Nathaniel Wolf

Racket assignment #4: Recursive List Processing and High Order Functions

3/30/2022

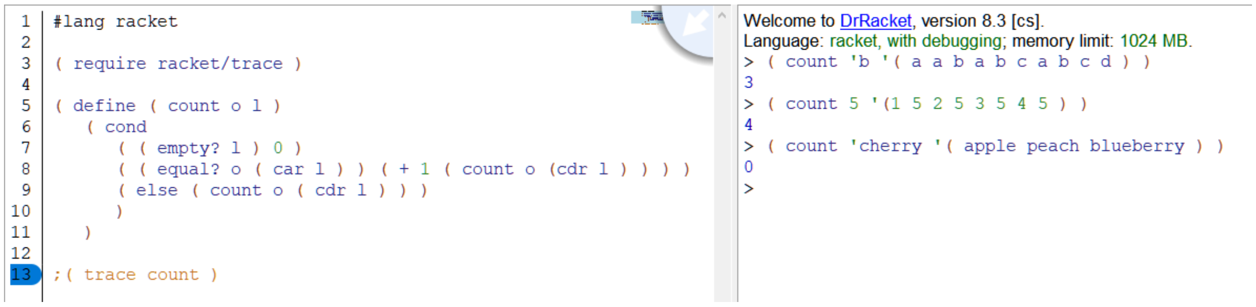
CSC 344

Learning Abstract

This assignment involves making use of high order functions to perform recursive Racket list processing. The first few demos contain boiler plate code for said high order functions which get progressively used more with each assignment with the final assignment making use of the majority of the priorly defined high order functions, as well as simple code provided by the professor. Question 5 contains analysis of predefined code including identify high order functions and how functions such as map works in Racket. The reason for this is not so much repetition, but to gain an understanding of how these individual functions work together .

Problem 1

Code and Demo

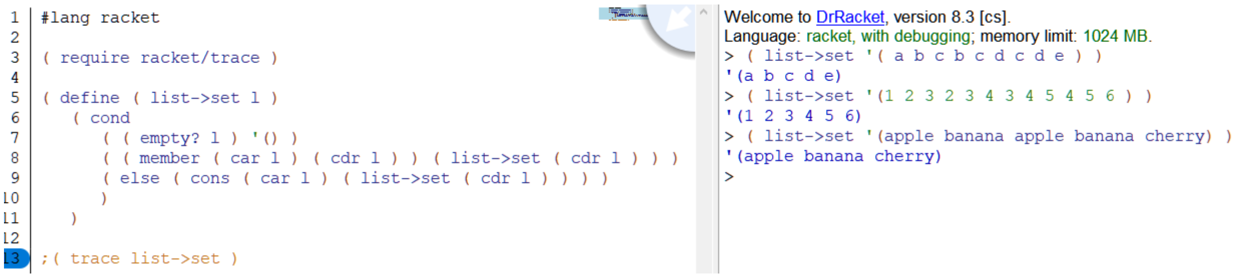


```
#lang racket

( require racket/trace )

( define ( count o l )
   ( cond
      ( ( empty? l ) 0 )
      ( ( equal? o ( car l ) ) ( + 1 ( count o (cdr l ) ) ) )
      ( else ( count o ( cdr l ) ) )
      )
   )

;( trace count )
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 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
>
```

Code and Demo

Problem 2

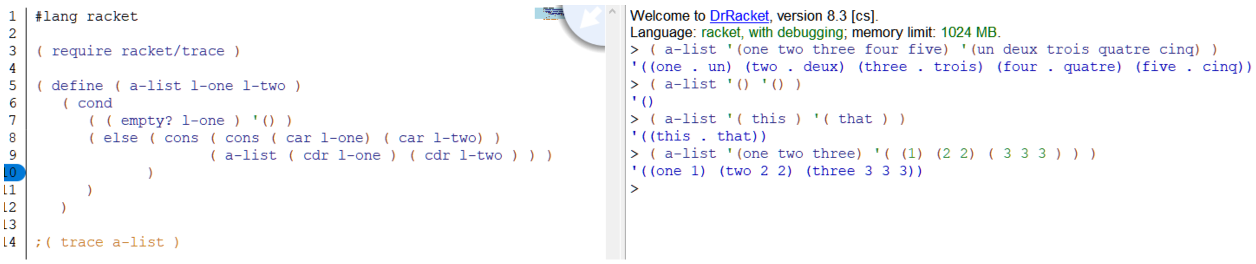


```
#lang racket

( require racket/trace )

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

;( trace list->set )
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 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)
>
```

Problem 3

Code and Demo



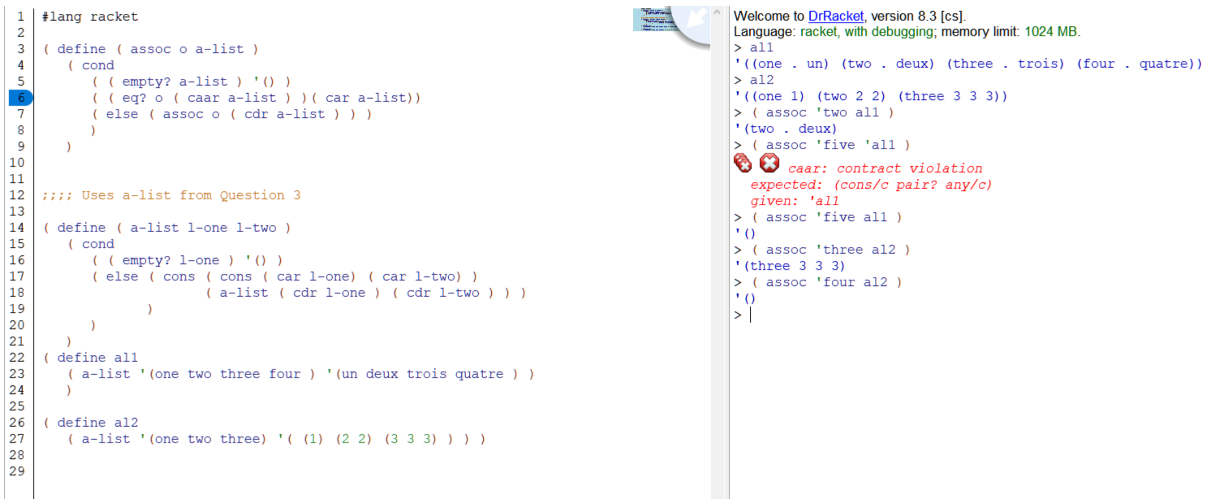
```
#lang racket

( require racket/trace )

( define ( a-list l-one l-two )
   ( cond
      ( ( empty? l-one ) '() )
      ( else ( cons ( cons ( car l-one) ( car l-two) )
                    ( a-list ( cdr l-one ) ( cdr l-two ) ) )
             )
      )
   )

;( trace a-list )
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 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))
>
```

Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344

Problem 4

Code and Demo

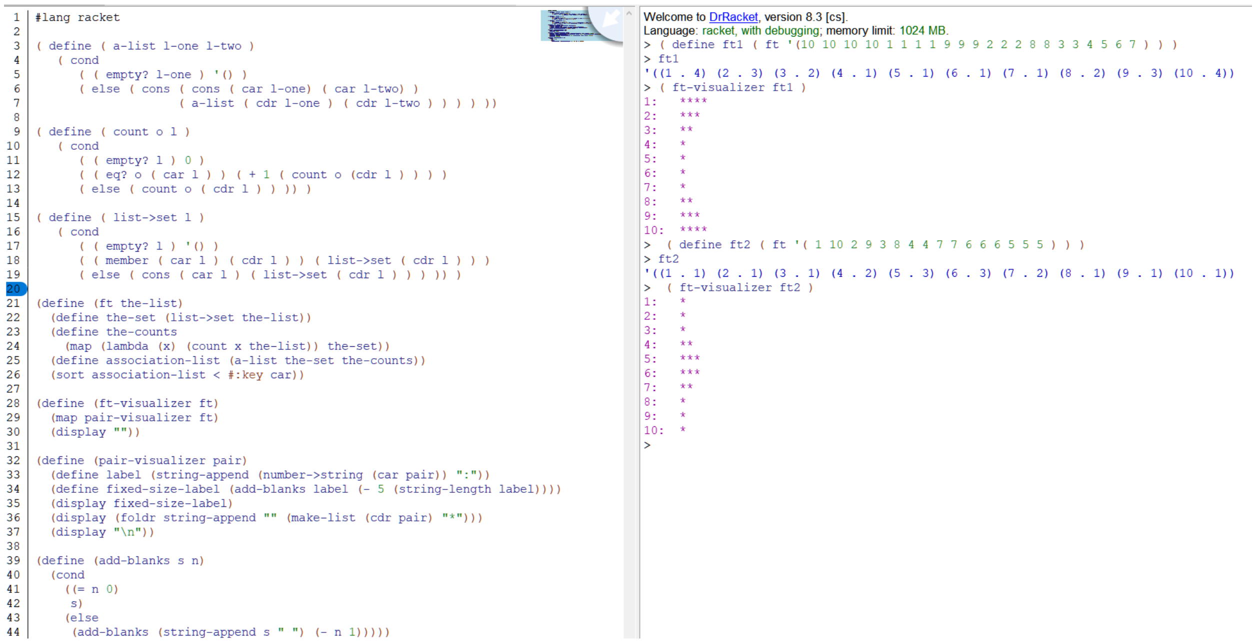


```racket
#lang racket

( define ( assoc o a-list )
   ( cond
      ( ( empty? a-list ) '() )
      ( ( eq? o ( caar a-list ) )( car a-list))
      ( else ( assoc o ( cdr a-list ) ) )
      )
   )


;;;; Uses a-list from Question 3

( define ( a-list l-one l-two )
   ( cond
      ( ( empty? l-one ) '() )
      ( else ( cons ( cons ( car l-one) ( car l-two) )
                    ( a-list ( cdr l-one ) ( cdr l-two ) ) )
             )
      )
   )
( 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) ) ) )
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
> al1
'((one . un) (two . deux) (three . trois) (four . quatre))
> al2
'((one 1) (two 2 2) (three 3 3 3))
> ( assoc 'two al1 )
'(two . deux)
> ( assoc 'five 'al1 )
caar: contract violation
  expected: (cons/c pair? any/c)
  given: 'al1
> ( assoc 'five al1 )
'()
> ( assoc 'three al2 )
'(three 3 3 3)
> ( assoc 'four al2 )
'()
> 
```

Problem 5

Demo Replication



```racket
#lang racket

( define ( a-list l-one l-two )
   ( cond
      ( ( empty? l-one ) '() )
      ( else ( cons ( cons ( car l-one) ( car l-two) )
                    ( a-list ( cdr l-one ) ( cdr l-two ) ) ) ) ))

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

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

(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)))))
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 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:  *
>
```

Short Answer

1) Count, a-list, list->set
2) (lambda (x) (count x the-list))
3) It takes two parameters
4) The primary challenge would be the need to return a set based on two parameters.
5) Association-list (assoc-list)

6) < #:keyword arg-expr > Allows for assigning of tangible names to arguments which can then be called into a function.
7) Pair-visualizer
8) Ft-visualizer requires les functional arguments than ft.
9) Make-list
10) No code is boilerplate albeit recursive logic.
11) It would impair on the visual integrity of the output with an excessive amount of whit spaces or ‘()s
12) The DS in question is a steam leaf plot. Stems are shown greatest to least and leaves are shown as asterisks to indicate how many leaves are in said numerical category.
13) No, function is higher order
14) The use of asterisks to indicate “tallys” in each category as well as the formatting in a human readable table format.
15) Using the trace import functionality to assist as needed, For ft1 and ft2, determine how many runs through add-blanks were processed for each numerical category. How many runs did it take for each list to process in total?

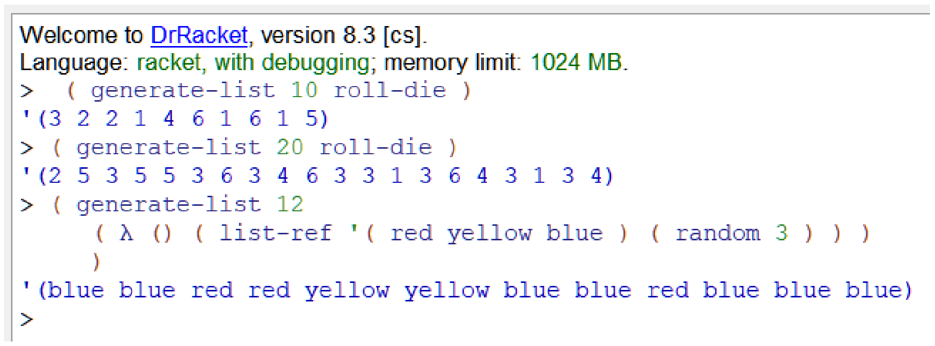
Problem 6

Code

```
#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 n o )
   ( cond
      ( ( eq? n 0 ) '() )
      ( else
        ( cons ( o ) ( generate-list ( - n 1 ) o ) )
        )
      )
   )
```

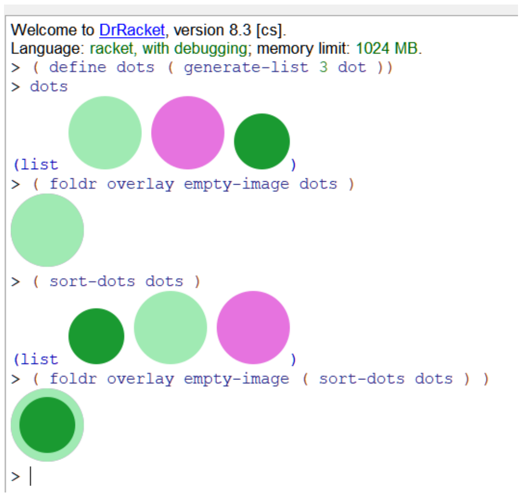
Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344

Demo 1



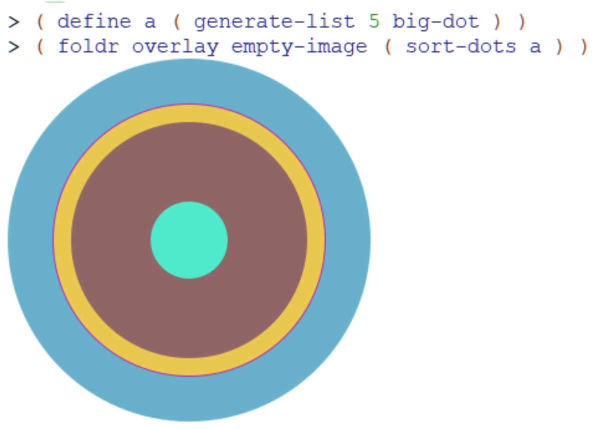
```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
>  ( generate-list 10 roll-die )
'(3 2 2 1 4 6 1 6 1 5)
> ( generate-list 20 roll-die )
'(2 5 3 5 5 3 6 3 4 6 3 3 1 3 6 4 3 1 3 4)
> ( generate-list 12
     ( λ () ( list-ref '( red yellow blue ) ( random 3 ) ) )
     )
'(blue blue red red yellow yellow blue blue red blue blue blue)
>
```

Demo 2

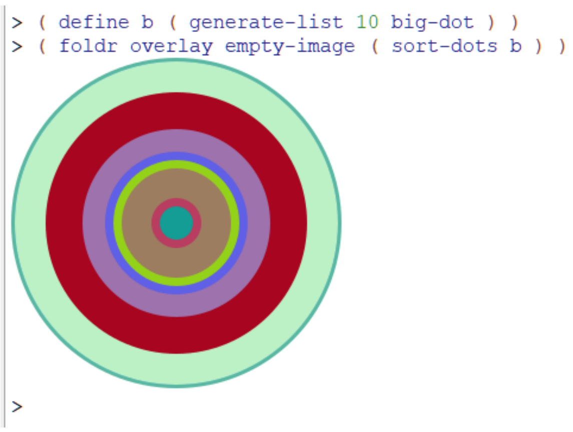


```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
> ( define dots ( generate-list 3 dot ))
> dots
(list                )
> ( foldr overlay empty-image dots )
> ( sort-dots dots )
(list                )
> ( foldr overlay empty-image ( sort-dots dots ) )
> |
```

Demo 3



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

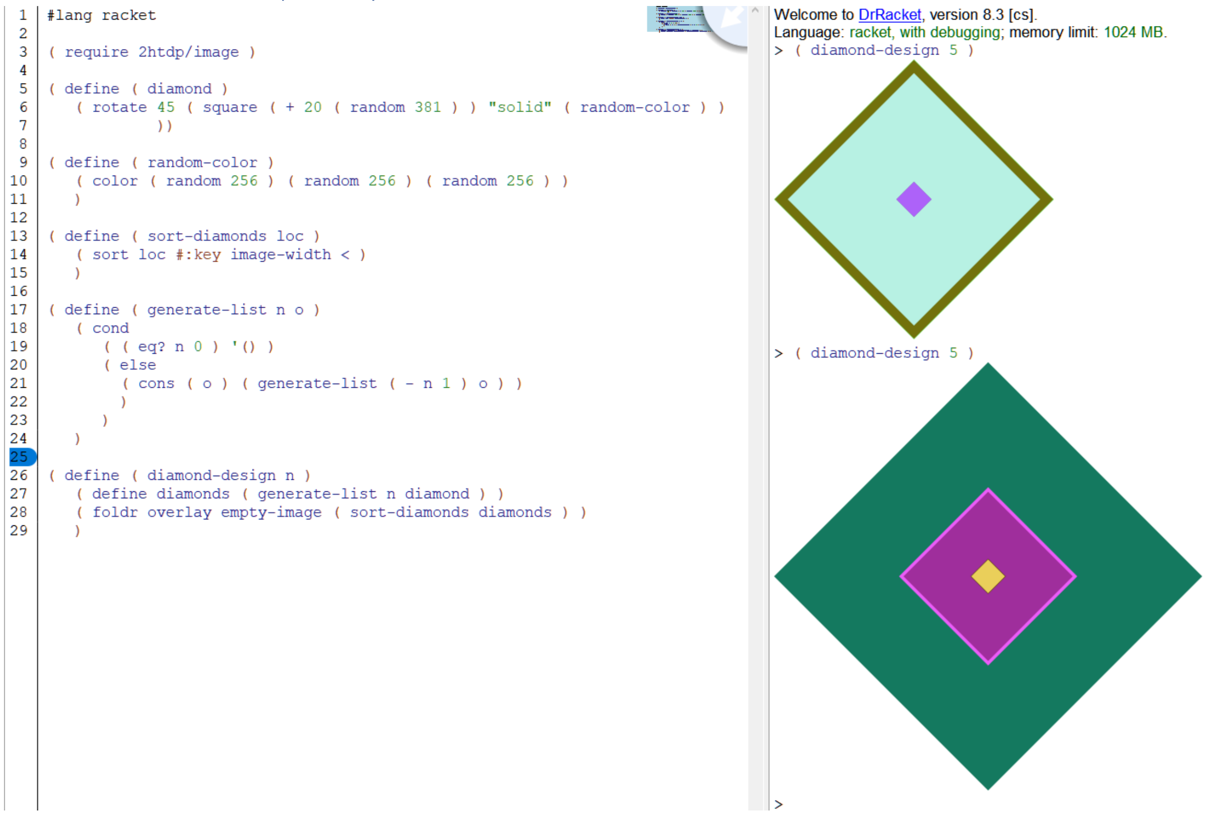


```
> ( define b ( generate-list 10 big-dot ) )
> ( foldr overlay empty-image ( sort-dots b ) )
>
```

Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344

Problem 7

Code, Demo 1 and 2 respectively

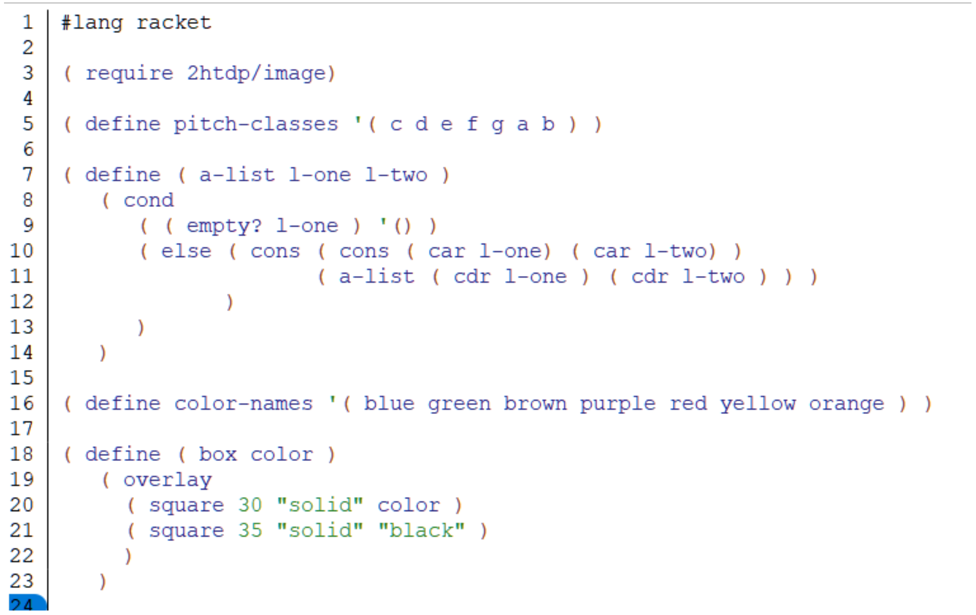


```
#lang racket

( require 2htdp/image )

( 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 < )
    )

( define ( generate-list n o )
    ( cond
       ( ( eq? n 0 ) '() )
       ( else
         ( cons ( o ) ( generate-list ( - n 1 ) o ) )
         )
       )
    )

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

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
> ( diamond-design 5 )
> ( diamond-design 5 )
>
```

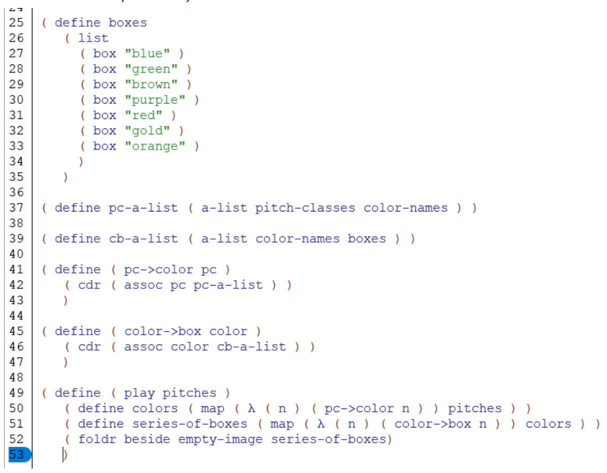
Problem 8

The Code



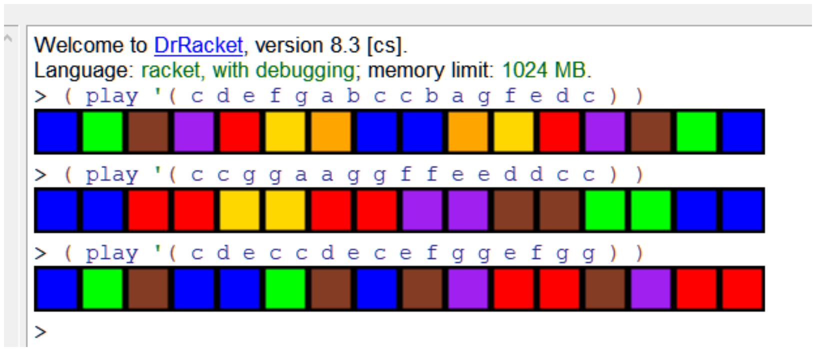
```
#lang racket

( require 2htdp/image)

( define pitch-classes '( c d e f g a b ) )

( define ( a-list l-one l-two )
   ( cond
      ( ( empty? l-one ) '() )
      ( else ( cons ( cons ( car l-one) ( car l-two) )
                     ( a-list ( cdr l-one ) ( cdr l-two ) ) ) )
              )
      )
   )

( define color-names '( blue green brown purple red yellow orange ) )

( define ( box color )
   ( overlay
     ( square 30 "solid" color )
     ( square 35 "solid" "black" )
     )
   )
```

The Code (cont.)



```
( 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 pitches )
   ( define colors ( map ( λ ( n ) ( pc->color n ) ) pitches ) )
   ( define series-of-boxes ( map ( λ ( n ) ( color->box n ) ) colors ) )
   ( foldr beside empty-image series-of-boxes)
   )
```

Demo

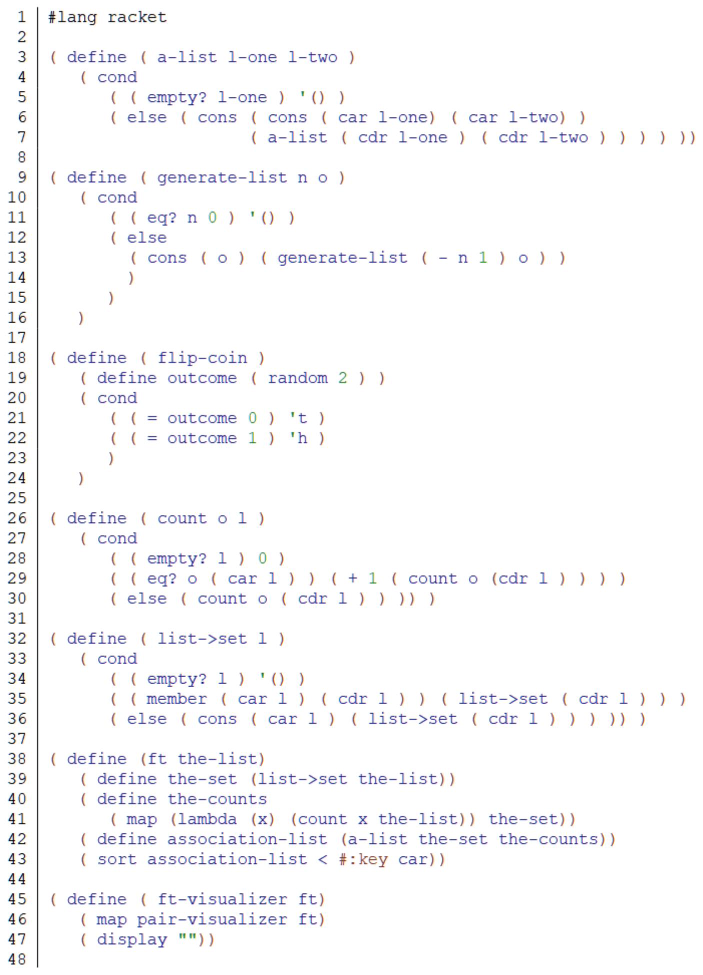


```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
> ( play '( c d e f g a b c c b a g f e d c ) )
> ( play '( c c g g a a g g f f e e d d c c ) )
> ( play '( c d e c c d e c e f g g e f g g ) )
>
```

Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344

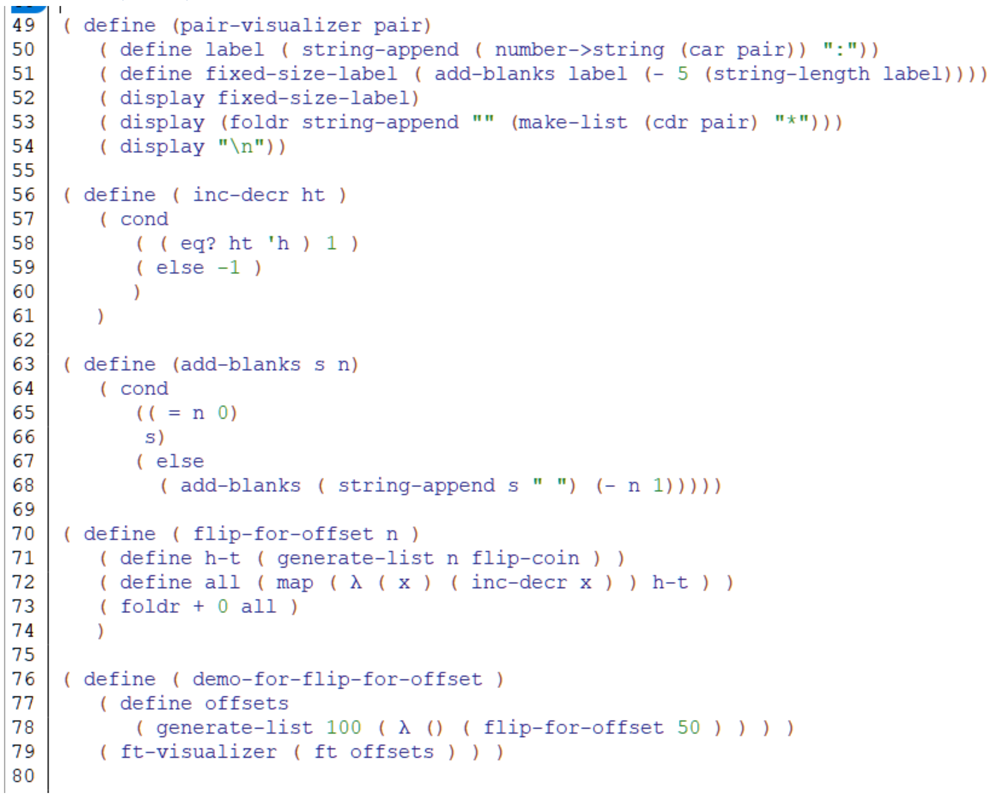
Problem 9

The Code



```
#lang racket

( define ( a-list l-one l-two )
   ( cond
      ( ( empty? l-one ) '() )
      ( else ( cons ( cons ( car l-one) ( car l-two) )
                    ( a-list ( cdr l-one ) ( cdr l-two ) ) ) ) ))

( define ( generate-list n o )
   ( cond
      ( ( eq? n 0 ) '() )
      ( else
        ( cons ( o ) ( generate-list ( - n 1 ) o ) )
        )
      )
   )

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

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

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

( 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 ""))

```

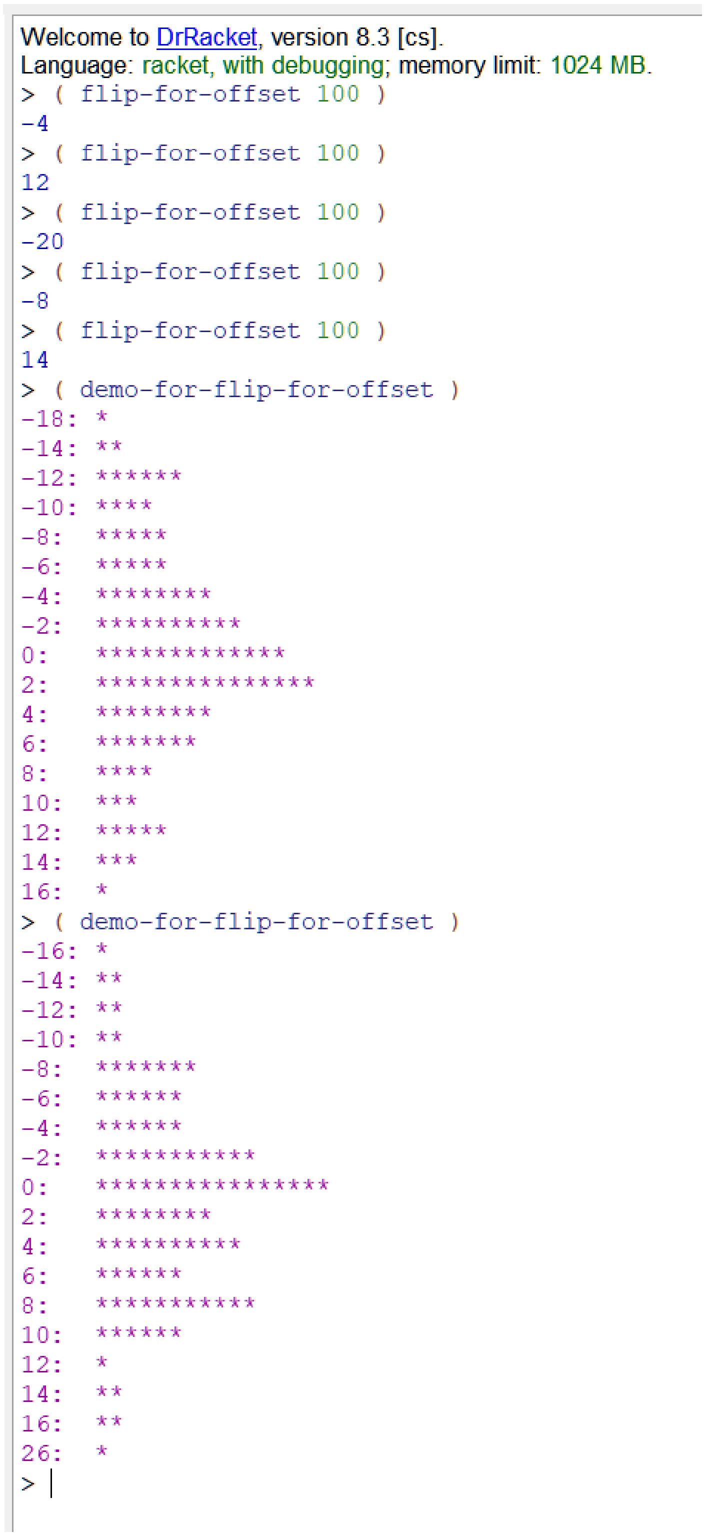
The Code (cont.)



```
( 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 ( inc-decr ht )
   ( cond
      ( ( eq? ht 'h ) 1 )
      ( else -1 )
      )
   )

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

( define ( flip-for-offset n )
   ( define h-t ( generate-list n flip-coin ) )
   ( define all ( map ( λ ( x ) ( inc-decr x ) ) h-t ) )
   ( foldr + 0 all )
   )

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

```

Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344

The Demo

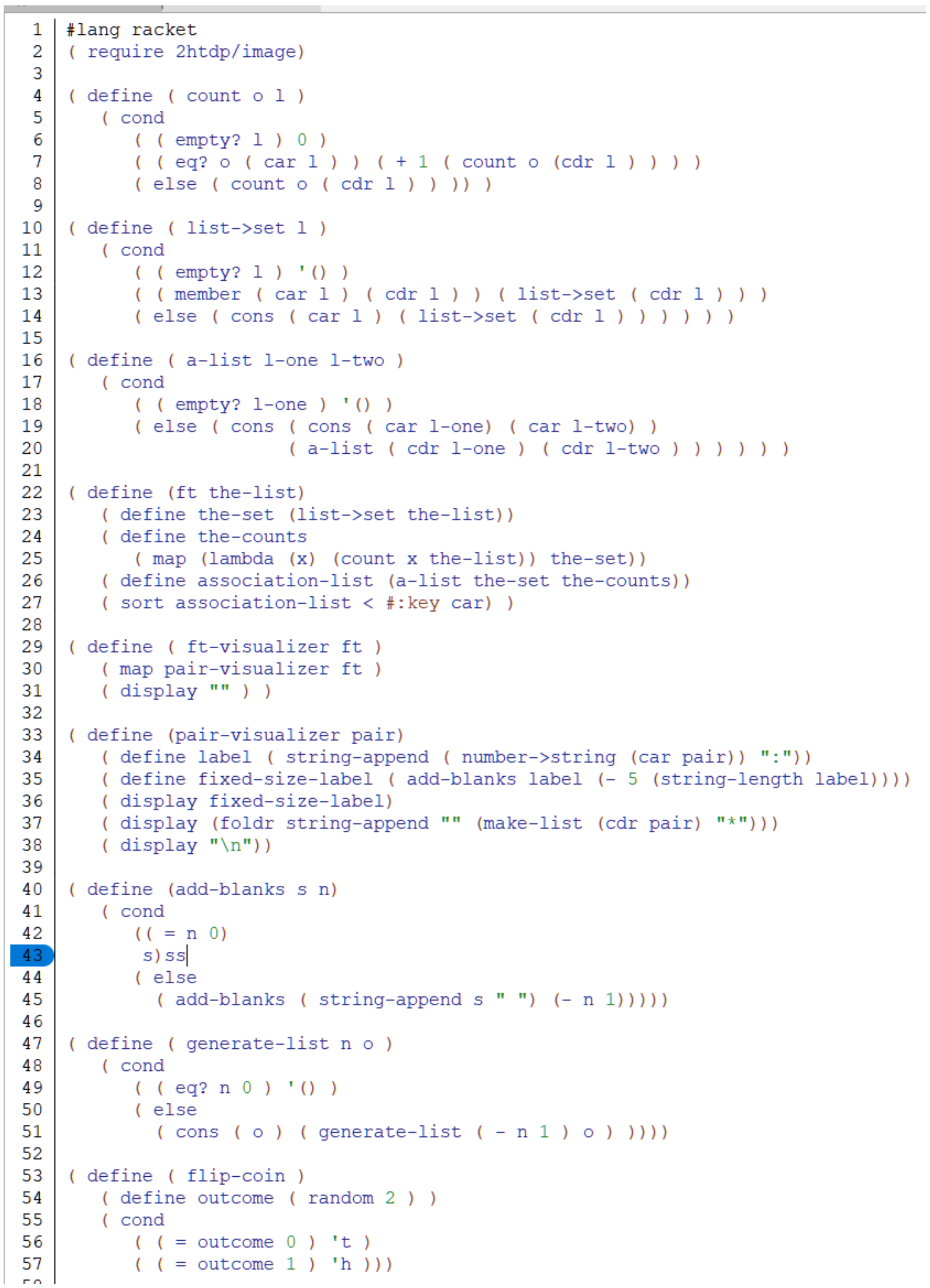


```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
> ( flip-for-offset 100 )
-4
> ( flip-for-offset 100 )
12
> ( flip-for-offset 100 )
-20
> ( flip-for-offset 100 )
-8
> ( flip-for-offset 100 )
14
> ( demo-for-flip-for-offset )
-18: *
-14: **
-12: ******
-10: ****
-8:  *****
-6:  *****
-4:  ********
-2:  **********
0:   *************
2:   ***************
4:   ********
6:   *******
8:   ****
10:  ***
12:  *****
14:  ***
16:  *
> ( demo-for-flip-for-offset )
-16: *
-14: **
-12: **
-10: **
-8:  *******
-6:  ******
-4:  ******
-2:  ***********
0:   ****************
2:   ********
4:   **********
6:   ******
8:   ***********
10:  ******
12:  *
14:  **
16:  **
26:  *
> |
```

Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344

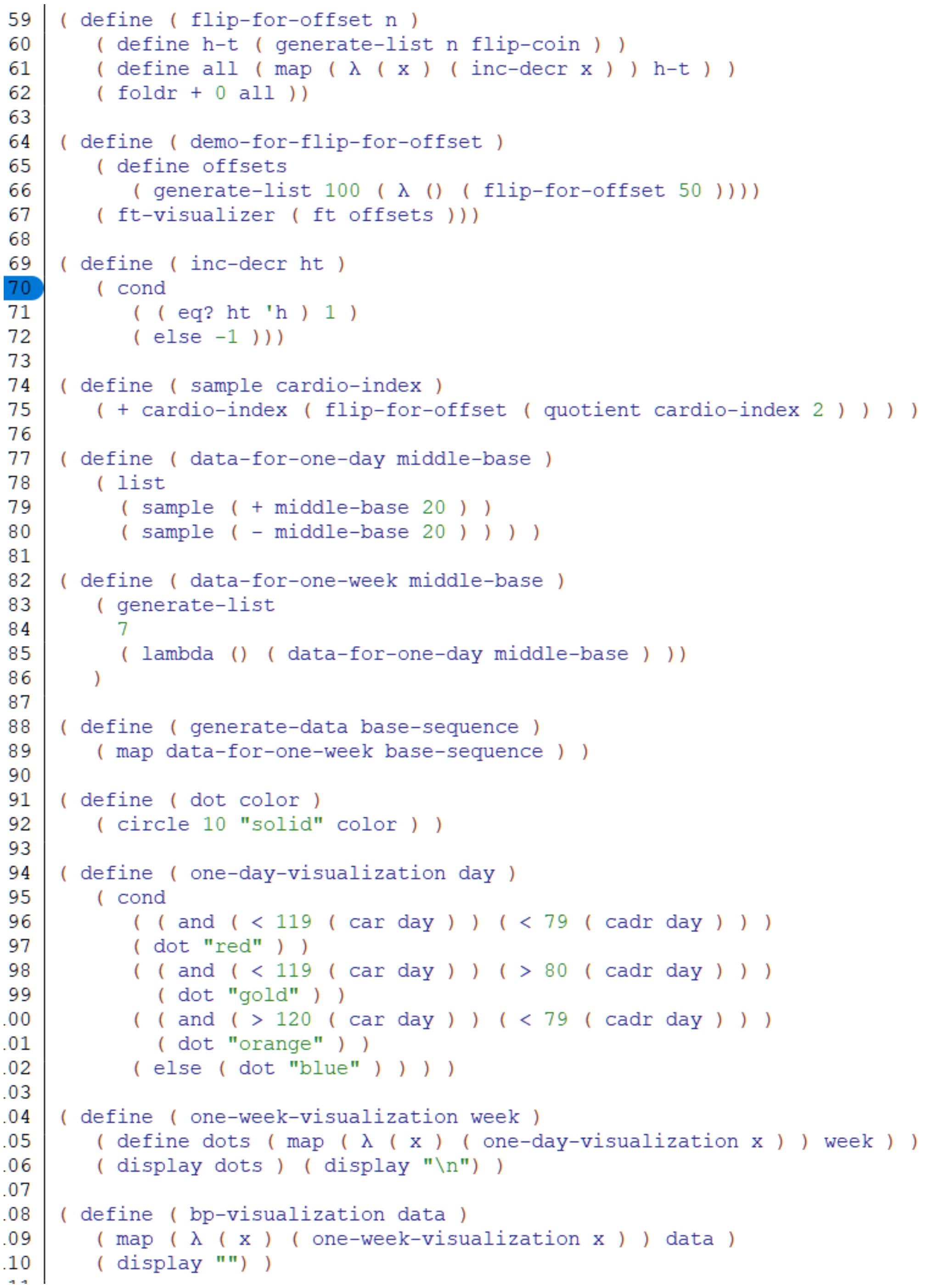
Problem 10

The Code



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

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

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

( define ( a-list l-one l-two )
   ( cond
      ( ( empty? l-one ) '() )
      ( else ( cons ( cons ( car l-one) ( car l-two) )
                    ( a-list ( cdr l-one ) ( cdr l-two ) ) ) ) ) )

( 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)ss
      ( else
        ( add-blanks ( string-append s " ") (- n 1)))))

( define ( generate-list n o )
   ( cond
      ( ( eq? n 0 ) '() )
      ( else
        ( cons ( o ) ( generate-list ( - n 1 ) o ) ))))

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

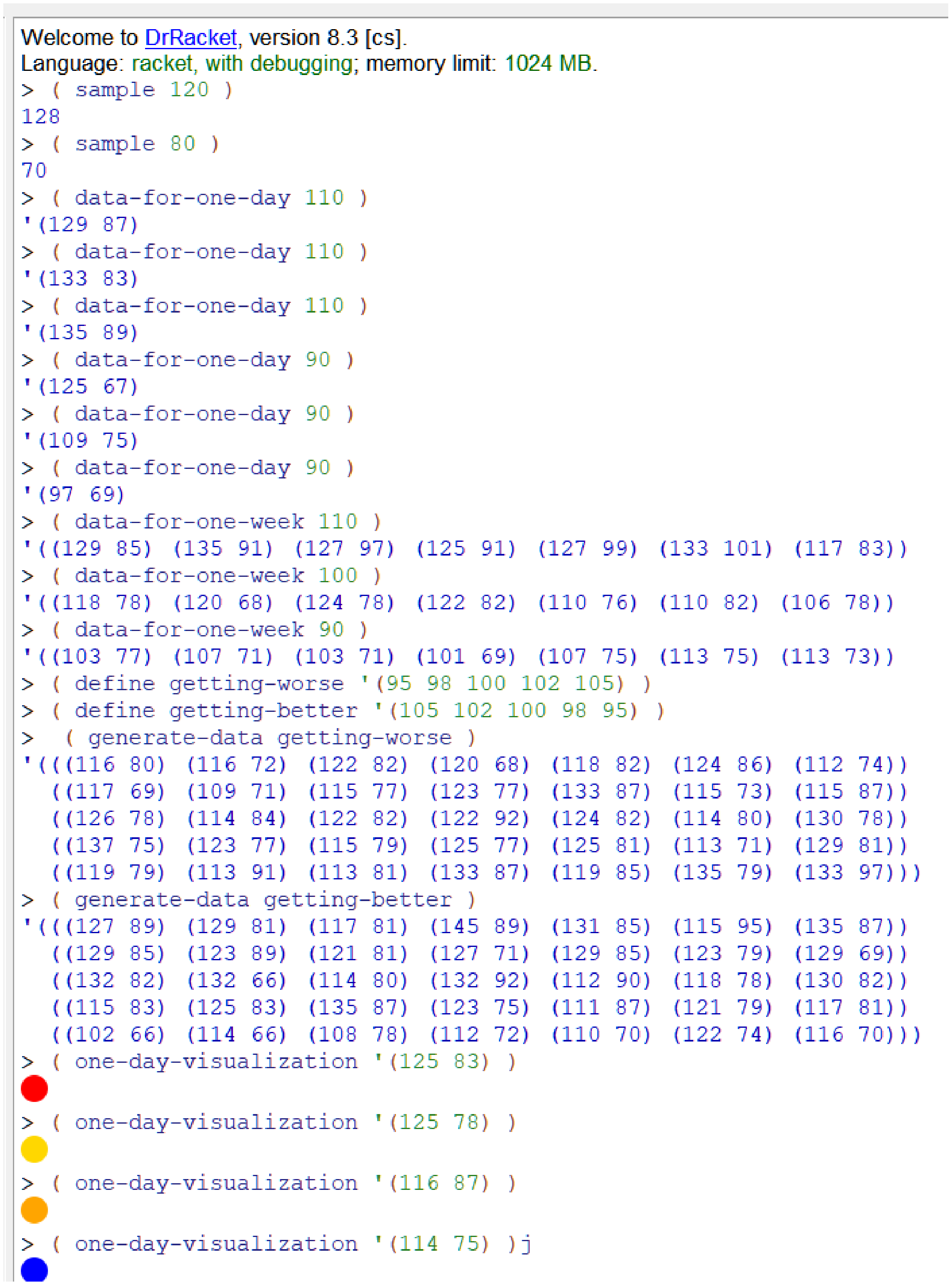
The Code (cont.)



```
( define ( flip-for-offset n )
   ( define h-t ( generate-list n flip-coin ) )
   ( define all ( map ( λ ( x ) ( inc-decr x ) ) h-t ) )
   ( foldr + 0 all ))

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

( define ( inc-decr ht )
   ( cond
      ( ( eq? ht 'h ) 1 )
      ( else -1 )))

( 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 ( dot color )
   ( circle 10 "solid" color ) )

( define ( one-day-visualization day )
   ( cond
      ( ( and ( < 119 ( car day ) ) ( < 79 ( cadr day ) ) )
      ( dot "red" ) )
      ( ( and ( < 119 ( car day ) ) ( > 80 ( cadr day ) ) )
        ( dot "gold" ) )
      ( ( and ( > 120 ( car day ) ) ( < 79 ( cadr day ) ) )
        ( dot "orange" ) )
      ( else ( dot "blue" ) ) ) ) )

( define ( one-week-visualization week )
   ( define dots ( map ( λ ( x ) ( one-day-visualization x ) ) week ) )
   ( display dots ) ( display "\n") )

( define ( bp-visualization data )
   ( map ( λ ( x ) ( one-week-visualization x ) ) data )
   ( display "") )
```

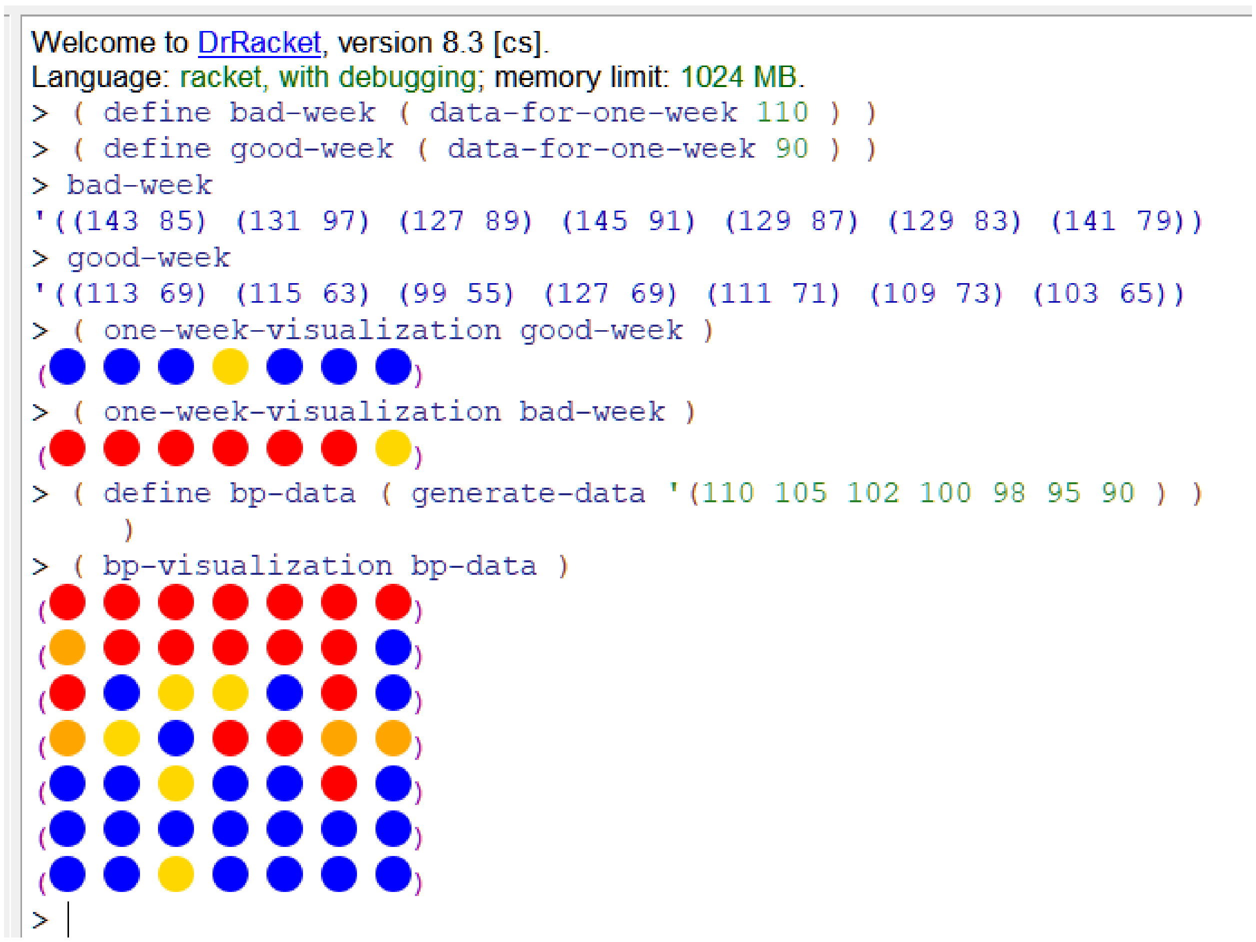
Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344

The Demos



```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
> ( sample 120 )
128
> ( sample 80 )
70
> ( data-for-one-day 110 )
'(129 87)
> ( data-for-one-day 110 )
'(133 83)
> ( data-for-one-day 110 )
'(135 89)
> ( data-for-one-day 90 )
'(125 67)
> ( data-for-one-day 90 )
'(109 75)
> ( data-for-one-day 90 )
'(97 69)
> ( data-for-one-week 110 )
'((129 85) (135 91) (127 97) (125 91) (127 99) (133 101) (117 83))
> ( data-for-one-week 100 )
'((118 78) (120 68) (124 78) (122 82) (110 76) (110 82) (106 78))
> ( data-for-one-week 90 )
'((103 77) (107 71) (103 71) (101 69) (107 75) (113 75) (113 73))
> ( define getting-worse '(95 98 100 102 105) )
> ( define getting-better '(105 102 100 98 95) )
>  ( generate-data getting-worse )
'(((116 80) (116 72) (122 82) (120 68) (118 82) (124 86) (112 74))
  ((117 69) (109 71) (115 77) (123 77) (133 87) (115 73) (115 87))
  ((126 78) (114 84) (122 82) (122 92) (124 82) (114 80) (130 78))
  ((137 75) (123 77) (115 79) (125 77) (125 81) (113 71) (129 81))
  ((119 79) (113 91) (113 81) (133 87) (119 85) (135 79) (133 97)))
> ( generate-data getting-better )
'(((127 89) (129 81) (117 81) (145 89) (131 85) (115 95) (135 87))
  ((129 85) (123 89) (121 81) (127 71) (129 85) (123 79) (129 69))
  ((132 82) (132 66) (114 80) (132 92) (112 90) (118 78) (130 82))
  ((115 83) (125 83) (135 87) (123 75) (111 87) (121 79) (117 81))
  ((102 66) (114 66) (108 78) (112 72) (110 70) (122 74) (116 70)))
> ( one-day-visualization '(125 83) )
●

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

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

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

Nathaniel Wolf
Racket assignment #4: Recursive List Processing and High Order Functions
3/30/2022
CSC 344



```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 1024 MB.
> ( define bad-week ( data-for-one-week 110 ) )
> ( define good-week ( data-for-one-week 90 ) )
> bad-week
'((143 85) (131 97) (127 89) (145 91) (129 87) (129 83) (141 79))
> good-week
'((113 69) (115 63) (99 55) (127 69) (111 71) (109 73) (103 65))
> ( one-week-visualization good-week )
(🔵 🔵 🔵 🟡 🔵 🔵 🔵)
> ( one-week-visualization bad-week )
(🔴 🔴 🔴 🔴 🔴 🔴 🟡)
> ( define bp-data ( generate-data '(110 105 102 100 98 95 90 ) )
    )
> ( bp-visualization bp-data )
(🔴 🔴 🔴 🔴 🔴 🔴 🔴)
(🟠 🔴 🔴 🔴 🔴 🔴 🔵)
(🔴 🔵 🟡 🟡 🔵 🔴 🔵)
(🟠 🟡 🔵 🔴 🔴 🟠 🟠)
(🔵 🔵 🟡 🔵 🔵 🔴 🔵)
(🔵 🔵 🔵 🔵 🔵 🔵 🔵)
(🔵 🔵 🟡 🔵 🔵 🔵 🔵)
> 
```