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

Learning Abstract

This Racket Programming Assignment #4 is a total of 10 problems. Problems 1-4 use recursive functions that will help in the remaining problems. Problems 5 we can critically exam the code in order to answer 15 different questions. Problems 5-10 will use the first 4 problems as well as higher order functions in order to give me more experience using them. Overall the assignment is excellent practice of lambda functions, recursive functions and higher order functions.

Problem 1: Count

Code:

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

Demo:

```
> ( 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
>
```

Problem 2: list->set

Code:

```
13 (define (list->set list)
14   (cond
15     ((empty? list) '())
16     (else
17      (cond
18        ((member (car list) (cdr list))
19         (list->set (cdr list)))
20        (else
21         (cons (car list) (list->set (cdr list))))))))
```

Demo:

```
> ( 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: Association List Generator

Code:

```
23 (define (a-list list1 list2)
24   (cond
25     ((empty? list1) '())
26     (else
27      (cons (cons (car list1)(car list2))(a-list (cdr list1)(cdr list2)))))))
```

Demo:

```
> ( 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))
>
```

Problem 4: Assoc

Code:

```
29 (define (assoc object al)
30   (cond
31     ((empty? al) '())
32   ((equal? object (car (car al)))
33    (cons (car al) (assoc object (cdr al))))
34   (else
35    (assoc object (cdr al)))))
```

Demo:

```
> ( 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 )
'()
>
```

Problem 5: Frequency Table

Code:

```
#lang racket

(define (count object object-list)
  (cond
    ((empty? object-list) 0)
    (else
     (cond
       ((equal? object (car object-list))
        (+ 1 (count object (cdr object-list))))
       (else
        (count object (cdr object-list)))))))

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

(define (a-list list1 list2)
  (cond
    ((empty? list1) '())
    (else
     (cons (cons (car list1)(car list2))(a-list (cdr list1)(cdr list2))))))

(define (assoc object al)
  (cond
    ((empty? al) '())
  ((equal? object (car (car al)))
   (cons (car al) (assoc object (cdr al))))
  (else
   (assoc object (cdr al)))))

( 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 )
```

```
47      ( map pair-visualizer ft )
48      ( display "" )
49  )
50
51  ( define ( pair-visualizer pair )
52      ( define label
53          ( string-append ( number->string ( car pair ) ) ":" )
54          )
55      ( define fixed-size-label ( add-blanks label ( - 5 ( string-length label ) ) ) )
56      ( display fixed-size-label )
57      ( display
58        ( foldr
59          string-append
60          ""
61          ( make-list ( cdr pair ) "*" )
62          )
63        )
64      ( display "\n" )
65  )
66
67  ( define ( add-blanks s n )
68      ( cond
69          ((= n 0)s)
70          ( else ( add-blanks ( string-append s " " ) ( - n 1 ) ) )
71          )
72  )
```

Demo:

```
>  ( 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:   *
>
```

Questions:

1. List the names of the functions used within the ft function that you were asked to write in this programming assignment.

I wrote the count, list->set, and a-list function in order for the ft function to 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.

(map (lambda (x) (count x the-list)) the-set)

The higher order function map is using a lambda function. The lambda function being used takes in one parameter, x, which is plugged into the function later as 'the-set', defined right beforehand. The-set is defined as the-list being plugged into list->set which removes all multiples in a list to give you a set of one of each item. When the lambda function takes this set in as an argument, it passes it to the count function, which counts the-set inside of the-list.

3. How many parameters must the functional argument to the application of map in the ft function take?

2

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.

The function would have to go through each value inside the list, and then still be mapped properly to the set making it difficult because map only takes in 2 arguments.

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.

Association-list

6. What is a "keyword argument"?

A keyword argument is specified with a keyword followed by the argument expression. The argument expression is often used as a key in a map function to find a specific object/item.

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.

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?

Because there are only 2 arguments provided for map, in which case the 2nd argument serves as the argument for map and the 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.

String-append

10. Does the add-blanks function make use of any higher order functions?

no

11. Why do you think the display function, with the empty string as its argument, was called in the ft-visualizer function?

This is used to help format the output properly.

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.

Stem and leaf structure. The function uses dotted pairs from the ft to output stars in the visualizer.

13. Is the make-list function used in the pair-visualizer function a primitive function in Racket?

Yes

14. What do you think is the most interesting aspect of the given frequency table generating code?

I enjoy that it uses the functions we made earlier inside the ft function. It gives me a better understanding on how the code works overall.

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.

I believe this code functions very well and is written well. However it may be difficult for a new user to racket to understand. Is there a way you would've writen it differently to make it easier to read for beginners?

Problem 6: Generate List

Code:

```
74 (require 2htdp/image)
75
76 (define (roll-die) (+(random 6)1))
77 (define (random-color)(color (random 256)(random 256)(random 256)))
78 (define (dot)(circle (+ 10 (random 41))"solid"(random-color)))
79 (define (big-dot)(circle(+ 10 (random 100))"solid"(random-color)))
80
81 (define (generate-list num object)
82   (cond ((equal? num 0) '())
83         (else
84          (cons (object)(generate-list (- num 1)object))
85          )))
86 (define (sort-dots loc)(sort loc #:key image-width <))
```

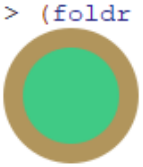
Demo 1:

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

Demo 2:

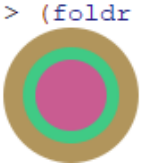
```
> (define dots (generate-list 3 dot))
> dots
```

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



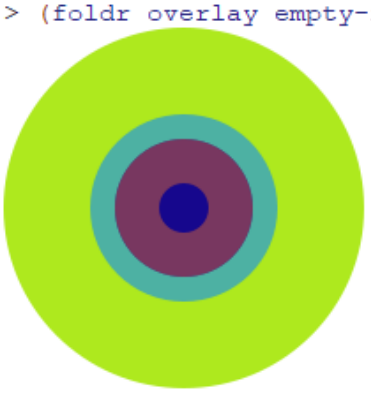
```
> (sort-dots dots)
```

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

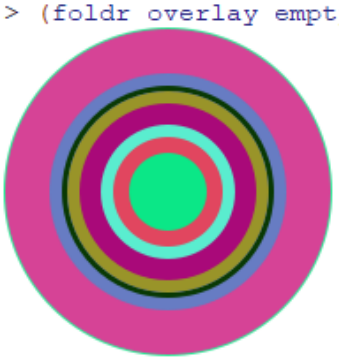


Demo 3:

```
> (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))
```



```
>
```

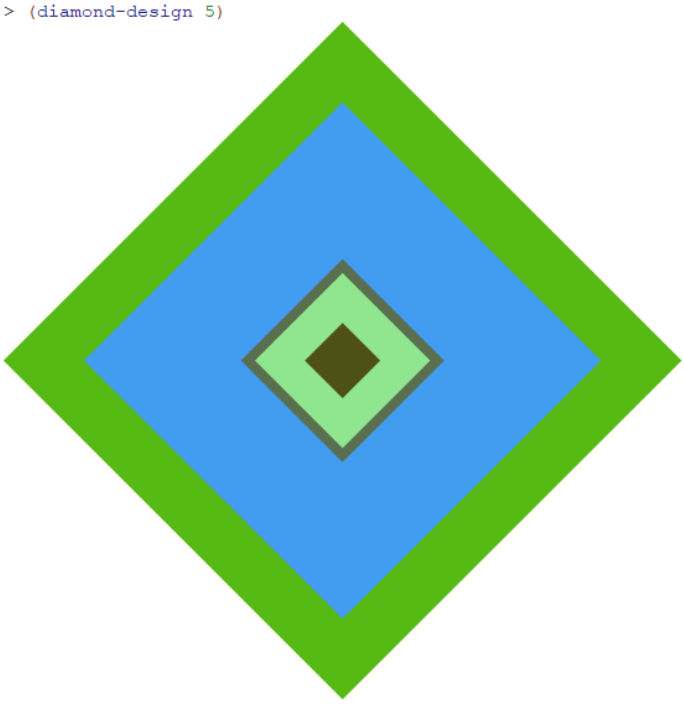
Problem 7: The Diamond

Code:

```
74 (require 2htdp/image)
75
76 (define (roll-die) (+(random 6)1))
77 (define (random-color)(color (random 256)(random 256)(random 256)))
78 (define (dot)(circle (+ 10 (random 41))"solid"(random-color)))
79 (define (big-dot)(circle(+ 10 (random 100))"solid"(random-color)))
80
81 (define (generate-list num object)
82   (cond ((equal? num 0) '())
83         (else
84          (cons (object)(generate-list (- num 1)object))
85          )))
86 (define (sort-dots loc)(sort loc #:key image-width <))
87
88 (define (diamond)(rotate 45 (square (+ 20 (random 381))"solid"(random-color))))
89
90 (define (diamond-design num)
91   (define diamonds (generate-list num diamond))
92   (foldr overlay empty-image(sort-dots diamonds)))
```

Demo 1:

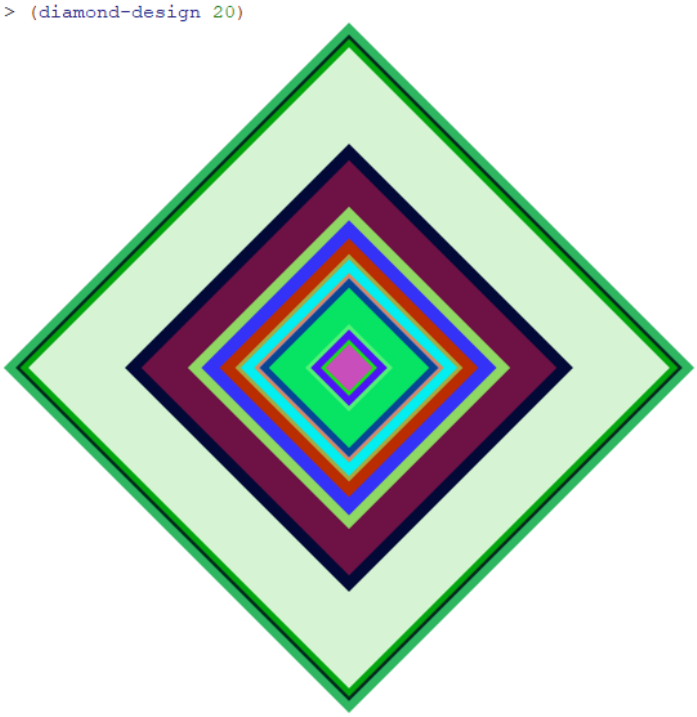
```
> (diamond-design 5)
```



```
>
```

Demo 2:

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



```
>
```

Problem 8: The Diamond

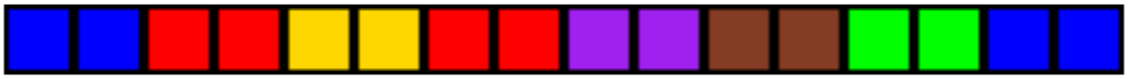
Code:

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

(define (a-list first-list second-list)
  (cond
    ((empty? first-list) '())
    (else
     (cons (cons (car first-list) (car second-list))
           (a-list (cdr first-list)(cdr second-list))))))

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

(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 pitch-list)
  (define pitch-class-color-list (map pc->color pitch-list))
  (define color-box-list (map color->box pitch-class-color-list))
  (foldr beside empty-image color-box-list)
  )
```

Demo:

```
> (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))
```

```
>
```

Problem 9: Transformation of a Recursive Sample

Code:

```
#lang racket
(define (generate-list num object)
  (cond
    ((zero? num)'())
    (else
     (cons (object)(generate-list(- num 1) object)))))


(define (count object object-list)
  (cond
    ((empty? object-list) 0)
    (else
     (cond
       ((equal? object (car object-list))
        (+ 1 (count object (cdr object-list))))
       (else
        (count object (cdr object-list)))))))

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

(define (a-list list1 list2)
  (cond
    ((empty? list1) '())
    (else
     (cons (cons (car list1)(car list2))(a-list (cdr list1)(cdr list2))))))

( 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
```

```
51        ( string-append ( number->string ( car pair ) ) ":" )
52        )
53     ( define fixed-size-label ( add-blanks label ( - 5 ( string-length label ) ) ) )
54     ( display fixed-size-label )
55     ( display
56       ( foldr
57         string-append
58         ""
59         ( make-list ( cdr pair ) "*" )
60         )
61       )
62     ( display "\n" )
63  )
64
65  ( define ( add-blanks s n )
66     ( cond
67        ((= n 0)s)
68        ( else ( add-blanks ( string-append s " " ) ( - n 1 ) ) )
69        )
70  )
71
72  ( define ( recursive-flip-for-offset n )
73  ( cond
74  ((= n 0)0)
75  ( else
76  ( define outcome ( flip-coin ) )
77  ( cond
78  ( ( eq? outcome 'h )
79  ( + ( recursive-flip-for-offset ( - n 1 ) ) 1 )
80  )
81  ( ( eq? outcome 't )
82  ( - ( recursive-flip-for-offset ( - n 1 ) ) 1 ))))))
83
84  ( define ( flip-coin )
85  ( define outcome ( random 2 ) )
86  ( cond
87  ( ( = outcome 1 )
88  'h
89  )
90  ( ( = outcome 0 ))))
91
92  ( define ( demo-for-recursive-flip-for-offset )
93  ( define offsets
94  ( generate-list 100
95  ( lambda () ( recursive-flip-for-offset 50 ) )))
96  ( ft-visualizer (ft offsets ) )
97  )
```

Demo:

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

Problem 10: Blood Pressure Trend Visualizer

Code:

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

(define (generate-list num object)
  (cond
    ((zero? num)'())
    (else
     (cons (object)(generate-list(- num 1) object)))))


(define (count object object-list)
  (cond
    ((empty? object-list) 0)
    (else
     (cond
       ((equal? object (car object-list))
        (+ 1 (count object (cdr object-list))))
       (else
        (count object (cdr object-list)))))))

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

(define (a-list list1 list2)
  (cond
    ((empty? list1) '())
    (else
     (cons (cons (car list1)(car list2))(a-list (cdr list1)(cdr list2))))))

( 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
```

```
51        ( string-append ( number->string ( car pair ) ) ":" )       )
52     ( define fixed-size-label ( add-blanks label ( - 5 ( string-length label ) ) ) )
53     ( display fixed-size-label )
54     ( display
55       ( foldr
56         string-append
57         ""
58         ( make-list ( cdr pair ) "*" )
59         ))
60     ( display "\n" ))
61
62  ( define ( add-blanks s n )
63     ( cond
64        ((= n 0)s)
65        ( else ( add-blanks ( string-append s " " ) ( - n 1 ) ) ) ) )
66
67  ( define ( flip-for-offset n )
68  ( cond
69  ((= n 0)0)
70  ( else
71  ( define outcome ( flip-coin ) )
72  ( cond
73  ( ( eq? outcome 'h )
74  ( + ( flip-for-offset ( - n 1 ) ) 1 )
75  )
76  ( ( eq? outcome 't )
77  ( - ( flip-for-offset ( - n 1 ) ) 1 ))))))
78
79  ( define ( flip-coin )
80  ( define outcome ( random 2 ) )
81  ( cond
82  ( ( = outcome 1 )'h)
83  ( ( = outcome 0 )'t)))
84
85  ( define ( demo-for-recursive-flip-for-offset )
86  ( define offsets
87  ( generate-list 100
88  ( lambda () ( flip-for-offset 50 ) ) ) )
89  ( ft-visualizer (ft offsets ) ) )
90
91  ( define ( sample cardio-index )
92  ( + cardio-index ( flip-for-offset ( quotient cardio-index 2))))
93
94  (define (data-for-one-day middle-base)
95    (list (sample (+ middle-base 20))
96          (sample (- middle-base 20))))
97
98  (define (data-for-one-week middle-base)
```

```
 99    (generate-list 7 (lambda () (data-for-one-day middle-base))))
100
101  (define (generate-data base-sequence)
102    (map data-for-one-week base-sequence))
103
104  (define (one-day-visualization bp)
105    (define red(circle 10 "solid" "red"))
106    (define gold(circle 10 "solid" "gold"))
107    (define orange(circle 10 "solid" "orange"))
108    (define blue (circle 10 "solid" "blue"))
109    (cond((< (car bp) 120)
110          (cond ((< (car(cdr  bp)) 80) blue)
111                (else orange)))
112          (else
113           (cond((< (car(cdr bp)) 80) gold)
114                (else red)))))
115
116  (define (one-week-visualization input)
117    (display (map one-day-visualization input))
118    (display "\n"))
119
120  (define (bp-visualization data)
121    (map one-week-visualization data)(display"""))
```

Demo:

```
> (sample 120)
134
> (sample 80)
84
> ( data-for-one-day 110 )
'(137 83)
> ( data-for-one-day 110 )
'(125 87)
> ( data-for-one-day 110 )
'(145 93)
>  ( data-for-one-day 90 )
'(109 85)
>  ( data-for-one-day 90 )
'(101 79)
>  ( data-for-one-day 90 )
'(111 71)
> ( data-for-one-week 110 )
'((131 81) (129 87) (147 81) (135 97) (129 97) (125 85) (127 83))
> ( data-for-one-week 100 )
'((126 76) (104 82) (128 70) (118 86) (120 84) (112 88) (120 76))
> ( data-for-one-week 90 )
'((101 71) (123 71) (113 75) (115 75) (113 65) (107 75) (99 75))
>  ( define getting-worse '(95 98 100 102 105) )
define: bad syntax (multiple expressions after identifier) in: (define getting-worse ' (95 98 100 102 105))
>  ( define getting-worse '(95 98 100 102 105) )
>  ( define getting-better '(105 102 100 98 95) )
>  ( generate-data getting-worse )
'(((124 74) (124 66) (106 70) (112 80) (114 80) (98 78) (114 78))
  ((119 77) (113 87) (107 83) (117 77) (117 93) (127 83) (111 83))
  ((132 76) (106 70) (108 84) (118 78) (118 84) (134 78) (124 72))
  ((117 79) (113 89) (119 85) (123 91) (115 85) (111 71) (117 95))
  ((115 97) (133 95) (135 91) (127 83) (125 91) (113 91) (121 85)))
> ( generate-data getting-better )
'(((135 89) (137 71) (113 85) (121 89) (125 93) (123 89) (117 89))
  ((125 85) (123 85) (123 75) (117 83) (113 73) (135 97) (129 77))
  ((110 86) (108 86) (110 84) (110 86) (112 76) (122 74) (120 86))
  ((109 85) (125 79) (125 69) (121 87) (123 67) (119 77) (127 75))
  ((118 78) (128 80) (108 76) (108 78) (132 86) (118 74) (114 68)))
> (one-day-visualization '(125 83))
```

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

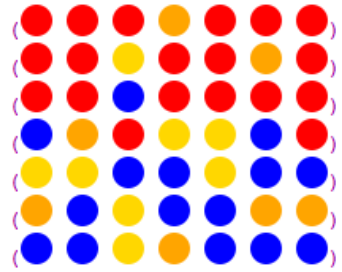
```
> (one-day-visualization '(116 87))
```

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

```
> (define bad-week(data-for-one-week 110))
> (define good-week(data-for-one-week 90))
> bad-week
'((125 93) (131 89) (131 93) (131 87) (121 89) (127 97) (137 95))
> (one-week-visualization bad-week)
```

```
> good-week
'((105 71) (111 75) (99 67) (105 67) (119 75) (103 61) (119 69))
> (one-week-visualization good-week)
```

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



```
>
```