:   *
7:   *
8:   **
9:   ***
10:  ****
> ( define ft2 ( ft '( 1 10 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:  *
```

1) count, a-list, list->set
2) (lambda (x) (count x the-list))
3) 2
4) Whatever is used to replace the lambda function would have to return a list that works well with the map function
5) association-list
6) An extra name that gets attached to a variable
7) pair-visualizer
8) The ft-visualizer
9) string-append
10) No it doesn't
11) Its a easy thing to return for the map function that doesn't throw off the code
12) Stem and Leaf Plot
13) Yes
14) I thought it was very interesting how it implement the previous parts of the assignment
15) Do keyword arguments in Racket similar to how they do in python?

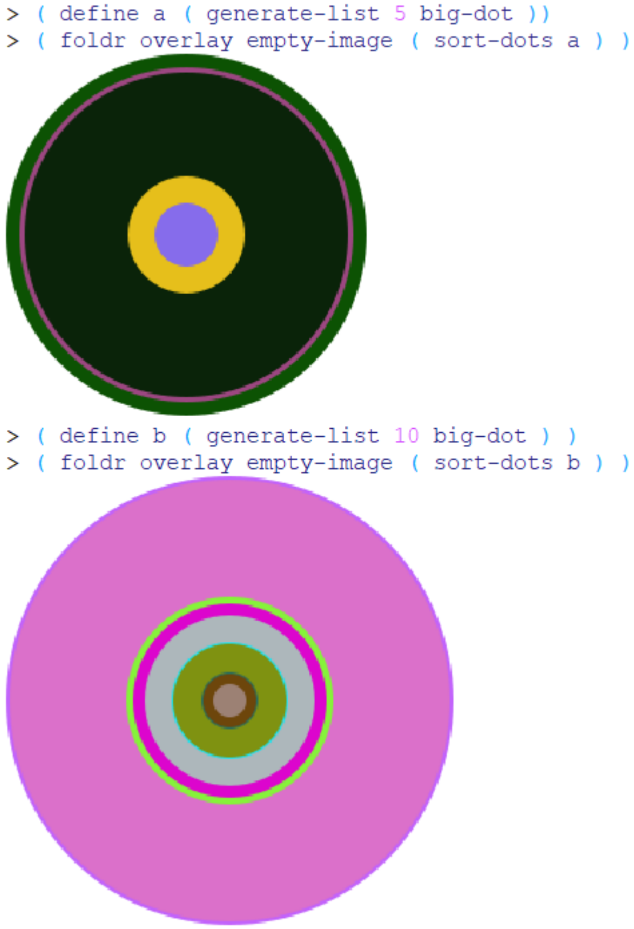
Task 6: Generate List



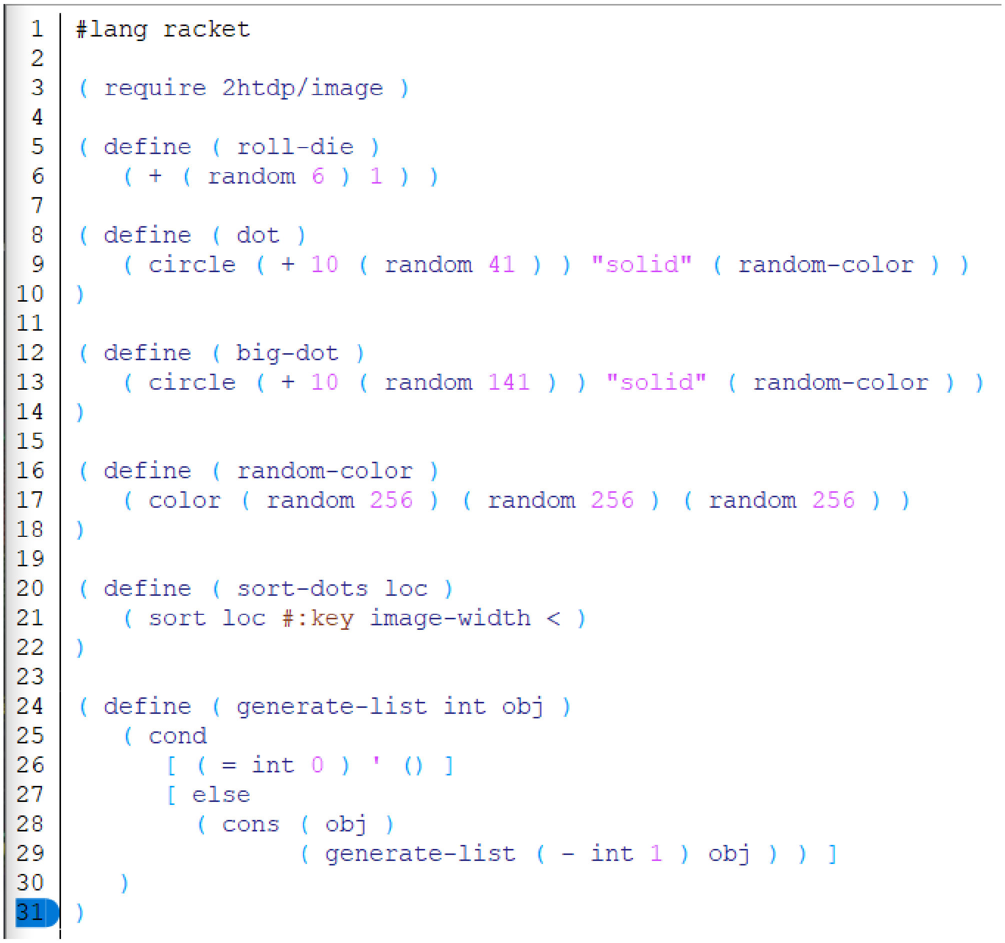
```
'(yellow blue blue red yellow yellow blue yellow yellow yellow blue blue)
> ( generate-list 10 roll-die )
'(2 5 6 2 3 3 6 3 3 4)
> ( generate-list 20 roll-die )
'(4 6 5 3 2 5 3 1 3 6 6 1 6 3 5 1 6 6 6 5)
> ( generate-list 12
( lambda () ( list-ref '( red yellow blue ) ( random 3 ) ) )
)
'(blue blue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

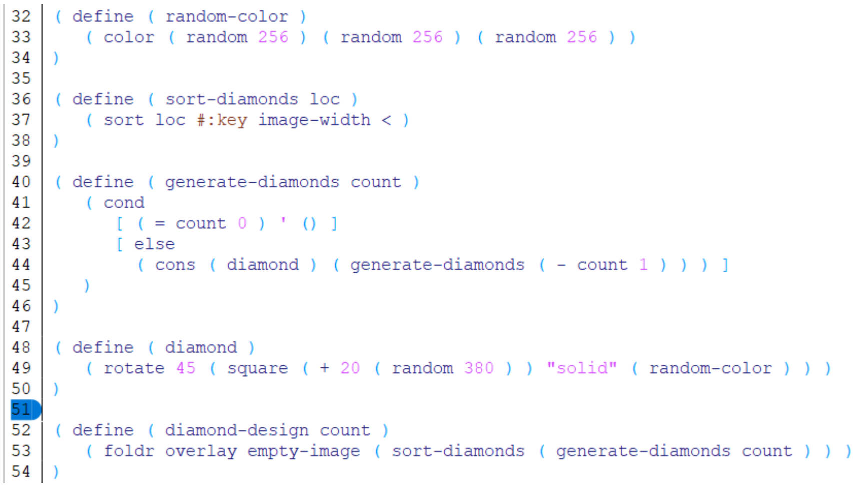


```
> ( define a ( generate-list 5 big-dot ))
> ( foldr overlay empty-image ( sort-dots a ) )
> ( define b ( generate-list 10 big-dot ) )
> ( foldr overlay empty-image ( sort-dots b ) )
```

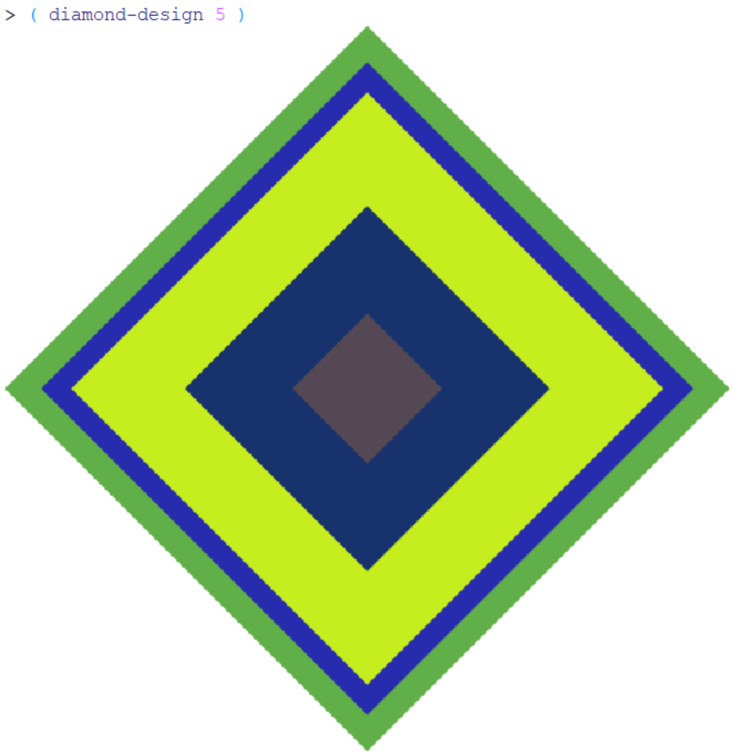


```
#lang racket

( require 2htdp/image )

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

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

( define ( big-dot )
   ( circle ( + 10 ( random 141 ) ) "solid" ( random-color ) )
)

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

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

( define ( generate-list int obj )
   ( cond
      [ ( = int 0 ) ' () ]
      [ else
        ( cons ( obj )
               ( generate-list ( - int 1 ) obj ) ) ]
   )
)
```

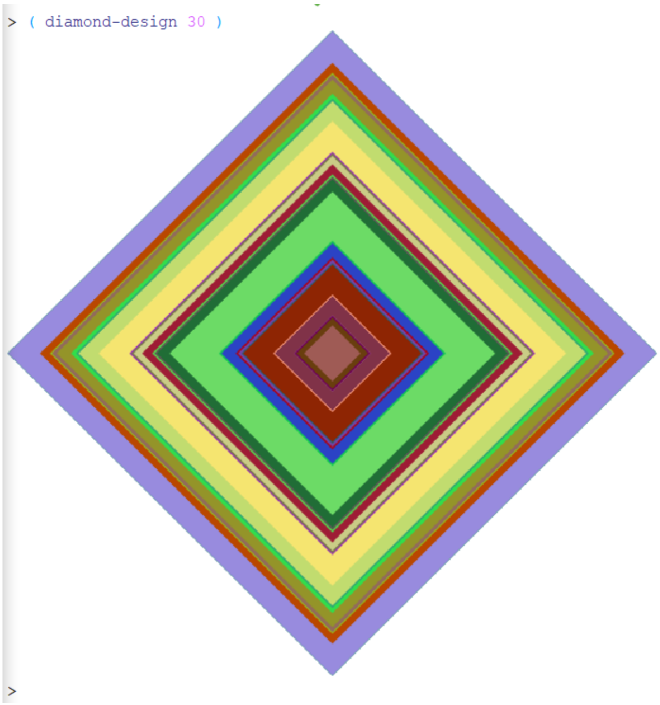
Task 7: The Diamond



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

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

( define ( generate-diamonds count )
   ( cond
      [ ( = count 0 ) ' () ]
      [ else
        ( cons ( diamond ) ( generate-diamonds ( - count 1 ) ) ) ]
   )
)

( define ( diamond )
   ( rotate 45 ( square ( + 20 ( random 380 ) ) "solid" ( random-color ) ) )
)

( define ( diamond-design count )
   ( foldr overlay empty-image ( sort-diamonds ( generate-diamonds count ) ) )
)
```

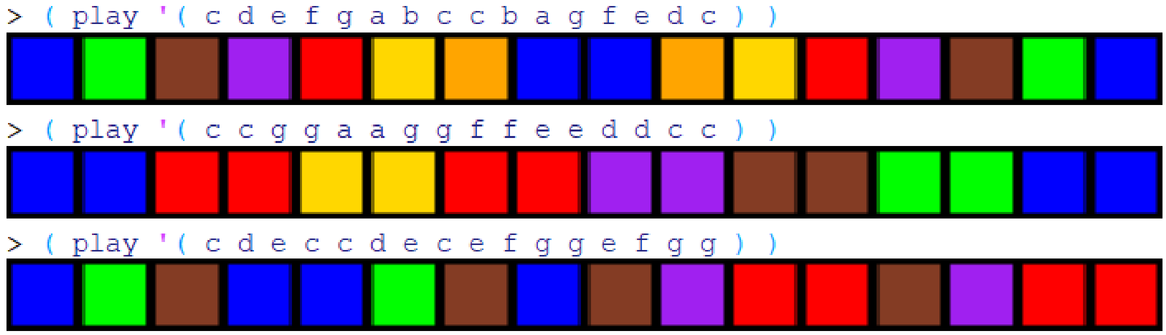


```
> ( diamond-design 5 )
```

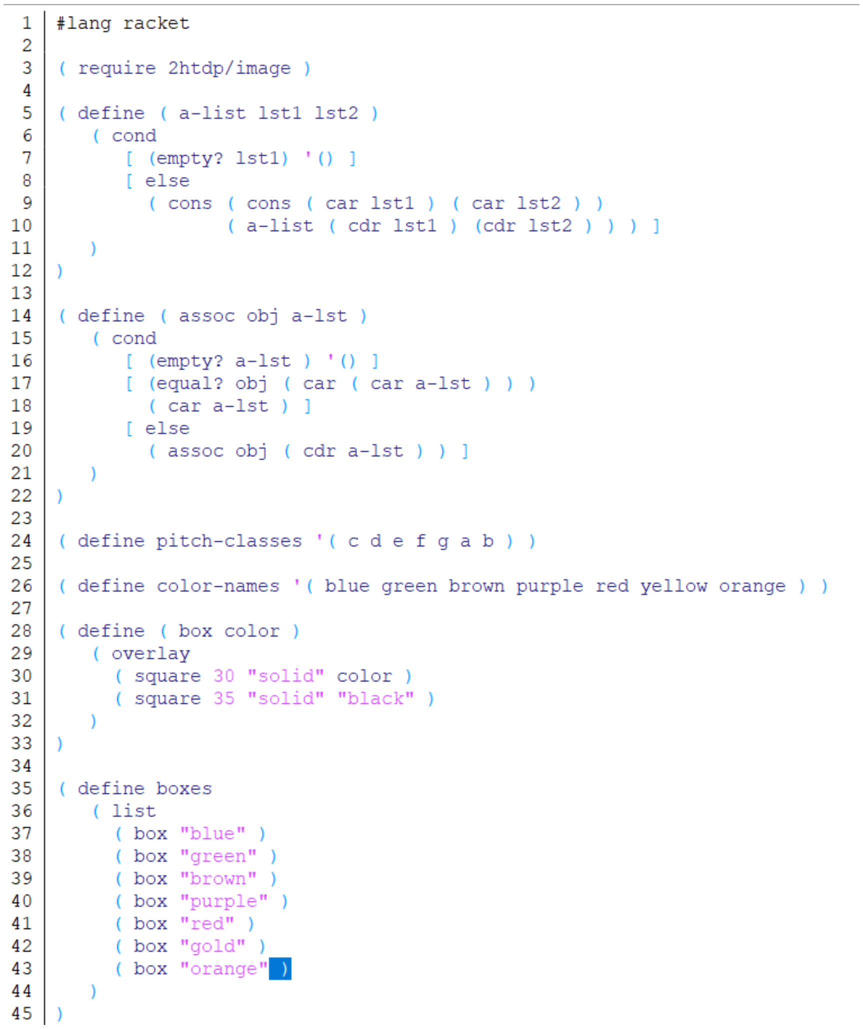


```
> ( diamond-design 30 )
>
```

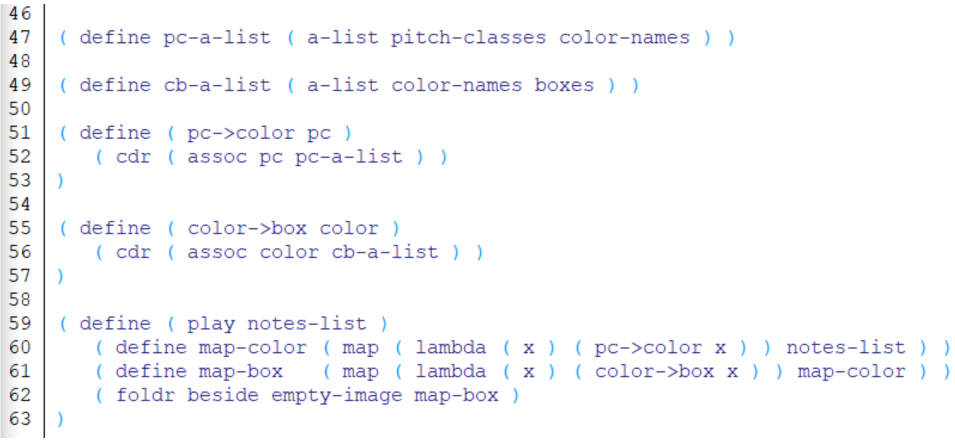
Task 8: Chromesthetic Renderings



```
> ( play '( c d e f g a b c c b a g f e d c ) )
🟦🟩🟫🟪🟥🟨🟧🟦🟦🟧🟨🟥🟪🟫🟩🟦
> ( play '( c c g g a a g g f f e e d d c c ) )
🟦🟦🟥🟥🟨🟨🟥🟥🟪🟪🟫🟫🟩🟩🟦🟦
> ( play '( c d e c c d e c e f g g e f g g ) )
🟦🟩🟫🟦🟦🟩🟫🟦🟫🟪🟥🟥🟫🟪🟥🟥
```

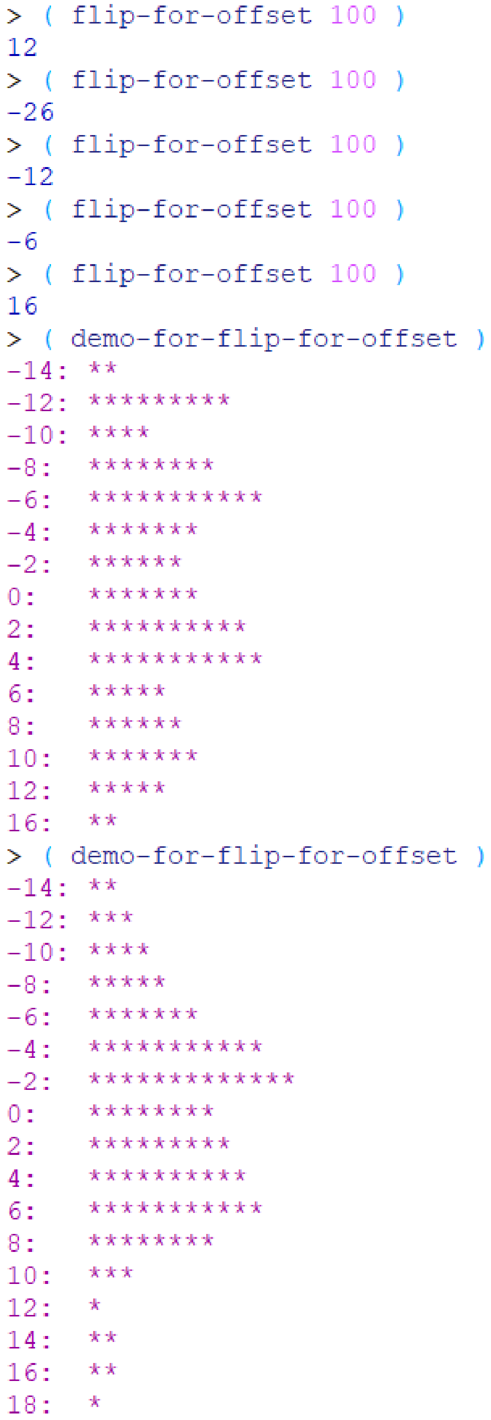


```
#lang racket

( require 2htdp/image )

( define ( a-list lst1 lst2 )
   ( cond
      [ (empty? lst1) '() ]
      [ else
        ( cons ( cons ( car lst1 ) ( car lst2 ) )
               ( a-list ( cdr lst1 ) (cdr lst2 ) ) ) ]
   )
)

( define ( assoc obj a-lst )
   ( cond
      [ (empty? a-lst ) '() ]
      [ (equal? obj ( car ( car a-lst ) ) )
        ( car a-lst ) ]
      [ else
        ( assoc obj ( cdr a-lst ) ) ]
   )
)

( 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" )
   )
)
```

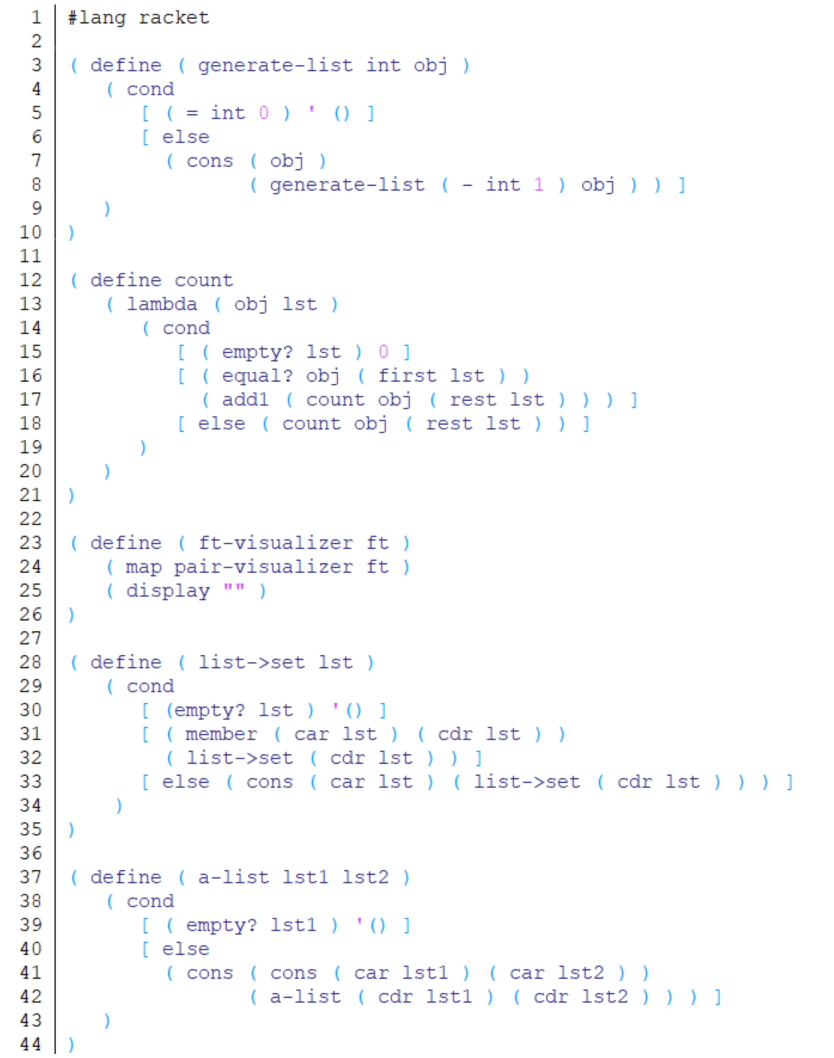


```
( define pc-a-list ( a-list pitch-classes color-names ) )

( 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 ) )
)

( define ( play notes-list )
   ( define map-color ( map ( lambda ( x ) ( pc->color x ) ) notes-list ) )
   ( define map-box   ( map ( lambda ( x ) ( color->box x ) ) map-color ) )
   ( foldr beside empty-image map-box )
)
```

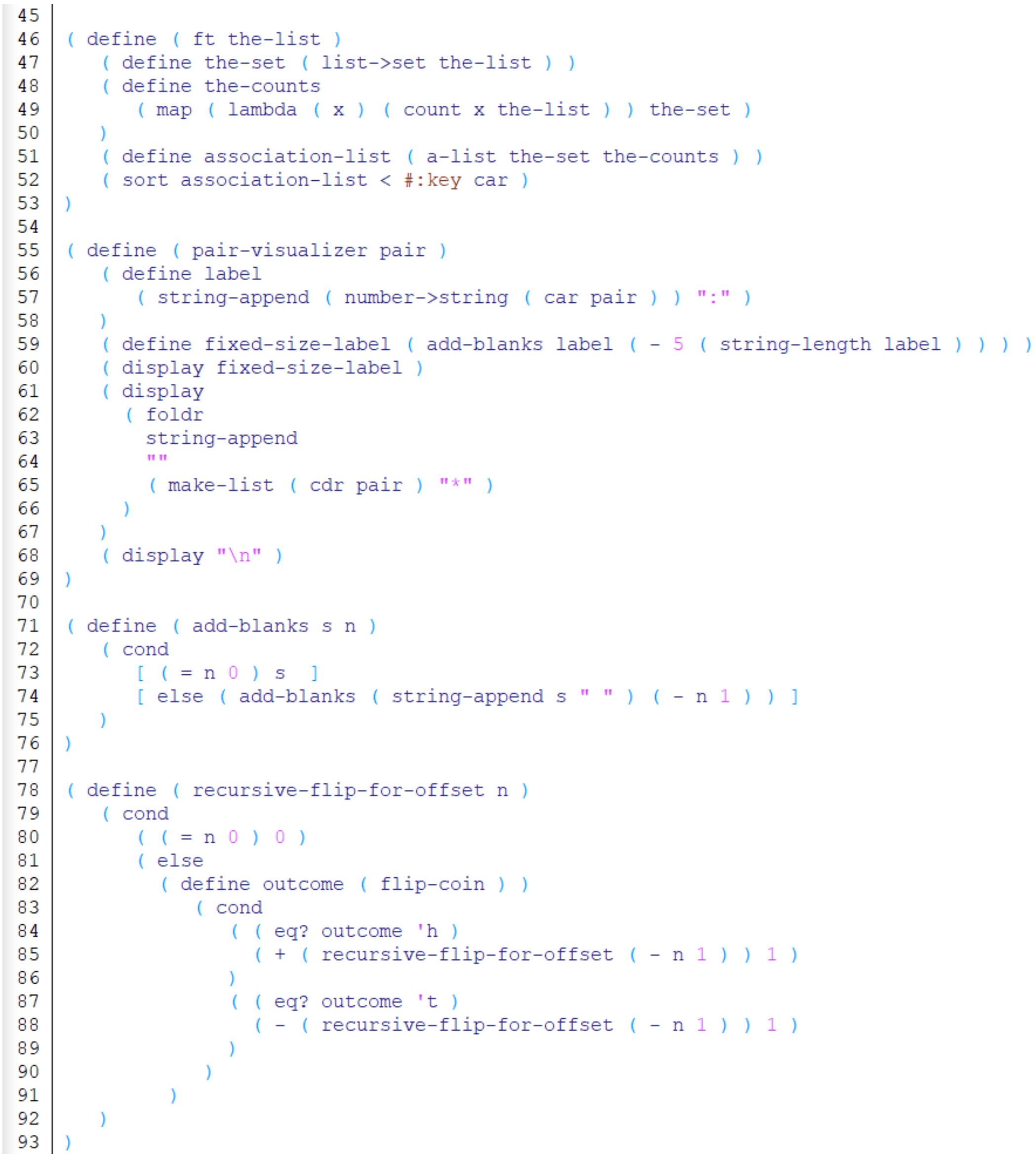
Task 9: Transformation of a Recursive Sampler



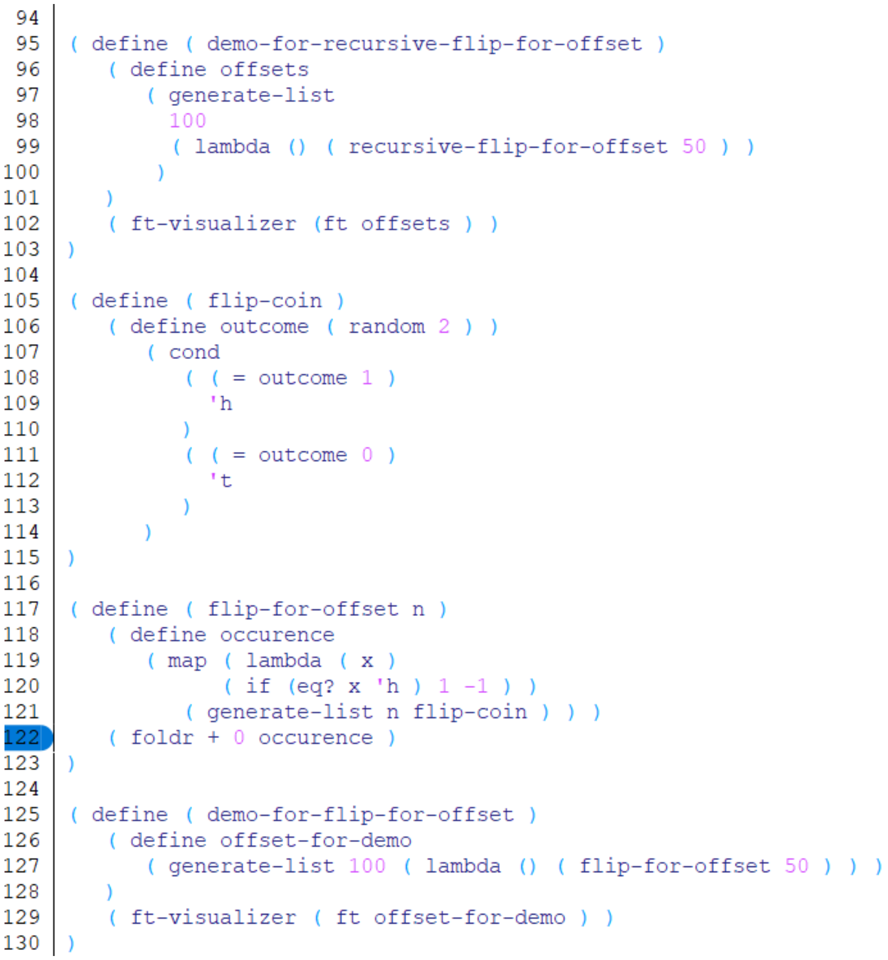
```
> ( flip-for-offset 100 )
12
> ( flip-for-offset 100 )
-26
> ( flip-for-offset 100 )
-12
> ( flip-for-offset 100 )
-6
> ( flip-for-offset 100 )
16
> ( demo-for-flip-for-offset )
-14: **
-12: *********
-10: ****
-8:  ********
-6:  ***********
-4:  *******
-2:  ******
0:   *******
2:   **********
4:   ***********
6:   *****
8:   ******
10:  *******
12:  *****
16:  **
> ( demo-for-flip-for-offset )
-14: **
-12: ***
-10: ****
-8:  *****
-6:  *******
-4:  ***********
-2:  *************
0:   ********
2:   *********
4:   **********
6:   ***********
8:   ********
10:  ***
12:  *
14:  **
16:  **
18:  *
```



```
#lang racket

( define ( generate-list int obj )
   ( cond
      [ ( = int 0 ) ' () ]
      [ else
        ( cons ( obj )
               ( generate-list ( - int 1 ) obj ) ) ]
   )
)

( define count
   ( lambda ( obj lst )
      ( cond
         [ ( empty? lst ) 0 ]
         [ ( equal? obj ( first lst ) )
           ( add1 ( count obj ( rest lst ) ) ) ]
         [ else ( count obj ( rest lst ) ) ]
      )
   )
)

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

( define ( list->set lst )
   ( cond
      [ (empty? lst ) '() ]
      [ ( member ( car lst ) ( cdr lst ) )
        ( list->set ( cdr lst ) ) ]
      [ else ( cons ( car lst ) ( list->set ( cdr lst ) ) ) ]
    )
)

( define ( a-list lst1 lst2 )
   ( cond
      [ ( empty? lst1 ) '() ]
      [ else
        ( cons ( cons ( car lst1 ) ( car lst2 ) )
               ( a-list ( cdr lst1 ) ( cdr lst2 ) ) ) ]
   )
)
```

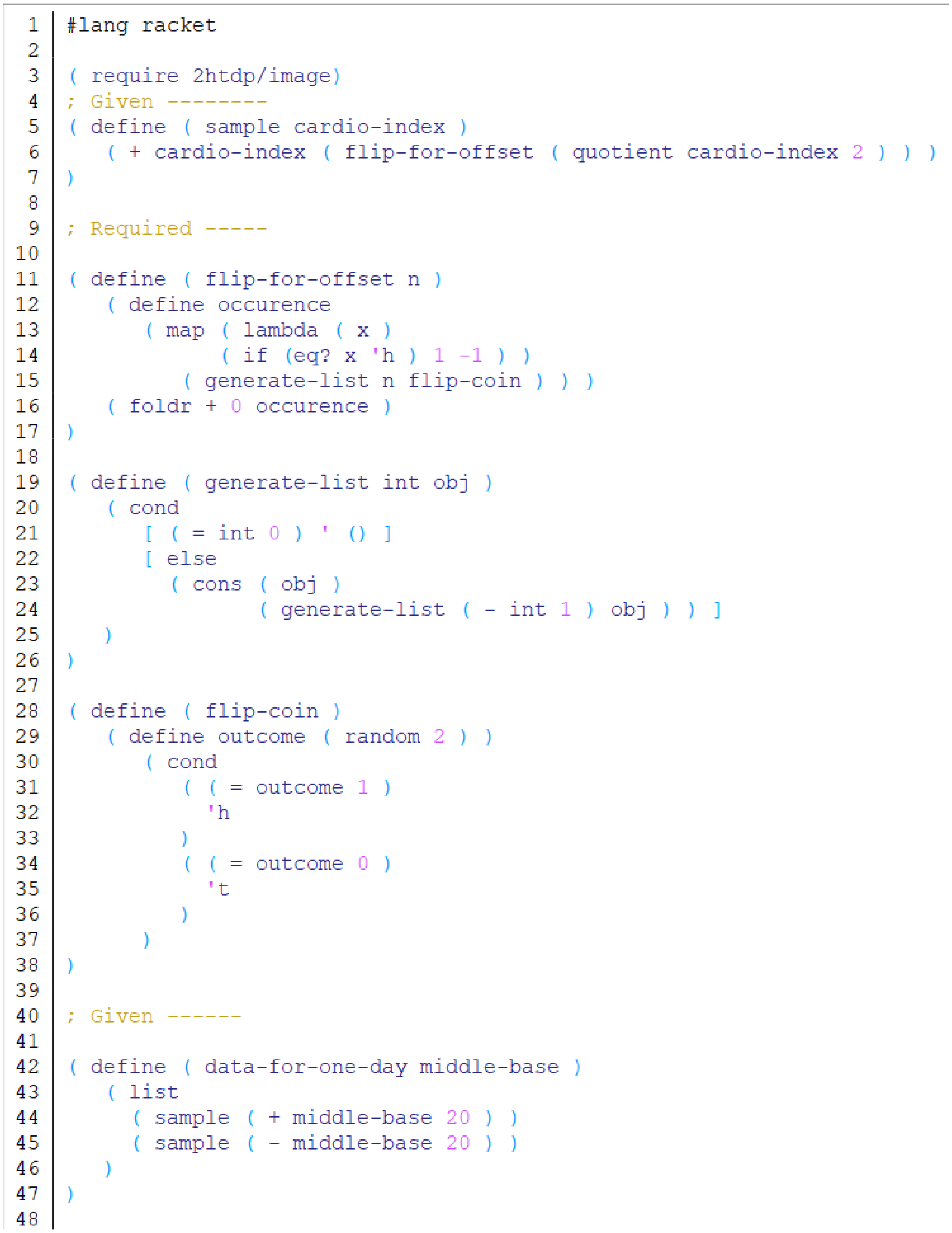


```

( 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 ( 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 ) ) ]
   )
)

( define ( recursive-flip-for-offset n )
   ( cond
      ( ( = n 0 ) 0 )
      ( else
        ( define outcome ( flip-coin ) )
           ( cond
              ( ( eq? outcome 'h )
                ( + ( recursive-flip-for-offset ( - n 1 ) ) 1 )
              )
              ( ( eq? outcome 't )
                ( - ( recursive-flip-for-offset ( - n 1 ) ) 1 )
              )
            )
         )
   )
)
```

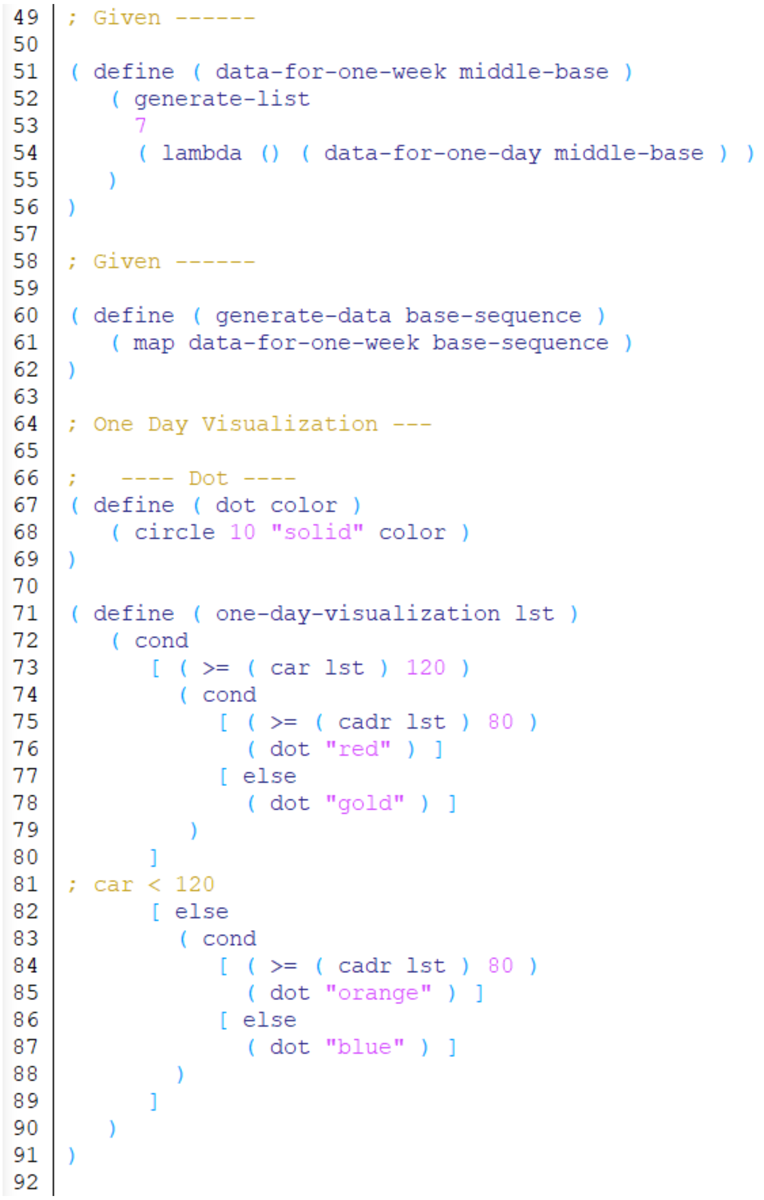


```scheme

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

( define ( flip-coin )
   ( define outcome ( random 2 ) )
      ( cond
         ( ( = outcome 1 )
           'h
         )
         ( ( = outcome 0 )
           'h
         )
      )
)

( define ( flip-for-offset n )
   ( define occurence
      ( map ( lambda ( x )
            ( if (eq? x 'h ) 1 -1 ) )
         ( generate-list n flip-coin ) ) )
   ( foldr + 0 occurence )
)

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

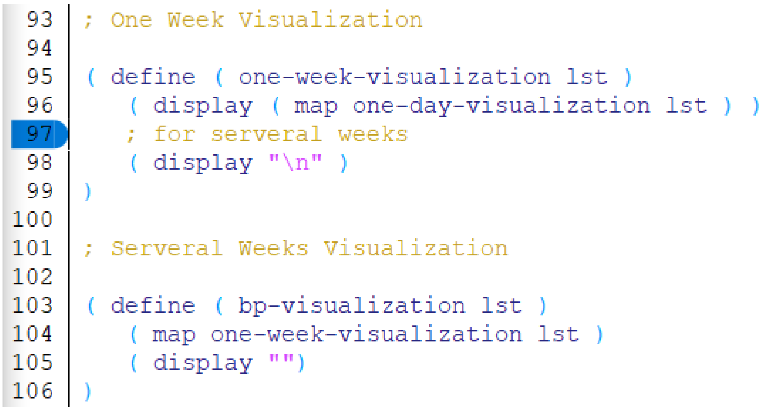
Task 10: Blood Pressure Trend Visualize



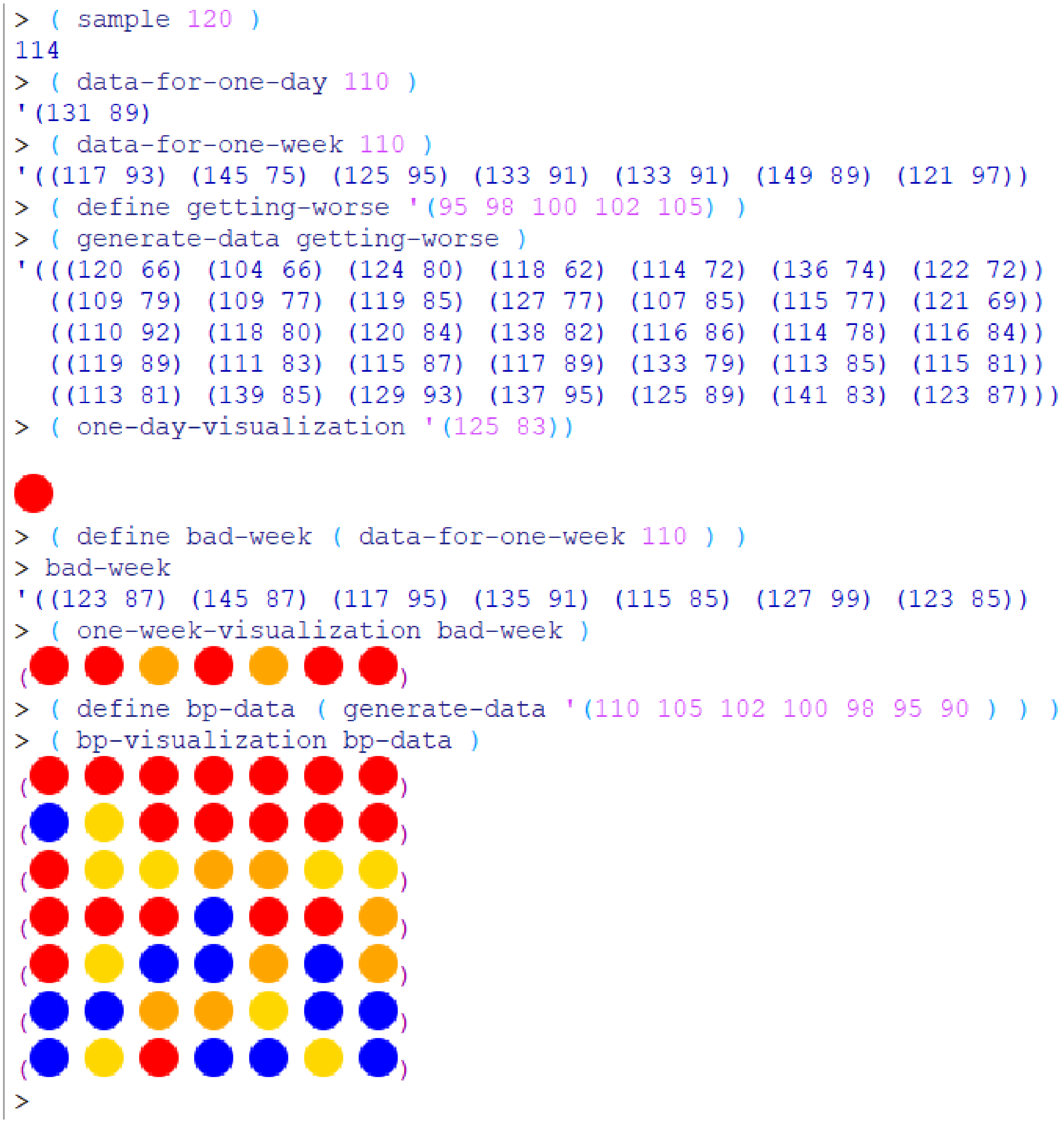
```racket
#lang racket

( require 2htdp/image)
; Given --------
( define ( sample cardio-index )
   ( + cardio-index ( flip-for-offset ( quotient cardio-index 2 ) ) )
)

; Required -----

( define ( flip-for-offset n )
   ( define occurence
      ( map ( lambda ( x )
            ( if (eq? x 'h ) 1 -1 ) )
         ( generate-list n flip-coin ) ) )
   ( foldr + 0 occurence )
)

( define ( generate-list int obj )
   ( cond
      [ ( = int 0 ) ' () ]
      [ else
        ( cons ( obj )
               ( generate-list ( - int 1 ) obj ) ) ]
   )
)

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

; Given ------

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

```



```
; Given ------

( define ( data-for-one-week middle-base )
   ( generate-list
     7
     ( lambda () ( data-for-one-day middle-base ) )
   )
)

; Given ------

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

; One Day Visualization ---

;   ---- Dot ----
( define ( dot color )
   ( circle 10 "solid" color )
)

( define ( one-day-visualization lst )
   ( cond
      [ ( >= ( car lst ) 120 )
        ( cond
           [ ( >= ( cadr lst ) 80 )
             ( dot "red" ) ]
           [ else
             ( dot "gold" ) ]
         )
      ]
; car < 120
      [ else
        ( cond
           [ ( >= ( cadr lst ) 80 )
             ( dot "orange" ) ]
           [ else
             ( dot "blue" ) ]
        )
      ]
   )
)

```



```
; One Week Visualization

( define ( one-week-visualization lst )
   ( display ( map one-day-visualization lst ) )
   ; for serveral weeks
   ( display "\n" )
)

; Serveral Weeks Visualization

( define ( bp-visualization lst )
   ( map one-week-visualization lst )
   ( display "")
)
```



```
> ( sample 120 )
114
> ( data-for-one-day 110 )
'(131 89)
> ( data-for-one-week 110 )
'((117 93) (145 75) (125 95) (133 91) (133 91) (149 89) (121 97))
> ( define getting-worse '(95 98 100 102 105) )
> ( generate-data getting-worse )
'((((120 66) (104 66) (124 80) (118 62) (114 72) (136 74) (122 72))
  ((109 79) (109 77) (119 85) (127 77) (107 85) (115 77) (121 69))
  ((110 92) (118 80) (120 84) (138 82) (116 86) (114 78) (116 84))
  ((119 89) (111 83) (115 87) (117 89) (133 79) (113 85) (115 81))
  ((113 81) (139 85) (129 93) (137 95) (125 89) (141 83) (123 87)))
> ( one-day-visualization '(125 83))

🔴
> ( define bad-week ( data-for-one-week 110 ) )
> bad-week
'((123 87) (145 87) (117 95) (135 91) (115 85) (127 99) (123 85))
> ( one-week-visualization bad-week )
(🔴 🔴 🟠 🔴 🟠 🔴 🔴)
> ( define bp-data ( generate-data '(110 105 102 100 98 95 90 ) ) )
> ( bp-visualization bp-data )
(🔴 🔴 🔴 🔴 🔴 🔴 🔴)
(🔵 🟡 🔴 🔴 🔴 🔴 🔴)
(🔴 🟡 🟡 🟠 🟠 🟡 🟡)
(🔴 🔴 🔴 🔵 🔴 🔴 🟠)
(🔴 🟡 🔵 🔵 🟠 🔵 🟠)
(🔵 🔵 🟠 🟠 🟡 🔵 🔵)
(🔵 🟡 🔴 🔵 🔵 🟡 🔵)
>
```