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

What's It All About?

Some simple recursive list processing functions are featured in the first part of this assignment, including a couple which feature association lists, a classic Lisp structure. Some simple higher order functions are featured in the second part of this assignment, including some which feature colorful renderings of diverse phenomena.

Problem 1 - Count

Specification

Define a **recursive** function called `count`, consistent with the given pseudocode, according to the following specification:

1. The first parameter is presumed to be a Lisp object.
2. The second parameter is presumed to be a list of Lisp objects .
3. The value of the function will be the number of occurrences of the object in the 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
>
```

Pseudocode

```
to count the number of occurrences of an object in a list do
   if ( the list is empty ) then
      return 0
   else if ( the object equals the first element of the list ) then
      return 1 +  the count of the object in the rest of the list
   else
      return the count of the object in the rest of the list
   end of the if
end of the function
```

Task

1. Write the recursive function definition, consistent with the given pseudocode.
2. Mimic the demo.
3. Build the source and the demo into your presentation document.

Problem 2 - list->set

Specification

Define a **recursive** function called `list->set`, consistent with the given pseudocode, according to the following specification:

1. The sole parameter is presumed to be a list.
2. The value of the function will be a list consisting of just one occurrence of each element in the given 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)
>
```

Pseudocode

```
to convert a list to a set do
   if ( the list is empty ) then
      return the empty list
   else if ( the first element of the list is in the rest of the list ) then
      return the result of converting of the rest of the list to a set
   else
      return the cons of the first element of the list with
             the result of converting of the rest of the list to a set
   end of the if
end of the function
```

Task

1. Write the recursive function definition, consistent with the given pseudocode.
2. Mimic the demo.
3. Build the source and the demo into your presentation document.

Problem 3 - Association List Generator

An **association list**, or **a-list**, is a a list of cons cells. Often the cons cells contain an atomic car and an atomic cdr, but this is not necessarily the case.

Specification

Define a **recursive** function called `a-list` according to the following specification:

1. The first parameter is presumed to be a list of objects.
2. The second parameter is presumed to be a list of objects of the same length as the value of the first parameter.
3. The value of the function will be a list of pairs obtained by "cons-ing" successive elements of the two lists.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 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))
>
```

Task

1. Write the recursive function definition.
2. Mimic the demo.
3. Build the source and the demo into your presentation document.

Problem 4 - Assoc

Specification

Define a **recursive** function called `assoc` according to the following specification:

1. The first parameter is presumed to be a lisp object.
2. The second parameter is presumed to be an association list.
3. The value of the function will be the first pair in the given association list for which the car of the pair equals the value of the first parameter, or '() if there is no such element.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( 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 )
'()
>
```

Task

1. Write the recursive function definition.
2. Mimic the demo.
3. Build the source and the demo into your presentation document.

Problem 5 - Frequency Table

The Nature of this Task

This task is more of a reading/study/analysis task than a function composition task. Here is the script:

1. I will provide you with a demo for (1) a frequency table generating function, and (2) a very simple frequency table visualization program.
2. I will provide you with the code for these two programs, and the supporting code that they reference.
3. You will be asked to type in the code, and mimic the demo.
4. You will be asked to answer several questions about the code.
5. You will be asked to incorporate the code, the demo, and your answers to the questions into your solution document.

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:  *
>
```

The Code

```
( 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 ) ) )
  )
)
```

Questions (for the most part)

1. List the names of the functions used within the `ft` function that you were asked to write in this programming assignment.
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.
3. How many parameters must the functional argument to the application of `map` in the `ft` function take?
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.
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.
6. What is a “keyword argument”?
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.
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?
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.
10. Does the `add-blanks` function make use of any higher order functions?
11. Why do you think the `display` function, with the empty string as its argument, was called in the `ft-visualizer` function?
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.
13. Is the `make-list` function used in the `pair-visualizer` function a primitive function in Racket?
14. What do you think is the most interesting aspect of the given frequency table generating code?
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.

Task

1. Type in the code.
2. Mimic the demo.
3. Answer the “questions”, being sure to contextualize each answer with its question.
4. Build the source and the demo and your question/answer sequence into your presentation document.

Problem 6 - Generate list

Specification

Define a **recursive** function called `generate-list` according to the following specification:

1. The first parameter is a nonnegative integer.
2. The second parameter is a parameterless function that returns a lisp object.
3. The function returns a list of length equal to the value of the first parameter containing objects created by calls to the function represented by the second parameter.

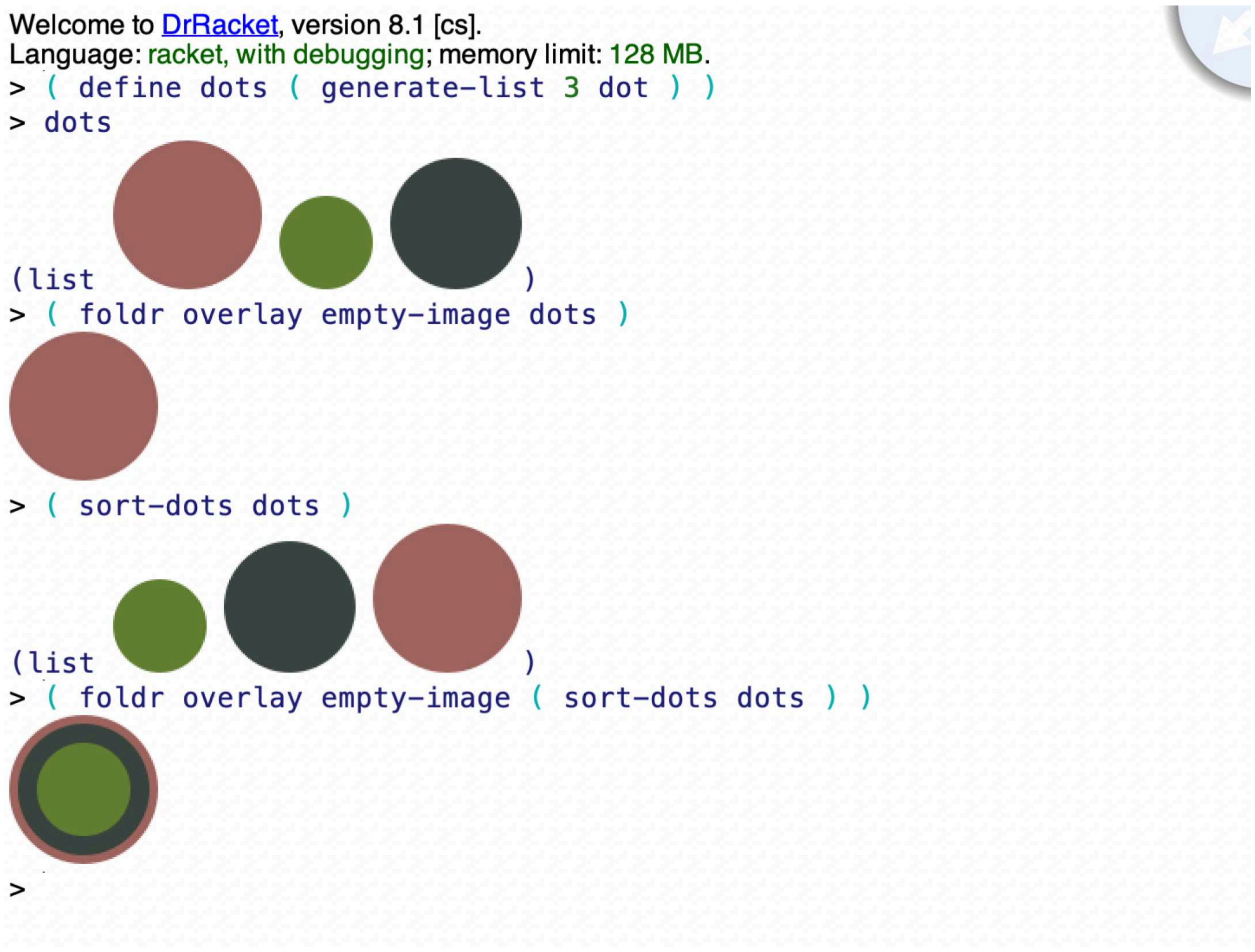
Some auxiliary code to support the demo

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

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

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

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

Demo 1

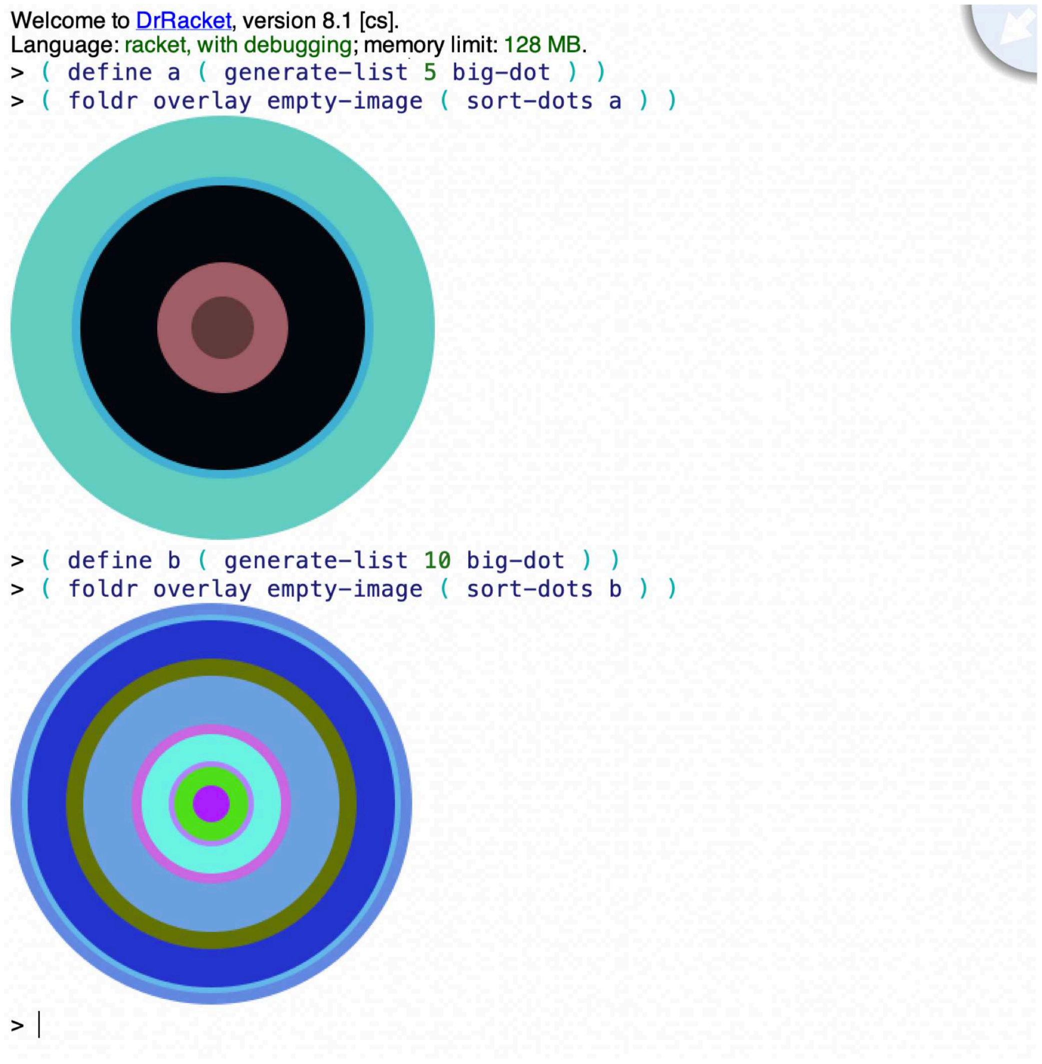
```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( generate-list 10 roll-die )
'(6 2 5 6 3 3 2 3 1 6)
> ( generate-list 20 roll-die )
'(3 2 5 4 3 3 5 1 4 6 1 6 6 2 5 6 1 1 2 6)
> ( generate-list 12
    ( lambda () ( list-ref '( red yellow blue ) ( random 3 ) ) )
  )
'(yellow yellow blue yellow blue blue blue blue blue red blue yellow)
>
```

Demo 2



```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 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



```
Welcome to DrRacket, version 8.1 [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 ) )
```

```
> |
```

Task

Write the recursive function definition, mimic the **three** demos, and build the source code and the **three** demos into your presentation document.

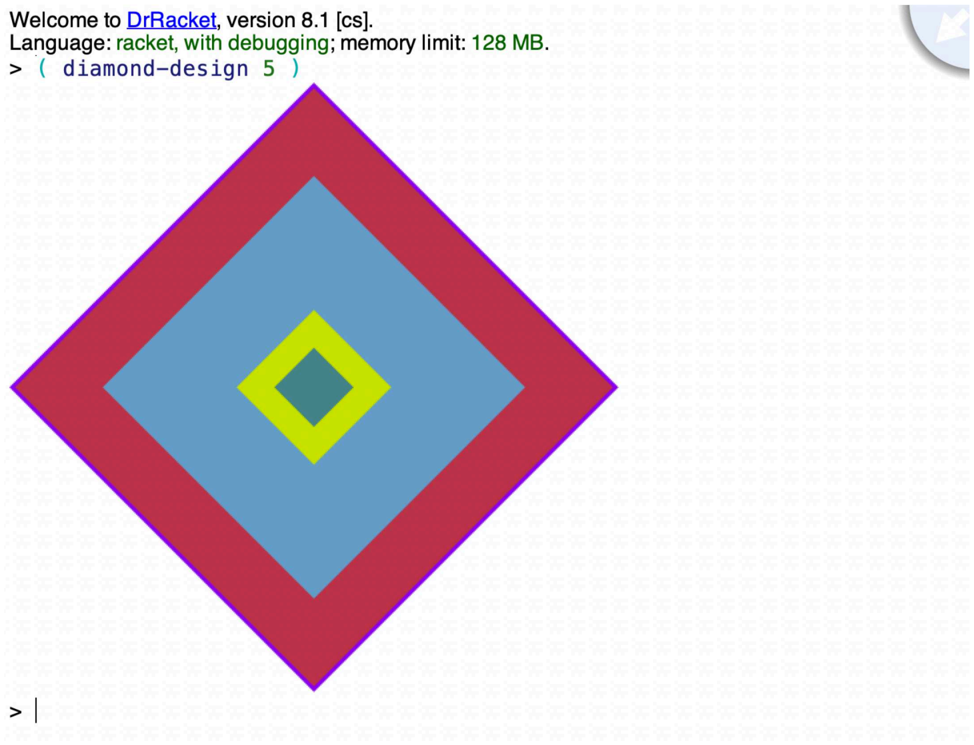
Problem 7 - The Diamond

Specification

Using what you learned from Problem 6 as a hint, define a function called `diamond` that is consistent with the following specification:

1. The sole parameter is a number indicating how many diamonds will be featured in the design.
2. The function returns an image which consists of the number of diamonds specified by the parameter, where each diamond is randomly colored and has a side length between 20 and 400.

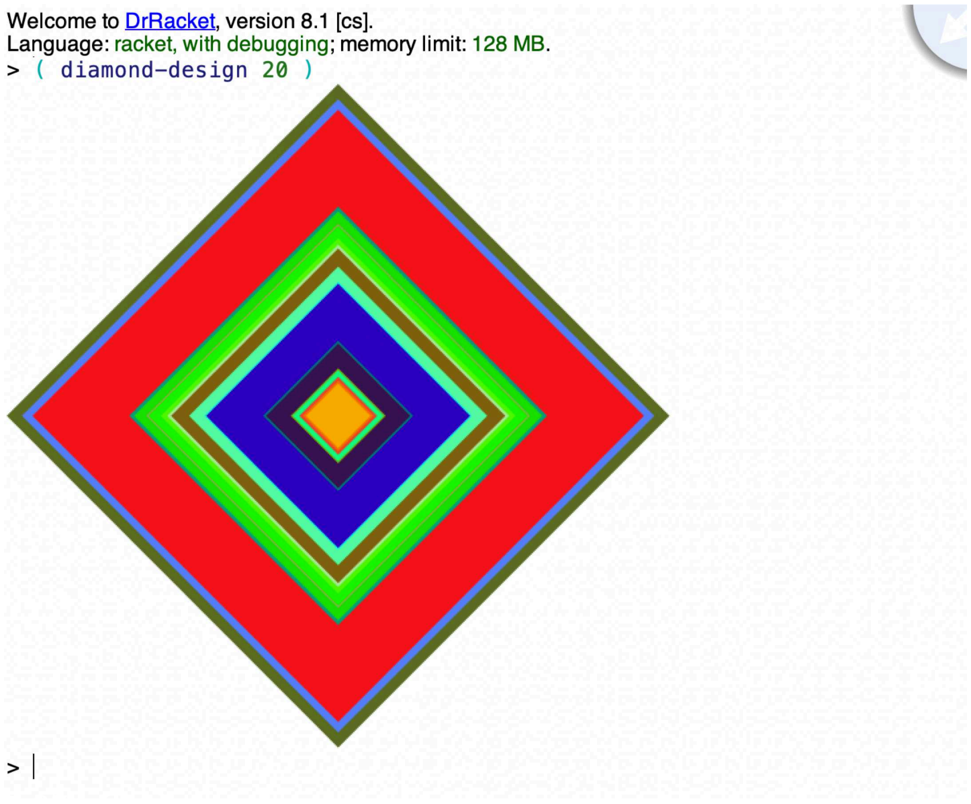
Demo 1



```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( diamond-design 5 )
```

```
> |
```

Demo 2



```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( diamond-design 20 )
```

```
> 
```

Task

Write the function recursive function definition, mimic the **two** demos, and build the source code and the **two** demos into your presentation document.

Problem 8 - Chromesthetic renderings

Specification

Define a function called `play` according to the following specification:

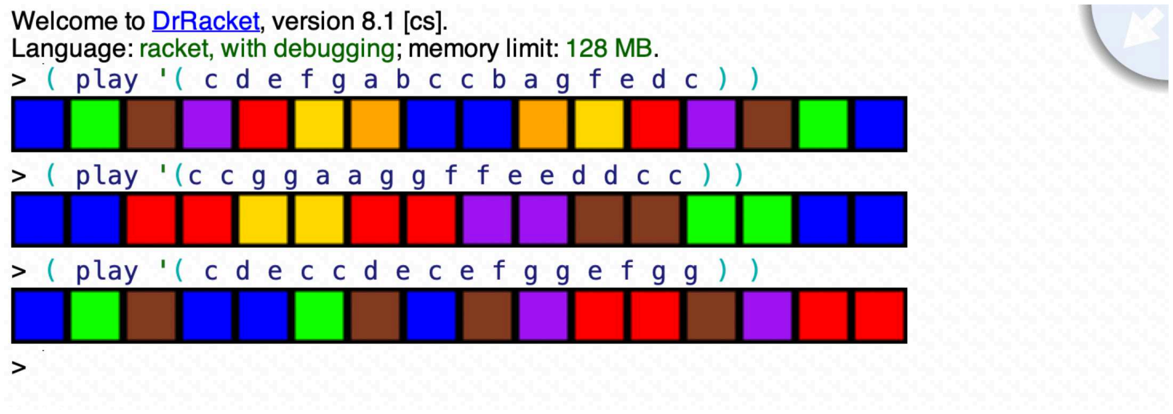
1. The sole parameter is a list of pitch names drawn from the set {c, d, e, f, g, a, b}.
2. The result is an image consisting of a sequence of colored squares in black frames, with the colors determined by the following mapping: c→blue; d→green; e→brown; f→purple; g→red; a→gold; b→orange.

Constraint: Your function definition must use `map` twice and `foldr` one time.

Some auxiliary for you to use

```
( 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 ) )
)
```

Demo



```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 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 ) )
>
```

Task

Write the function definition subject to the various constraints, mimic the demo, and build the source code and the demo into your presentation document.

Problem 9 - Transformation of a Recursive Sampler

1. Please study the "recursive flip for offset" function, which computes an "offset" from zero that is determined by flipping a coin a specified number of times. The **offset** is essentially the number of heads minus the number of tails, for a given number of flips. Then, study the demo.
2. Define an equivalent function which does not use recursion, but which uses three higher order functions, `generate-list`, `map`, and `foldr`.

Code to Study

```
( 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 )
          )
        )
      )
  )
)

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

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

Demo to Study

```
> ( recursive-flip-for-offset 100 )
12
> ( recursive-flip-for-offset 100 )
8
> ( recursive-flip-for-offset 100 )
-14
> ( recursive-flip-for-offset 100 )
0
> ( recursive-flip-for-offset 100 )
12
> ( demo-for-recursive-flip-for-offset )
-22: *
-20: *
-10: ***
-8:  *******
-6:  ********
-4:  **************
-2:  *********
0:   ***************
2:   *********
4:   ***********
```

```
6:   *************
8:   ***
10:  ***
12:  *
18:  *
20:  *
> ( demo-for-recursive-flip-for-offset )
-16: *
-14: ***
-12: ***
-10: *******
-8:  ******
-6:  *********
-4:  **********
-2:  **********
0:   *********
2:   *******
4:   ********
6:   ****
8:   ********
10:  ******
12:  **
14:  ***
16:  **
18:  *
20:  *
>
```

Task

1. Define a **non-recursive** functionally equivalent variant of `recursive-flip-for-offset`, which uses all three of the higher order functions `generate-list`, and `map`, and `foldr`. Call this function `flip-for-offset`. Note that you will need to write some simple functions so support your use of the higher order function approach.
2. Define a variant of the given `demo-for-recursive-flip-for-offset` function called `demo-for-flip-for-offset` which demonstrates that your nonrecursive variant of my recursive function works as it should.
3. Perform a demo that is analogous to the given demo, but which uses your `flip-for-demo` function and your `demo-for-flip-for-offset` function.
4. Build the source code that you are asked to write and the demo that you are asked to generate into your solution document.

Problem 10 - Blood Preassure Trend Visualizer

This problem pertains to visualizing tends in blood pressure over time. To set the stage for the programming that you are going to be asked to do, a bit of information pertaining to blood pressure is presented, and then some demos are presented. After that, you're subtasks for this problem are presented.

Please think of this problem as a collaborative programming exercise. I will share some code. You will add some. In the end, we will have written a complete blood pressure visualization program. This will be accomplished incrementally. When I provide a bit of code with demo, please enter the code and do the demo, just to be sure that you have entered the code correctly. When you are asked to write some code, please do so, and then demo it in the manner suggested.

After completing the sequence of tasks related to getting the program to work, I will specify how you should integrate code and demo into your presentation document.

Something preliminary about blood pressure

Your blood pressure is usually determined by two numbers, the first higher than the second. For example, a blood pressure reading might be: 118/77. Your blood pressure is generally considered normal if the higher number is less than 120 and the lower number is less than 80. Normally, a "cuff" is used to take a blood pressure reading. In the situation at hand, we will merely generate readings at random, subject to two normal distributions, or something like it.

Generating a Sample

Some code for you to type in ...

```
( define ( sample cardio-index )
  ( + cardio-index ( flip-for-offset ( quotient cardio-index 2 ) ) )
)
```

Here is a suggestive demo ...

```
> ( sample 120 )
116
> ( sample 80 )
78
>
```

Generating One Blood Pressure Reading

Some code for you to type in ...

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

Here is a suggestive demo ...

```
> ( data-for-one-day 110 )
'(133 85)
> ( data-for-one-day 110 )
'(128 95)
> ( data-for-one-day 110 )
'(131 89)
> ( data-for-one-day 90 )
'(101 71)
> ( data-for-one-day 90 )
'(115 73)
> ( data-for-one-day 90 )
'(111 69)
>
```

Generating One Week of Blood Pressure Readings

Some code for you to type in ...

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

Here is a suggestive demo ...

```
> ( data-for-one-week 110 )
'((113 85) (121 95) (123 97) (133 81) (141 91) (133 101) (135 83))
> ( data-for-one-week 100 )
'((126 88) (106 76) (118 68) (116 82) (114 78) (120 76) (120 74))
> ( data-for-one-week 90 )
'((127 77) (111 63) (111 59) (109 73) (115 69) (111 81) (117 69))
>
```

Generating Several Weeks of Blood Pressure Readings

Some code for you to type in ...

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

Here is a suggestive demo ...

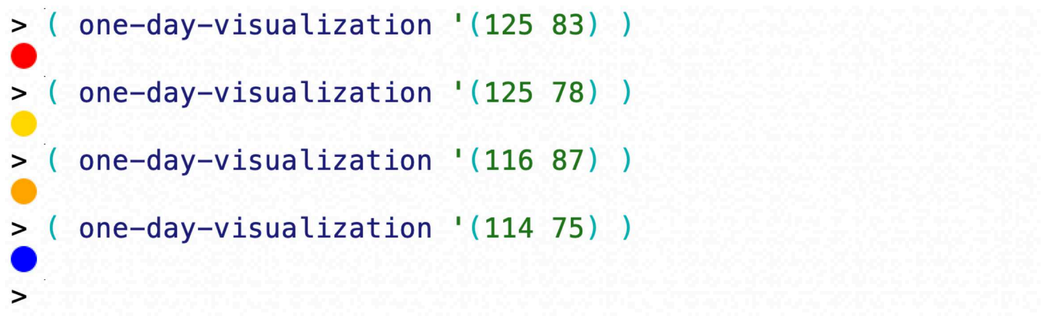
```
> ( define getting-worse '(95 98 100 102 105) )
> ( define getting-better '(105 102 100 98 95) )
> ( generate-data getting-worse )
'(((106 82) (114 76) (120 78) (112 76) (120 70) (116 78) (114 70))
  ((109 75) (115 85) (121 83) (127 77) (111 71) (125 81) (121 91))
  ((126 78) (134 80) (106 84) (120 90) (112 86) (122 88) (126 82))
  ((137 81) (129 89) (133 83) (113 79) (125 77) (121 75) (115 75))
  ((137 75) (125 85) (129 79) (135 79) (119 87) (119 87) (127 71)))
> ( generate-data getting-better )
'(((123 97) (123 81) (131 73) (125 95) (119 95) (127 87) (139 83))
  ((119 75) (127 83) (133 87) (117 83) (125 73) (125 97) (125 95))
  ((118 80) (116 82) (124 84) (110 74) (122 74) (120 72) (120 86))
  ((127 79) (117 75) (125 71) (131 71) (111 89) (133 75) (117 69))
  ((106 76) (106 82) (116 74) (106 68) (104 70) (130 74) (112 78)))
>
```

Visualizaing One Day of BP Data

Please define a function called `one-day-visualization` taking one numeric list of length two representing one BP reading which returns a colored dot of radius 10 according to the following constraints:

- If the first number is greater than or equal to 120 and the second number is greater than or equal to 80, then a red dot is returned.
- If the first number is greater than or equal to 120 and the second number is less than 80, then a gold dot is returned.
- If the first number is less than 120 and the second number is greater than or equal to 80, then an orange dot is returned.
- If the first number is less than 120 and the second number is less than 80, then a blue dot is returned.

Here is a suggestive demo ...

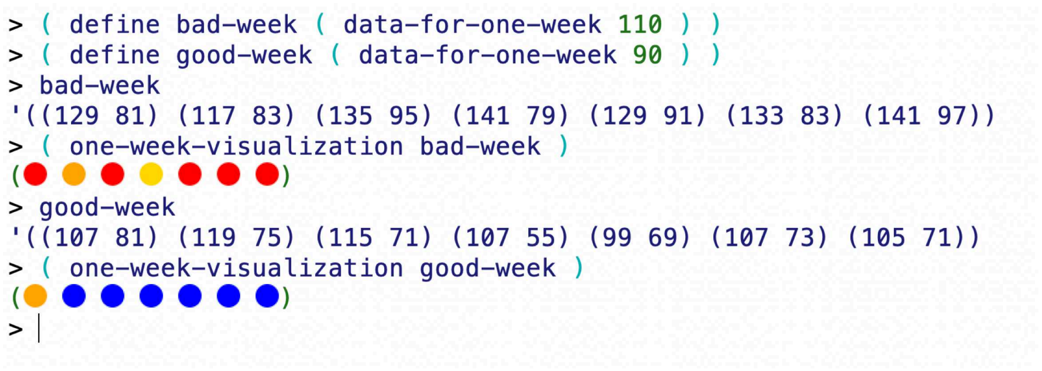


```
> ( one-day-visualization '(125 83) )
●
> ( one-day-visualization '(125 78) )
●
> ( one-day-visualization '(116 87) )
●
> ( one-day-visualization '(114 75) )
●
>
```

Visualizaing One Week of BP Data

Please define a function called `one-week-visualization` taking one list of seven numeric lists of length two, each representing one BP reading, which **displays** a list of seven colored dots corresponding to the seven BP readings, followed by a newline command. **Constraint: Use the `map` function to create the list of seven dots to be displayed.**

Here is a suggestive demo ...

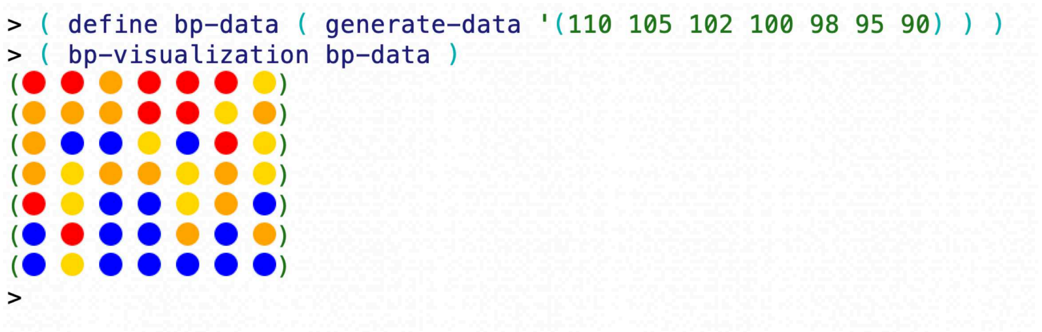


```
> ( define bad-week ( data-for-one-week 110 ) )
> ( define good-week ( data-for-one-week 90 ) )
> bad-week
'((129 81) (117 83) (135 95) (141 79) (129 91) (133 83) (141 97))
> ( one-week-visualization bad-week )
(🔴 🟠 🔴 🟡 🔴 🔴 🔴)
> good-week
'((107 81) (119 75) (115 71) (107 55) (99 69) (107 73) (105 71))
> ( one-week-visualization good-week )
(🟠 🔵 🔵 🔵 🔵 🔵 🔵)
> |
```

Visualizaing Several Weeks of BP Data

Please define a function called `BP-visualization` taking one list of several lists of length seven containing numeric lists of length two, each representing one BP reading, which **displays** the blood pressure readings, one week per line. **Constraint: Use the `map` function to display the successive weeks of BP readings.**

Here is a suggestive demo ...



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

Task

1. Collect the code that I presented and that I asked you to write for this BP visualization task.
2. Generate one demo that features all of the suggested demos for this BP visualization task.
3. Incorporate the collected code and the complete demo into your solution document.

Additional Notes on your Solution Document

Craft a nicely structured solution document that contains:

1. A nice title, indicating that this is your second Racket assignment.
2. A nice learning abstract, which foreshadows the tasks that you are asked to complete in this assignment.
3. A section for each of the 10 problems, complete with the required code and demos, and the question/answer sequence for Problem 5.

Then post your document, in **pdf** format, to you web work site.

Due Date

Friday, April 1, 2022.