

---

## Fourth Racket Programming Assignment Specification

---

---

---

### Learning Abstract

---

This Racket programming assignment I got familiar with the map, filter, and foldr functions. I also learned to use recursion to do a variety of exercises. This assignment is a good way to understand when to use which of the many recourses provided since there are many methods that work but some are often simpler and easier to read.

---

### Task 1 - Generate Uniform List

---

### Specification

---

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

1. The first parameter is presumed to be a nonnegative integer.
2. The second parameter is presumed to be a Lisp object.
3. The value of the function will be a list of length equal to the value of the first parameter containing just that many instances of the value of the second parameter.

---

### Demo

---

```

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

(define (generate-uniform-list length obj)
  (cond
    ((= length 0)
     (list))
    ((> length 0)
     (define myList (append (list obj) (generate-uniform-list (- length 1) obj)))
     myList)
  )
)

Language: racket, with debugging, memory limit: 128 MB.
> ( generate-uniform-list 5 'kitty )
'(kitty kitty kitty kitty kitty)
> ( generate-uniform-list 10 2 )
'(2 2 2 2 2 2 2 2 2 2)
> ( generate-uniform-list 0 'whatever )
'()
> ( generate-uniform-list 2 '(racket prolog haskell rust) )
'((racket prolog haskell rust) (racket prolog haskell rust))
>

```

---

## Task 2 - Association List Generator

---

### 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.

The value of the function will be a list of pairs obtained by “consing” successive elements of the two lists.

---

### Demo

---

```

(define (a-list objList objListPair)
  (cond
    ((> (length objList) 0)
     (cons
      (cons (list-ref objList 0) (list-ref objListPair 0))
      (a-list (cdr objList) (cdr objListPair))
     )
    )
    ((= (length objList) 0)
     '()
    )
  )
)

```

```

> ( 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 3 - 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 third 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

---

```
(define (assoc obj assocList)
  (cond
    ((= (length assocList) 0)
     (list))
    ((eq? obj (car (list-ref assocList 0)))
     (list-ref assocList 0))
    (else
     (assoc obj (cdr assocList)))
  )
)

(define all
  (a-list '(one two three four) '(un deux trois quatre) )
)

(define al2
  (a-list '(one two three) '( (1) (2 2) (3 3 3) ) )
)
```

```
> (assoc 'two all)
'(two . deux)
> (assoc 'five all)
'()
> (assoc 'three al2)
'(three 3 3 3)
> (assoc 'four al2)
'()
> |
```

---

## Task 4 - Rassoc

---

### Specification

---

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

1. The first parameter is presumed to be a lisp object.
  2. The third 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 cdr of the pair equals the value of the first parameter, or `'()` if there is no such element.
-

## Demo

---

```
(define (rassoc obj assocList)
  (cond
    ((= (length assocList) 0)
     (list))
    ((eq? obj (cdr (list-ref assocList 0)))
     (list-ref assocList 0))
    (else
     (rassoc obj (cdr assocList)))
  )
)
```

```
> ( rassoc 'three all )
'()
> ( rassoc 'trois all )
'(three . trois)
> ( rassoc '(1) al2 )
'()
> ( rassoc '(3 3 3) al2 )
'()
> ( rassoc 1 al2 )
'()
>
```

---

## Task 5 - Los->s

---

### Specification

---

Define a **recursive** function called `los->s` according to the following specification:

1. The first and only parameter is presumed to be a list of character strings.
  2. The value of the function will a string containing the strings found in the value of the parameter separated by spaces.
-

## Demo

---

```
(define (los->s stringList)
  (cond
    ((> (length stringList) 1)
     (string-append (list-ref stringList 0) " " (los->s (cdr stringList))))
    ((= (length stringList) 1)
     (string-append (list-ref stringList 0) (los->s (cdr stringList))))
    (else
     ""))
  )
)
```

-----

```
Welcome to DrRacket, version 8.2 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( los->s '( "red" "yellow" "blue" "purple" ) )
"red yellow blue purple"
> ( los->s ( generate-uniform-list 20 "-" ) )
"-----"
> ( los->s '() )
""
> ( los->s '( "whatever" ) )
"whatever"
> |
```

---

## Task 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 (rgb-value) (rgb-value) (rgb-value)))

(define (rgb-value)
  (random 256))

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

```

---

## Demo 1

---

```

(define (generate-list length objFunc)
  (cond
    ((> length 0)
     (cons (objFunc) (generate-list (- length 1) objFunc)))
    (else
     (list))))

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

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

(define (big-dot)
  (circle (+ 40 (random 80)) "solid" (random-color)))

(define (random-color)
  (color (rgb-value) (rgb-value) (rgb-value)))

(define (rgb-value)
  (random 256))

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

(define a (generate-list 5 big-dot))

```

```

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




'(blue red yellow blue blue yellow yellow blue red yellow red blue)
> |




```



## Demo 2


```

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

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


> (sort-dot dots)
  sort-dot: undefined;
cannot reference an identifier before its definition
> (sort-dots dots)

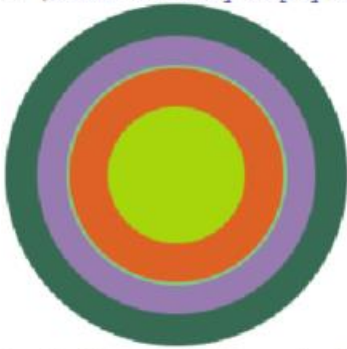
(list   )
> (foldr overlay empty-image (sort-dots dots))


>

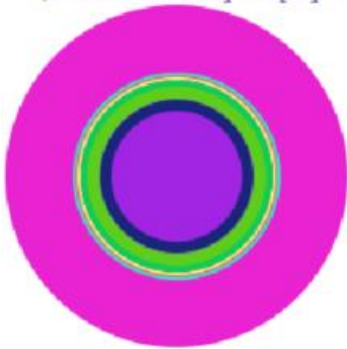
```

## Demo 3

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



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



```
>
```

---

## Task 7 - The Diamond

---

### Specification

---

Using what you learned from Task 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

---

```

(define (diamond length)
  (define a (generate-list length singleDiamond))
  (foldr overlay empty-image (sort-diamonds a))
)

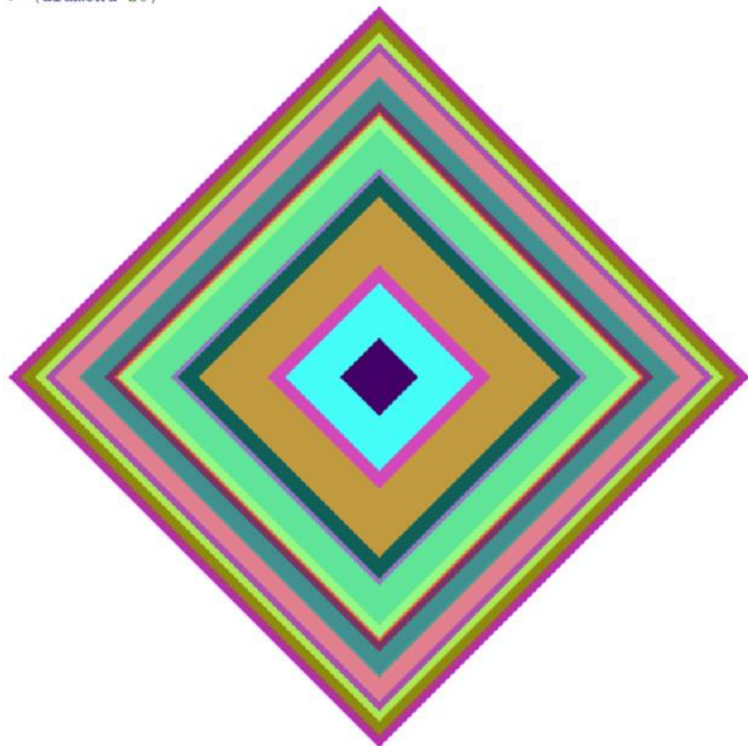
(define (singleDiamond)
  (rotate 45
    (square (+ 40 (random 400)) "solid" (random-color))
  )
)

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

```

## Demo 2

```
> (diamond 20)
```




---

## Task 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 with 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 auxilliary 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

```
(define ( play pitchList )
  (define colorList
    (map (lambda (x) (pc->color x)) pitchList)
  )
  (define boxList
    (map (lambda (x) (color->box x)) colorList)
  )
  (foldr beside empty-image boxList)
)
```

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 9 - Diner

---

### Specification

---

Imagine a diner which has a menu of exactly 6 items. Furthermore, assume the menu is maintained as an association list of item/price pairs. Also, assume that the items sold for a day are maintained as a linear list. With this in mind, define a function called `total` according to the following specification:

1. The first parameter is a linear list of the items sold over some period of time.
2. The second parameter is an item that appears on the menu.
3. The result is the total amount of money collected on the sale of the given item over the period of time.

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

**Hints:** (1) You might want to write a function which takes the name of an item as its sole parameter and returns the price of the item. (2) Lambda functions can be very useful.

---

## Demo

---

```
> menu
'((hamburger . 5.5) (grilledcheese . 4.5) (malt . 3) (coke . 1) (coffe . 1) (pie . 3.5))
> sales
'(hamburger
  coke
  grilledcheese
  malt
  grilledcheese
  coke
  pie
  coffee
  hamburger
  hamburger
  coke
  hamburger
  malt
  hamburger
  malt
  pie
  coffee
  pie
  coffee
  grilledcheese
  malt
  hamburger
  hamburger
  coke
  pie
  coffee
  pie
  coffee
  hamburger
  malt
  hamburger
  malt)
> (total sales 'hamburger)
49.5
> (total sales 'hotdog)
0
> (total sales 'grilledcheese)
13.5
> (total sales 'malt)
18
> (total sales 'coke)
4

> (total sales 'coffee)
5
>
```

---

## Task 10 - Wild Card

---

---

## Specification

---

Numbers can be represented in many ways, the most common is in base ten. This program will take a base 10 int and turn it into any base representation of your choosing from base 2 to base 36

---

## Demo

---

```
(define (baseRep num base)
  (define revolutions (baseRevolutions num base))
  (createNum revolutions num base)
)

(define (createNum revolutionsNeeded num base)
  (append
    (cond
      ((> revolutionsNeeded base)
        (append
          (list (- (- revolutionsNeeded (- revolutionsNeeded base)) 1))
          (list (createNum2 (- revolutionsNeeded base) num base))
        )
      )
      ((< revolutionsNeeded base)
        (list revolutionsNeeded)
      )
      ((= revolutionsNeeded base)
        (- (mod num base) (- base 1))
      )
    )
    ;get final element of the list of Numbers
    (list(list-ref (myNumList num base) (- (length (myNumList num base)) 1)))
  )
)

(define (createNum2 revolutionsNeeded num base)
  (append
    (cond
      ((> revolutionsNeeded base)
        (- (- revolutionsNeeded (- revolutionsNeeded base)) 1)
        (createNum2 (- revolutionsNeeded base) num base)
      )
      ((< revolutionsNeeded base)
        revolutionsNeeded
      )
      ((= revolutionsNeeded base)
        (- (mod num base) (- base 1))
      )
    )
  )
)

; (trace baseRep)
; (trace createNum)
; (trace createNum2)
(define (baseRevolutions num base)
  (define zerosList (filter (lambda (x) (eq? 0 x)) (myNumList num base)))
  (define numOfZeros (length zerosList))
  (- numOfZeros 1)
)
```

```

(define (myNumList num base)
  (reverse (map (lambda (x) (list-ref numbers x)) (repeatingNum num base))))
)

(define (repeatingNum num base)
  (cond
    ((>= num 0)
     (append
      (list (mod num base)) (repeatingNum (- num 1) base)
      )
     )
    ((< num 0)
     '()
     )
  )
)

(define numbers
  '(0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z)
)

(define (mod x y)
  (define dividedNum (/ x y))
  (define decimalRemainder (- dividedNum (floor dividedNum)))
  (define finalNum (* decimalRemainder y))
  finalNum
)

> (baseRep 6 2)
'(1 1 0)
> (baseRep 93 18)
'(5 3)
> (baseRep 102 18)
'(5 c)
> (baseRep 255 16)
'(f f)

```

---