

# 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
>
```

```
1 | #lang racket
2 |
3 | ( define count
4 |   ( lambda ( object list )
5 |     ( cond
6 |       ( ( empty? list ) 0 )
7 |       ( ( equal? object ( first list ) )
8 |         ( add1 ( count object ( rest list ) ) ) )
9 |       ( else ( count object ( rest list ) ) ) )
10 |   )
11 | )
12 |
```

---

## Task 2: List -> Set

---

```
1 | #lang racket
2 |
3 | (define (list->set lst)
4 |   (cond
5 |     [ (empty? lst) '() ]
6 |     [ ( member (car lst) (cdr lst) )
7 |       (list->set (cdr lst)) ]
8 |     [else (cons (car lst) (list->set (cdr lst)))]
9 |   )
10| )
```

```
> (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

---

```
12| (define ( a-list lst1 lst2 )
13|   ( cond
14|     [ (empty? lst1) '() ]
15|     [ else
16|       ( cons ( cons (car lst1) (car lst2) )
17|         ( a-list (cdr lst1) (cdr lst2) ) ) ]
18|   )
19| )

> (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

---

```
12 (define ( a-list lst1 lst2 )
13   ( cond
14     [ (empty? lst1) '() ]
15     [ else
16       ( cons ( cons (car lst1) (car lst2) )
17         ( a-list (cdr lst1) (cdr lst2) ) ) ]
18   )
19 )
20
21 (define ( assoc obj a-list )
22   ( cond
23     [ (empty? a-list ) '() ]
24     [ (equal? obj (car (car a-list)))
25       (car a-list) ]
26     [ else
27       (assoc obj ( cdr a-list ) ) ]
28   )
29 )

> ( define all (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))) )
> all
'((one . un) (two . deux) (three . trois) (four . quatre))
> ( assoc 'two all)
'(two . deux)
> ( assoc 'five all)
'()
> 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-list )
  ( cond
    [ (empty? a-list ) '() ]
    [ (equal? obj (car (car a-list)))
      (car a-list) ]
    [ else
      (assoc obj ( cdr a-list ) ) ]
    )
  )

( 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 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 )
```



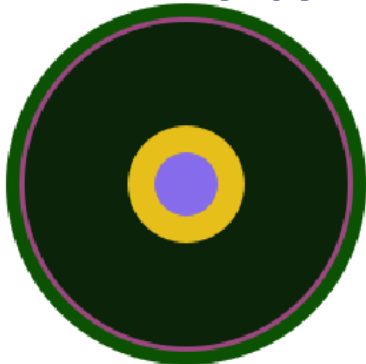
```
> ( sort-dots dots )
```



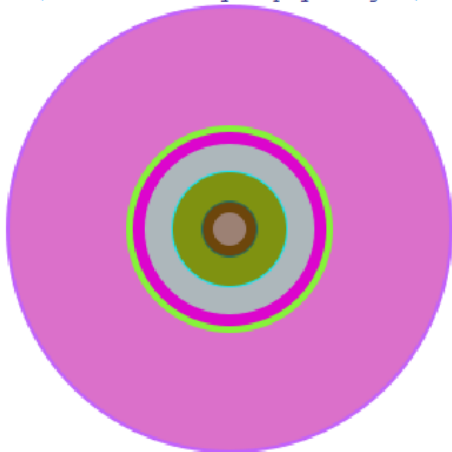
```
(list  
> ( foldr overlay empty-image ( sort-dots dots ) )
```



```
> ( 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 ) )
```



```

1  #lang racket
2
3  ( require 2htdp/image )
4
5  ( define ( roll-die )
6    ( + ( random 6 ) 1 ) )
7
8  ( define ( dot )
9    ( circle ( + 10 ( random 41 ) ) "solid" ( random-color ) )
10 )
11
12 ( define ( big-dot )
13   ( circle ( + 10 ( random 141 ) ) "solid" ( random-color ) )
14 )
15
16 ( define ( random-color )
17   ( color ( random 256 ) ( random 256 ) ( random 256 ) )
18 )
19
20 ( define ( sort-dots loc )
21   ( sort loc #:key image-width < )
22 )
23
24 ( define ( generate-list int obj )
25   ( cond
26     [ ( = int 0 ) ' ( ) ]
27     [ else
28       ( cons ( obj )
29         ( generate-list ( - int 1 ) obj ) ) ]
30   )
31 )

```

---

## Task 7: The Diamond

---

```

32 ( define ( random-color )
33   ( color ( random 256 ) ( random 256 ) ( random 256 ) )
34 )
35
36 ( define ( sort-diamonds loc )
37   ( sort loc #:key image-width < )
38 )
39
40 ( define ( generate-diamonds count )
41   ( cond
42     [ ( = count 0 ) ' ( ) ]
43     [ else
44       ( cons ( diamond ) ( generate-diamonds ( - count 1 ) ) ) ]
45   )
46 )
47
48 ( define ( diamond )
49   ( rotate 45 ( square ( + 20 ( random 380 ) ) "solid" ( random-color ) ) )
50 )
51
52 ( define ( diamond-design count )
53   ( foldr overlay empty-image ( sort-diamonds ( generate-diamonds count ) ) )
54 )

```



A diamond-shaped logo composed of concentric layers. The outermost layer is a thick green border. Inside this is a thin blue border, followed by a thick yellow border. The center of the logo is a solid dark blue diamond.

---

## 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 ) )
```



---

```
1 #lang racket
2
3 ( require 2htdp/image )
4
5 ( define ( a-list lst1 lst2 )
6   ( cond
7     [ (empty? lst1) '() ]
8     [ else
9       ( cons ( cons ( car lst1 ) ( car lst2 ) )
10              ( a-list ( cdr lst1 ) (cdr lst2 ) ) ) ]
11   )
12 )
13
14 ( define ( assoc obj a-list )
15   ( cond
16     [ (empty? a-list ) '() ]
17     [ (equal? obj ( car ( car a-list ) ) )
18       ( car a-list ) ]
19     [ else
20       ( assoc obj ( cdr a-list ) ) ]
21   )
22 )
23
24 ( define pitch-classes '( c d e f g a b ) )
25
26 ( define color-names '( blue green brown purple red yellow orange ) )
27
28 ( define ( box color )
29   ( overlay
30     ( square 30 "solid" color )
31     ( square 35 "solid" "black" )
32   )
33 )
34
35 ( define boxes
36   ( list
37     ( box "blue" )
38     ( box "green" )
39     ( box "brown" )
40     ( box "purple" )
41     ( box "red" )
42     ( box "gold" )
43     ( box "orange" )
44   )
45 )
```

```
46
47 ( define pc-a-list ( a-list pitch-classes color-names ) )
48
49 ( define cb-a-list ( a-list color-names boxes ) )
50
51 ( define ( pc->color pc )
52   ( cdr ( assoc pc pc-a-list ) )
53 )
54
55 ( define ( color->box color )
56   ( cdr ( assoc color cb-a-list ) )
57 )
58
59 ( define ( play notes-list )
60   ( define map-color ( map ( lambda ( x ) ( pc->color x ) ) notes-list ) )
61   ( define map-box   ( map ( lambda ( x ) ( color->box x ) ) map-color ) )
62   ( foldr beside empty-image map-box )
63 )
```

---

## 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:  *
```

```
1 | #lang racket
2 |
3 | ( define ( generate-list int obj )
4 |   ( cond
5 |     [ ( = int 0 ) ' ( ) ]
6 |     [ else
7 |       ( cons ( obj )
8 |         ( generate-list ( - int 1 ) obj ) ) ]
9 |   )
10 | )
11 |
12 | ( define count
13 |   ( lambda ( obj lst )
14 |     ( cond
15 |       [ ( empty? lst ) 0 ]
16 |       [ ( equal? obj ( first lst ) )
17 |         ( add1 ( count obj ( rest lst ) ) ) ]
18 |       [ else ( count obj ( rest lst ) ) ]
19 |     )
20 |   )
21 | )
22 |
23 | ( define ( ft-visualizer ft )
24 |   ( map pair-visualizer ft )
25 |   ( display "" )
26 | )
27 |
28 | ( define ( list->set lst )
29 |   ( cond
30 |     [ (empty? lst ) ' ( ) ]
31 |     [ ( member ( car lst ) ( cdr lst ) )
32 |       ( list->set ( cdr lst ) ) ]
33 |     [ else ( cons ( car lst ) ( list->set ( cdr lst ) ) ) ]
34 |   )
35 | )
36 |
37 | ( define ( a-list lst1 lst2 )
38 |   ( cond
39 |     [ ( empty? lst1 ) ' ( ) ]
40 |     [ else
41 |       ( cons ( cons ( car lst1 ) ( car lst2 ) )
42 |         ( a-list ( cdr lst1 ) ( cdr lst2 ) ) ) ]
43 |   )
44 | )
```

```

45
46 ( define ( ft the-list )
47   ( define the-set ( list->set the-list ) )
48   ( define the-counts
49     ( map ( lambda ( x ) ( count x the-list ) ) the-set )
50   )
51   ( define association-list ( a-list the-set the-counts ) )
52   ( sort association-list < #:key car )
53 )
54
55 ( define ( pair-visualizer pair )
56   ( define label
57     ( string-append ( number->string ( car pair ) ) ":" )
58   )
59   ( define fixed-size-label ( add-blanks label ( - 5 ( string-length label ) ) ) )
60   ( display fixed-size-label )
61   ( display
62     ( foldr
63       string-append
64       ""
65       ( make-list ( cdr pair ) "*" )
66     )
67   )
68   ( display "\n" )
69 )
70
71 ( define ( add-blanks s n )
72   ( cond
73     [ ( = n 0 ) s ]
74     [ else ( add-blanks ( string-append s " " ) ( - n 1 ) ) ]
75   )
76 )
77
78 ( define ( recursive-flip-for-offset n )
79   ( cond
80     ( ( = n 0 ) 0 )
81     ( else
82       ( define outcome ( flip-coin ) )
83       ( cond
84         ( ( eq? outcome 'h )
85           ( + ( recursive-flip-for-offset ( - n 1 ) ) 1 )
86         )
87         ( ( eq? outcome 't )
88           ( - ( recursive-flip-for-offset ( - n 1 ) ) 1 )
89         )
90       )
91     )
92   )
93 )

```

```

94 |
95 | ( define ( demo-for-recursive-flip-for-offset )
96 |   ( define offsets
97 |     ( generate-list
98 |       100
99 |       ( lambda () ( recursive-flip-for-offset 50 ) )
100 |     )
101 |   )
102 |   ( ft-visualizer (ft offsets ) )
103 | )
104 |
105 | ( define ( flip-coin )
106 |   ( define outcome ( random 2 ) )
107 |   ( cond
108 |     ( ( = outcome 1 )
109 |       'h
110 |     )
111 |     ( ( = outcome 0 )
112 |       't
113 |     )
114 |   )
115 | )
116 |
117 | ( define ( flip-for-offset n )
118 |   ( define occurence
119 |     ( map ( lambda ( x )
120 |       ( if (eq? x 'h ) 1 -1 ) )
121 |     ( generate-list n flip-coin ) ) )
122 |   ( foldr + 0 occurence )
123 | )
124 |
125 | ( define ( demo-for-flip-for-offset )
126 |   ( define offset-for-demo
127 |     ( generate-list 100 ( lambda () ( flip-for-offset 50 ) ) )
128 |   )
129 |   ( ft-visualizer ( ft offset-for-demo ) )
130 | )

```

---

## Task 10: Blood Pressure Trend Visualize

---

```
1 #lang racket
2
3 ( require 2htdp/image)
4 ; Given -----
5 ( define ( sample cardio-index )
6   ( + cardio-index ( flip-for-offset ( quotient cardio-index 2 ) ) )
7 )
8
9 ; Required -----
10
11 ( define ( flip-for-offset n )
12   ( define occurence
13     ( map ( lambda ( x )
14       ( if (eq? x 'h ) 1 -1 ) )
15     ( generate-list n flip-coin ) ) )
16   ( foldr + 0 occurence )
17 )
18
19 ( define ( generate-list int obj )
20   ( cond
21     [ ( = int 0 ) ' ( ) ]
22     [ else
23       ( cons ( obj )
24         ( generate-list ( - int 1 ) obj ) ) ]
25   )
26 )
27
28 ( define ( flip-coin )
29   ( define outcome ( random 2 ) )
30   ( cond
31     ( ( = outcome 1 )
32       'h
33     )
34     ( ( = outcome 0 )
35       't
36     )
37   )
38 )
39
40 ; Given -----
41
42 ( define ( data-for-one-day middle-base )
43   ( list
44     ( sample ( + middle-base 20 ) )
45     ( sample ( - middle-base 20 ) )
46   )
47 )
48
```



```

49 ; Given -----
50
51 ( define ( data-for-one-week middle-base )
52   ( generate-list
53     7
54     ( lambda () ( data-for-one-day middle-base ) )
55   )
56 )
57
58 ; Given -----
59
60 ( define ( generate-data base-sequence )
61   ( map data-for-one-week base-sequence )
62 )
63
64 ; One Day Visualization ---
65
66 ; ---- Dot ----
67 ( define ( dot color )
68   ( circle 10 "solid" color )
69 )
70
71 ( define ( one-day-visualization lst )
72   ( cond
73     [ ( >= ( car lst ) 120 )
74       ( cond
75         [ ( >= ( cadr lst ) 80 )
76           ( dot "red" ) ]
77         [ else
78           ( dot "gold" ) ]
79       )
80     ]
81   ; car < 120
82   [ else
83     ( cond
84       [ ( >= ( cadr lst ) 80 )
85         ( dot "orange" ) ]
86       [ else
87         ( dot "blue" ) ]
88     )
89   ]
90 )
91 )
92
93 ; One Week Visualization
94
95 ( define ( one-week-visualization lst )
96   ( display ( map one-day-visualization lst ) )
97   ; for serveral weeks
98   ( display "\n" )
99 )
100
101 ; Serveral Weeks Visualization
102
103 ( define ( bp-visualization lst )
104   ( map one-week-visualization lst )
105   ( display "" )
106 )

```

