Racket Programming Assignment #3: Lambda and Basic Lisp

Task 1: Lambda

Task 1a - Three Ascending Integers

Demo:

```
> ((lambda (x) (list x (+ x 1) (+ x 2))) 5)
' (5 6 7)
> ((lambda (x) (list x (+ x 1) (+ x 2))) 0)
' (0 1 2)
> ((lambda (x) (list x (+ x 1) (+ x 2))) 108)
' (108 109 110)
```

Task 1b - Make List in Reverse order

Demo:

```
> ((lambda (x y z) (list z y x)) 'red 'yellow 'blue)
' (blue yellow red)
> ((lambda (x y z) (list z y x)) 10 20 30)
' (30 20 10)
> ((lambda (x y z) (list z y x)) "Professor Plum" "Colonel Mustard" "Miss Scarlet")
' ("Miss Scarlet" "Colonel Mustard" "Professor Plum")
```

Task 1c - Random Number Generator:

Demo:

```
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
5
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
5
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
5
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
4
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
3
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
4
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
5
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
3
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
3
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 3 5)
5
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
12
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
15
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
13
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
11
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
13
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
13
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
16
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
11
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
11
> ((lambda (x y) (+ (random (- (+ y 1) x)) x)) 11 17)
17
```

Task 2: List Processing Referencers and Constructors

Demo:

```
> (define colors '(red blue yellow orange))
> colors
'(red blue yellow orange)
> 'colors
'colors
> (quote colors)
'colors
> (car colors)
'red
> (cdr colors)
'(blue yellow orange)
> (car (cdr colors))
'blue
> (cdr (cdr colors))
'(yellow orange)
> (cadr colors)
'blue
> (cddr colors)
'(yellow orange)
> (first colors)
'red
> (second colors)
'blue
> (third colors)
'yellow
> (list-ref colors 2)
'yellow
> (define key-of-c '(c d e))
> (define key-of-g '(g a b))
> (cons key-of-c key-of-g)
'((c d e) g a b)
> (list key-of-c key-of-g)
'((c d e) (g a b))
```

```
> (append key-of-c key-of-g)
'(c d e g a b)
> (define pitches '(do re mi fa so la ti))
> (define animals '(alligator pussycat chimpanzee dog))
> (car (cdr (cdr (cdr animals))))
'dog
> (cadddr pitches)
'fa
> (list-ref pitches 3)
'fa
> (define a 'alligator)
> (define b 'pussycat)
> (define c 'chimpanzee)
> (cons a (cons b (cons c '())))
'(alligator pussycat chimpanzee)
> (list a b c)
'(alligator pussycat chimpanzee)
> (define x '(1 one))
> (define y '(2 two))
> (cons (car x) (cons (car (cdr x)) y))
'(1 one 2 two)
> (append x y)
'(1 one 2 two)
```

Task 3: Little Color Interpreter

Task 3a - Establishing the Sampler code from Lesson 6

Code:

```
#lang racket

(define (sampler)
  (display "(?): ")
  (define the-list (read))
  (define the-element
        (list-ref the-list (random (length the-list)))
  )
  (display the-element) (display "\n")
```

```
  (sampler)
)
```

Demo:

```
> (sampler)
(?): (red orange yellow green blue indigo violet)
red
(?): (red orange yellow green blue indigo violet)
blue
(?): (red orange yellow green blue indigo violet)
yellow
(?): (red orange yellow green blue indigo violet)
yellow
(?): (red orange yellow green blue indigo violet)
orange
(?): (red orange yellow green blue indigo violet)
violet
(?): (aet ate eat eta tae tea)
eta
(?): (aet ate eat eta tae tea)
ate
(?): (aet ate eat eta tae tea)
tea
(?): (aet ate eat eta tae tea)
tea
(?): (aet ate eat eta tae tea)
tae
(?): (aet ate eat eta tae tea)
tea
(?): (0 1 2 3 4 5 6 7 8 9)
3
(?): (0 1 2 3 4 5 6 7 8 9)
7
(?): (0 1 2 3 4 5 6 7 8 9)
4
(?): (0 1 2 3 4 5 6 7 8 9)
0
(?): (0 1 2 3 4 5 6 7 8 9)
6
```

```
(?): (0 1 2 3 4 5 6 7 8 9)
4
(?):  . . user break
```

Task 3b - Color Thing Interpreter

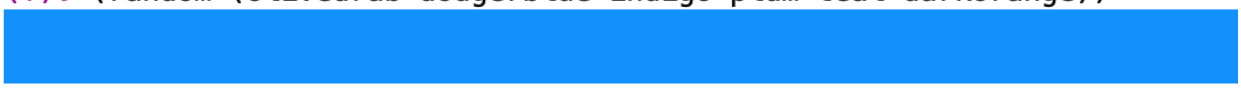
Code:

```
#lang racket

(require 2htdp/image)

(define (rect color)
  (rectangle 500 30 "solid" color)
)

(define (rect-all colors n)
  (display (rect (list-ref colors n)))
  (cond
        ((< n (- (length colors) 1))
        (rect-all colors (+ n 1))
        )
  )
)

(define (color-thing)
  (display "(?): ")
  (define the-list (read))
  (define input (list-ref the-list 0))
  (define colors (list-ref the-list 1))
  (cond
        ((equal? input 'random)
        (display (rect (list-ref colors (random (length colors)))))
        )
        ((equal? input 'all)
        (rect-all colors 0)
        )
        (else
        (display (rect (list-ref colors (- input 1))))
```

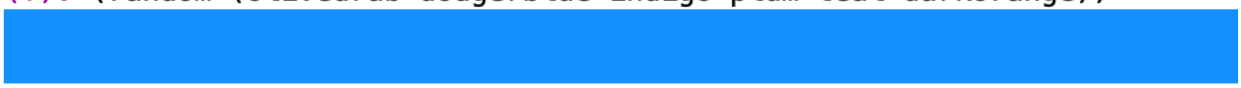
```
        )
  )
  (display "\n")
  (color-thing)
)
```

Demos:

```
> (color-thing)
(?): (random (olivedrab dodgerblue indigo plum teal darkorange))
```

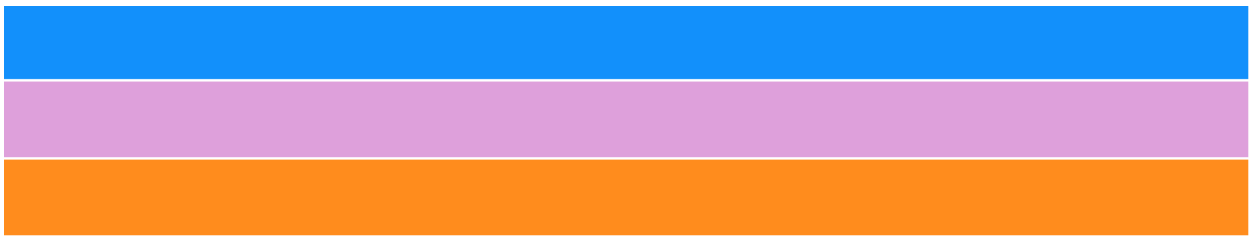


```
(?): (random (olivedrab dodgerblue indigo plum teal darkorange))
```



```
(?): (random (olivedrab dodgerblue indigo plum teal darkorange))
```

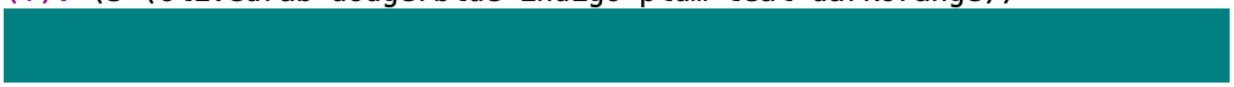
```
(?): (all (olivedrab dodgerblue indigo plum teal darkorange))
```



```
(?): (2 (olivedrab dodgerblue indigo plum teal darkorange))
```

```
(?): (3 (olivedrab dodgerblue indigo plum teal darkorange))
```

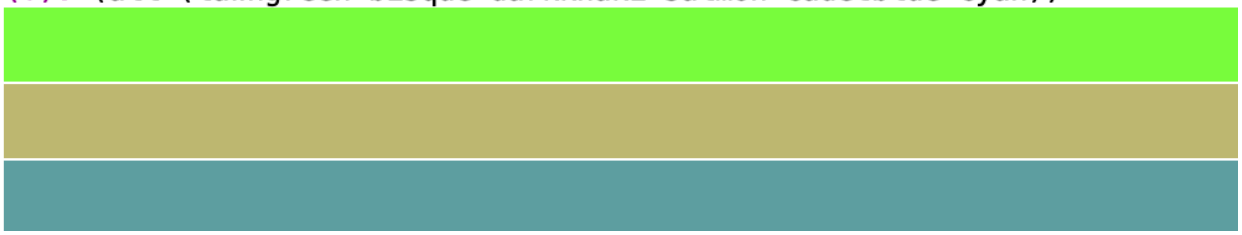
```
(?): (5 (olivedrab dodgerblue indigo plum teal darkorange))
```

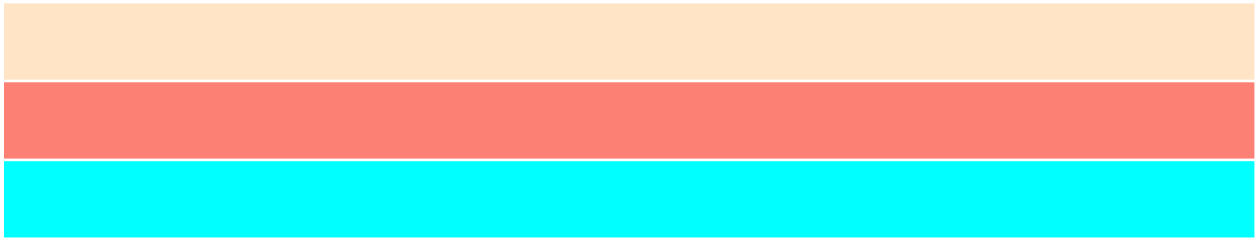


```
(?): (random (lawngreen bisque darkkhaki salmon cadetblue cyan))
```

```
(?): (random (lawngreen bisque darkkhaki salmon cadetblue cyan))
```

```
(?): (random (lawngreen bisque darkkhaki salmon cadetblue cyan))
```

```
(?): (all (lawngreen bisque darkkhaki salmon cadetblue cyan))
```




```
(?): (4 (lawngreen bisque darkkhaki salmon cadetblue cyan))
```

```
(?): (5 (lawngreen bisque darkkhaki salmon cadetblue cyan))
```

```
(?): (2 (lawngreen bisque darkkhaki salmon cadetblue cyan))
```

Task 4: Two Card Poker

Code:

```
#lang racket

(define (ranks rank)
  (list
   (list rank 'C)
   (list rank 'D)
   (list rank 'H)
   (list rank 'S)
  )
)

(define (deck)
  (append
   (ranks 2)
   (ranks 3)
   (ranks 4)
   (ranks 5)
   (ranks 6)
   (ranks 7)
   (ranks 8)
   (ranks 9)
   (ranks 'X)
   (ranks 'J)
   (ranks 'Q)
   (ranks 'K)
   (ranks 'A)
  )
)

(define (pick-a-card)
  (define cards (deck))
  (list-ref (random (length cards)))
)

(define (pick-two-cards)
```

```
  (define cards (deck))
  (define card1 (list-ref cards (random (length cards))))
  (define card2 (list-ref cards (random (length cards))))
  (cond
        ((equal? card1 card2)
        (pick-two-cards)))
  (list card1 card2)
)

(define (number-of-rank rank)
  (cond
        ((equal? rank 2) 1)
        ((equal? rank 3) 2)
        ((equal? rank 4) 3)
        ((equal? rank 5) 4)
        ((equal? rank 6) 5)
        ((equal? rank 7) 6)
        ((equal? rank 8) 7)
        ((equal? rank 9) 8)
        ((equal? rank 'X) 9)
        ((equal? rank 'J) 10)
        ((equal? rank 'Q) 11)
        ((equal? rank 'K) 12)
        ((equal? rank 'A) 13)
  )
)

(define (higher-rank card1 card2)
  (define rank-card1 (number-of-rank(list-ref card1 0)))
  (define rank-card2 (number-of-rank(list-ref card2 0)))
  (cond
        ((> rank-card1 rank-card2) (list-ref card1 0))
        (else
        (list-ref card2 0)
        )
   )
  )

(define (classify-two-cards-ur list)
  (define card1 (list-ref list 0))
```

```
  (define card2 (list-ref list 1))
  (define rank-card1 (number-of-rank (list-ref card1 0)))
  (define rank-card2 (number-of-rank (list-ref card2 0)))
  (display list) (display ": ")
  (define higher (higher-rank card1 card2))
  (display higher) (display " high")
  (cond
        ((or (= rank-card1 (- 1 rank-card2)) (= rank-card2 (- 1 rank-card1))) (display "
straight"))
        ((equal? (list-ref card1 1) (list-ref card2 1)) (display " flush"))
        (= rank-card1 rank-card2) (display " pair")
  )
  (display "\n")
)

(define (show card)
  (display (rank card))
  (display (suit card))
)

(define (rank card)
  (car card)
)

(define (suit card)
  (cadr card)
)

(define (red? card)
  (or
   (equal? (suit card) 'D)
   (equal? (suit card) 'H)
  )
)

(define (black? card)
  (not (red? card))
)

(define (aces? card1 card2)
```

```
  (and
   (equal? (rank card1) 'A)
   (equal? (rank card2) 'A)
  )
)
```

Ur classifier Demo:

```
> (classify-two-cards-ur (pick-two-cards))
((J H) (2 S)): J high
> (classify-two-cards-ur (pick-two-cards))
((4 C) (8 H)): 8 high
> (classify-two-cards-ur (pick-two-cards))
((A C) (9 D)): A high
> (classify-two-cards-ur (pick-two-cards))
((J C) (5 S)): J high
> (classify-two-cards-ur (pick-two-cards))
((4 H) (X C)): X high
> (classify-two-cards-ur (pick-two-cards))
((8 H) (9 C)): 9 high
> (classify-two-cards-ur (pick-two-cards))
((7 H) (9 D)): 9 high
> (classify-two-cards-ur (pick-two-cards))
((7 H) (8 H)): 8 high flush
> (classify-two-cards-ur (pick-two-cards))
((Q D) (5 C)): Q high
> (classify-two-cards-ur (pick-two-cards))
((X S) (Q D)): Q high
> (classify-two-cards-ur (pick-two-cards))
((Q H) (7 S)): Q high
> (classify-two-cards-ur (pick-two-cards))
((6 D) (3 D)): 6 high flush
> (classify-two-cards-ur (pick-two-cards))
((5 D) (8 D)): 8 high flush
> (classify-two-cards-ur (pick-two-cards))
((9 C) (J D)): J high
> (classify-two-cards-ur (pick-two-cards))
((3 H) (6 H)): 6 high flush
> (classify-two-cards-ur (pick-two-cards))
((6 S) (K H)): K high
```

```
> (classify-two-cards-ur (pick-two-cards))
((Q S) (6 D)): Q high
> (classify-two-cards-ur (pick-two-cards))
((8 C) (9 S)): 9 high
> (classify-two-cards-ur (pick-two-cards))
((9 C) (Q H)): Q high
```

Pick two cards Demo:

```
> (pick-two-cards)
'((9 H) (9 S))
> (pick-two-cards)
'((K C) (9 D))
> (pick-two-cards)
'((2 H) (A S))
> (pick-two-cards)
'((8 D) (A D))
> (pick-two-cards)
'((6 C) (X H))
```

Higher rank Demo:

```
> (higher-rank (pick-a-card) (pick-a-card))
7
> (higher-rank (pick-a-card) (pick-a-card))
9
> (higher-rank (pick-a-card) (pick-a-card))
'K
> (higher-rank (pick-a-card) (pick-a-card))
'K
> (higher-rank (pick-a-card) (pick-a-card))
'X
```