Matilda Knapp **3/3/2022**
CSC344 **Racket**

Racket Programming Assignment #3: Lambda and Basic Lisp

Learning Abstract

Within this assignment I work with lists and lambda functions. In the first task I am mainly using lambda to generate simple lists. The first lambda function generates three integers in ascending order, based on the start integer which is inputted into the function. The second lambda function reverses the order of a simple list of three values, and the third generates a random number between or equal to two chosen numbers. In the second task I am referencing pieces of a list using car, cdr and similar commands. For example, for a list x defined as '(apple orange), (x car) would be equal to 'apple. This task is based off of a demo in racket lesson 6. Following this there is the third task which is a color interpreter, unlike the first two tasks this requires me to mess with actual code. The color interpreter task has two parts, one where I generate a demo from the sampler code, and the second where I change the sampler code to generate blocks of color from a list, this is the color-thing code. I added three functions to the color-thing code, one to pick a number location in the list to get a particular color, one to randomly pick a color from the list, and one which lists all colors in the list, when given the word all. Lastly in task 4 I work with a separate piece of code which generates a list of cards. This code I add several functions to, one which picks two cards with different ranks from the deck, another which compares these cards, and two more which use both these functions. I thought this task was rather interesting since it's based on poker, and I've never actually played poker, so I learned a bit.

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

```
14
> ( ( lambda ( x y ) ( random x ( + y 1 ) ) ) 11 17 )
17
> ( ( lambda ( x y ) ( random x ( + y 1 ) ) ) 11 17 )
14
> ( ( lambda ( x y ) ( random x ( + y 1 ) ) ) 11 17 )
15
> ( ( lambda ( x y ) ( random x ( + y 1 ) ) ) 11 17 )
16
> ( ( lambda ( x y ) ( random x ( + y 1 ) ) ) 11 17 )
12
> ( ( lambda ( x y ) ( random x ( + y 1 ) ) ) 11 17 )
13
> ( ( lambda ( x y ) ( random x ( + y 1 ) ) ) 11 17 )
16
```

Task 2 - List Processing Referencers and Constructors

Task 2 (DEMO)

```
> ( define languages '(racket prolog haskell rust) )
> languages
'(racket prolog haskell rust)
> 'languages
'languages
> ( quote languages )
'languages
> ( car languages )
'racket
> ( cdr languages )
'(prolog haskell rust)
> ( car ( cdr languages ) )
'prolog
> ( cdr ( cdr languages ) )
'(haskell rust)
> ( cadr languages )
'prolog
```

```
> ( cddr languages )
'(haskell rust)
> ( first languages )
'racket
> ( second languages )
'prolog
> ( third languages )
'haskell
> ( list-ref languages 2 )
'haskell
> ( define numbers '(1 2 3) )
> ( define letters '(a b c) )
> ( cons numbers letters )
'((1 2 3) a b c)
> ( list numbers letters )
'((1 2 3) (a b c))
> ( append numbers letters )
'(1 2 3 a b c)
> ( define animals '(ant bat cat dot eel) )
> ( car ( cdr ( cdr ( cdr animals ) ) ) )
'dot
> ( cadddr animals )
'dot
> ( list-ref animals 3 )
'dot
> ( define a 'apple )
> ( define b 'peach )
> ( define c 'cherry )
> ( cons a ( cons b ( cons c '() ) ) )
'(apple peach cherry)
> ( list a b c )
'(apple peach cherry)
> ( define x '(one fish) )
> ( define y '(two fish) )
> ( cons ( car x ) ( cons ( car ( cdr x ) ) y ) )
'(one fish two fish)
> ( append x y )
'(one fish two fish)
```

Task 3 - Little Color Interpreter

Task 3a - Establishing the Sampler code (DEMO)

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

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

Task 3a - Samper (CODE)

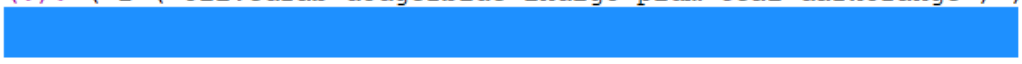
```
#lang racket
( define ( sampler )
   ( display "(?): " )
   ( define the-list ( read ) )
   ( define the-element
      ( list-ref the-list ( random ( length the-list ) ) )
      )
   ( display the-element ) ( display "\n" )
   ( sampler )
   )
```

Task 3b - Color Thing Interpreter (DEMO)

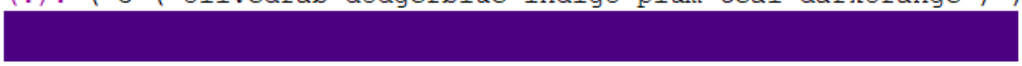
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.

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

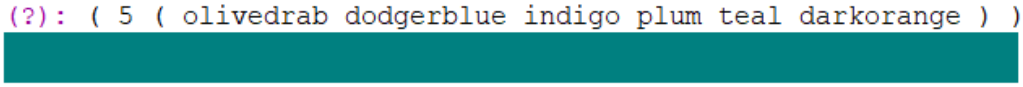
```
(?): ( 2 ( olivedrab dodgerblue indigo plum teal darkorange ) )
```



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

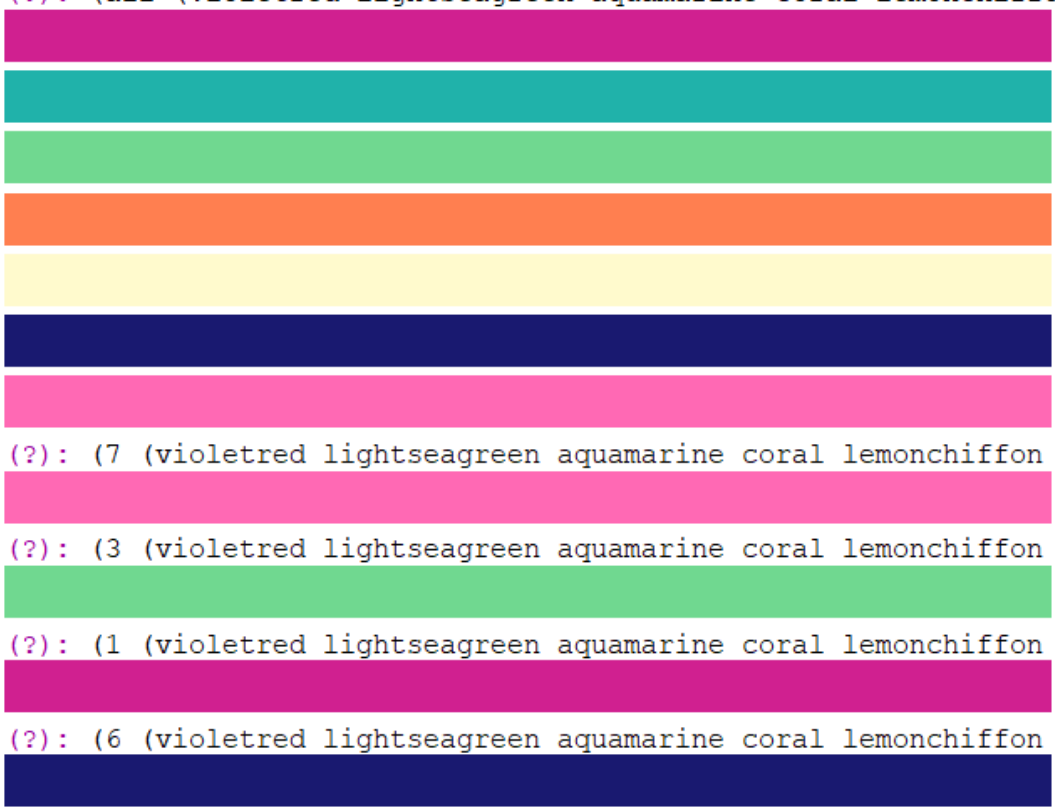


```
(?): ( 5 ( olivedrab dodgerblue indigo plum teal darkorange ) )
```



Task 3b (Extension) - My Colors (DEMO)

```
> ( color-thing )
(?): (all (violetred lightseagreen aquamarine coral lemonchiffon midnightblue hotpink) )
```



```
(?): (7 (violetred lightseagreen aquamarine coral lemonchiffon midnightblue hotpink) )
```



```
(?): (3 (violetred lightseagreen aquamarine coral lemonchiffon midnightblue hotpink) )
```



```
(?): (1 (violetred lightseagreen aquamarine coral lemonchiffon midnightblue hotpink) )
```



```
(?): (6 (violetred lightseagreen aquamarine coral lemonchiffon midnightblue hotpink) )
```


*Note “I may have went a bit overboard with the list length”

Task 3a - Color-Thing (CODE)

```
#lang racket
(require 2htdp/image)
( define ( color-thing )
   ( display "(?): " )
   ( define the-list ( read ) )
   ( define the-element
      ( car the-list  )
      )
   (define color-list (car ( cdr the-list )
```

```
                          )
     )
   ( define i -1)
   ( cond ( (eq? the-element 'all )
            (for ([length color-list])
              ( set! i (+ i 1 ))
              ( define rec ( rectangle 500 25 "solid" ( list-ref
color-list i) ) )
              (display rec)  ( display "\n" )
              )
            )
          ( (eq? the-element 'random )
            ( define rec ( rectangle 500 25 "solid" ( list-ref
color-list (random (length color-list)) ) ) )
            (display rec)  ( display "\n" )
            )
          ( (number? the-element)
            ( define rec ( rectangle 500 25 "solid" ( list-ref
color-list (- the-element 1) ) ) )
            (display rec)  ( display "\n" )
            )
          )
   ( color-thing )
   )
```

Task 4 - Two Card Poker

Task 4a - Card Code from Lesson 6 (DEMO)

```
> ( define c1 '(7 C ) )
> ( define c2 '(Q H ) )
> c1
'(7 C)
> c2
'(Q H)
> (rank c1)
7
```

```
> (suit c1)
'C
> (rank c2)
'Q
> (suit c2)
'H
> (red? c1)
#f
> (red? c2)
#t
> (black? c1)
#t
> (black? c2)
#f
> (aces? '(A C) '(A S) )
#t
> (aces? '(K S) '(A C) )
#f
> ( ranks 4 )
'((4 C) (4 D) (4 H) (4 S))
> ( ranks 'K )
'((K C) (K D) (K H) (K S))
> ( length ( deck ) )
52
> ( display ( deck ) )
((2 C) (2 D) (2 H) (2 S) (3 C) (3 D) (3 H) (3 S) (4 C) (4 D) (4 H) (4
S) (5 C) (5 D) (5 H) (5 S) (6 C) (6 D) (6 H) (6 S) (7 C) (7 D) (7 H)
(7 S) (8 C) (8 D) (8 H) (8 S) (9 C) (9 D) (9 H) (9 S) (X C) (X D) (X
H) (X S) (J C) (J D) (J H) (J S) (Q C) (Q D) (Q H) (Q S) (K C) (K D)
(K H) (K S) (A C) (A D) (A H) (A S))
> (pick-a-card)
'(Q D)
> (pick-a-card)
'(A H)
> (pick-a-card)
'(2 S)
> (pick-a-card)
'(4 H)
> (pick-a-card)
'(4 S)
```

```
> (pick-a-card)
'(5 S)
```

Task 4a - Card Code from Lesson 6 (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 cards ( random ( length cards ) ) )
   )
( 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 )
     )
   )
```

Task 4b - Two Card Poker Classifier, IR Version

Pick Two Cards (DEMO)

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

Higher Rank (DEMO)

```
>  ( higher-rank ( pick-a-card ) ( pick-a-card ) )
```

```
>(higher-rank '(J H) '(6 H))
<'(J)
'(J)
>  ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(K H) '(X D))
<'(K)
'(K)
>  ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(9 S) '(5 C))
<'(9)
'(9)
>  ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(A S) '(7 H))
<'(A)
'(A)
>  ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(5 D) '(K H))
<'(K)
'(K)
```

UR classifier (DEMO)

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

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

Task 4b (CODE)

```
#lang racket
( require racket/trace )
( 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 cards ( random ( length cards ) ) )
   )
( 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 )
     )
   )
```

```
( define (pick-two-cards)
   ( define card-1
      (pick-a-card)
      )
   ( define card-2
      (pick-a-card)
      )
   ( cond ((eq? (rank card-1) (rank card-2))
          (pick-two-cards)
          ) (else
             (list card-1 card-2 )
             )
            )
   )
(define (convert-rank card-a)
  ( cond
     ((eq? (rank card-a) 2)
      2
      )
     ((eq? (rank card-a) 3)
      3
      )
     ((eq? (rank card-a) 4)
      4
      )
     ((eq? (rank card-a) 5)
      5
      )
     ((eq? (rank card-a) 6)
      6
      )
     ((eq? (rank card-a) 7)
      7
      )
     ((eq? (rank card-a) 8)
      8
      )
     ((eq? (rank card-a) 9)
      9
      )
```

```
      ((eq? (rank card-a) 'X)
       10
       )
      ((eq? (rank card-a) 'J)
       11
       )
      ((eq? (rank card-a) 'Q)
       12
       )
      ((eq? (rank card-a) 'K)
       13
       )
      ((eq? (rank card-a) 'A)
       14
       )

      )
  )

( define (check-flush card-1 card-2)
   (cond ((eq? (suit card-1) (suit card-2))
          (display " flush" )
          )
         )
   )
( define (check-straight card-1 card-2)
   (cond ((eq? (+ (convert-rank card-1) 1) (convert-rank card-2))
          (display " straight" )
          )
         ((eq? (convert-rank card-1) (+ (convert-rank card-2) 1))
          (display " straight" )
          )
         )
   )

( define (higher-rank card-1 card-2)
   ( cond
      ( (> (convert-rank card-1) (convert-rank card-2))
        ( list (rank card-1))
        ) (else
```

```
          ( list (rank card-2))
          )
         )
   )
( define (classify-two-cards-ur lst)
   (define first (list-ref lst 0))
   (define last (list-ref lst 1))
   (define res (list-ref (higher-rank first last) 0) )
   (display (list first last)) (display ":")
   ( cond
      ((eq? res '3)
       (display "  ") (display '3) (display " high")
       )
      ((eq? res '4)
       (display "  ") (display '4) (display " high")
       )
      ((eq? res '5)
       (display "  ") (display '5) (display " high")
       )
      ((eq? res '6)
       (display "  ") (display '6) (display " high")
       )
      ((eq? res '7)
       (display "  ") (display '7) (display " high")
       )
      ((eq? res '8)
       (display "  ") (display '8) (display " high")
       )
      ((eq? res '9)
       (display "  ") (display '9) (display " high")
       )
      ((eq? res 'X)
       (display "  ") (display 'X) (display " high")
       )
      ((eq? res 'J)
       (display "  ") (display 'J) (display " high")
       )
      ((eq? res 'Q)
       (display "  ") (display 'Q) (display " high")
       )
```

```
      ((eq? res 'K)
       (display "  ") (display 'K) (display " high")
       )
      ((eq? res 'A)
       (display "  ") (display 'A) (display " high")
       )
      )
   (check-straight first last)
   (check-flush first last)
   )

;(classify-two-cards-ur (pick-two-cards))
;I added this here to help me call the function easier.

;( trace pick-two-cards )
```

Task 4c - Two Card Poker Classifier (DEMO)

```
> (classify-two-cards (pick-two-cards))
((3 C) (X H)):  ten high
> (classify-two-cards (pick-two-cards))
((7 S) (Q H)):  queen high
> (classify-two-cards (pick-two-cards))
((J C) (Q C)):  queen high straight flush
> (classify-two-cards (pick-two-cards))
((J H) (2 C)):  jack high
> (classify-two-cards (pick-two-cards))
((K H) (4 H)):  king high flush
> (classify-two-cards (pick-two-cards))
((9 D) (7 S)):  nine high
> (classify-two-cards (pick-two-cards))
((Q S) (4 S)):  queen high flush
> (classify-two-cards (pick-two-cards))
((2 D) (5 H)):  five high
> (classify-two-cards (pick-two-cards))
((5 H) (X D)):  ten high
> (classify-two-cards (pick-two-cards))
((K D) (Q D)):  king high straight flush
> (classify-two-cards (pick-two-cards))
```

```
((7 C) (4 D)):  seven high
> (classify-two-cards (pick-two-cards))
((7 D) (6 S)):  seven high straight
> (classify-two-cards (pick-two-cards))
((3 S) (6 D)):  six high
> (classify-two-cards (pick-two-cards))
((A D) (5 D)):  ace high flush
> (classify-two-cards (pick-two-cards))
((7 C) (J S)):  jack high
> (classify-two-cards (pick-two-cards))
((4 H) (X H)):  ten high flush
> (classify-two-cards (pick-two-cards))
((8 C) (2 H)):  eight high
> (classify-two-cards (pick-two-cards))
((A D) (4 H)):  ace high
> (classify-two-cards (pick-two-cards))
((6 D) (K C)):  king high
> (classify-two-cards (pick-two-cards))
((6 D) (K D)):  king high flush
```

Task 4c - Two Card Poker Classifier (CODE)

```
#lang racket
( require racket/trace )
( 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 cards ( random ( length cards ) ) )
   )
( 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 )
     )
   )
( define (pick-two-cards)
   ( define card-1
```

```
      (pick-a-card)
      )
   ( define card-2
      (pick-a-card)
      )
   ( cond ((eq? (rank card-1) (rank card-2))
          (pick-two-cards)
          ) (else
             (list card-1 card-2 )
             )
            )
   )
(define (convert-rank card-a)
  ( cond
     ((eq? (rank card-a) 2)
      2
      )
     ((eq? (rank card-a) 3)
      3
      )
     ((eq? (rank card-a) 4)
      4
      )
     ((eq? (rank card-a) 5)
      5
      )
     ((eq? (rank card-a) 6)
      6
      )
     ((eq? (rank card-a) 7)
      7
      )
     ((eq? (rank card-a) 8)
      8
      )
     ((eq? (rank card-a) 9)
      9
      )
     ((eq? (rank card-a) 'X)
      10
```

```
      )
     ((eq? (rank card-a) 'J)
      11
      )
     ((eq? (rank card-a) 'Q)
      12
      )
     ((eq? (rank card-a) 'K)
      13
      )
     ((eq? (rank card-a) 'A)
      14
      )

     )
  )

( define (check-flush card-1 card-2)
   (cond ((eq? (suit card-1) (suit card-2))
          (display " flush" )
          )
         )
   )
( define (check-straight card-1 card-2)
   (cond ((eq? (+ (convert-rank card-1) 1) (convert-rank card-2))
          (display " straight" )
          )
         ((eq? (convert-rank card-1) (+ (convert-rank card-2) 1))
          (display " straight" )
          )
         )
   )

( define (higher-rank card-1 card-2)
   ( cond
      ( (> (convert-rank card-1) (convert-rank card-2))
        ( list (rank card-1))
        ) (else
           ( list (rank card-2))
           )
```

```
          )
   )
( define (classify-two-cards lst)
   (define first (list-ref lst 0))
   (define last (list-ref lst 1))
   (define res (list-ref (higher-rank first last) 0) )
   (display (list first last)) (display ":")
   ( cond
      ((eq? res '3)
       (display "  ") (display "three") (display " high")
       )
      ((eq? res '4)
       (display "  ") (display "four") (display " high")
       )
      ((eq? res '5)
       (display "  ") (display "five") (display " high")
       )
      ((eq? res '6)
       (display "  ") (display "six") (display " high")
       )
      ((eq? res '7)
       (display "  ") (display "seven") (display " high")
       )
      ((eq? res '8)
       (display "  ") (display "eight") (display " high")
       )
      ((eq? res '9)
       (display "  ") (display "nine") (display " high")
       )
      ((eq? res 'X)
       (display "  ") (display "ten") (display " high")
       )
      ((eq? res 'J)
       (display "  ") (display "jack") (display " high")
       )
      ((eq? res 'Q)
       (display "  ") (display "queen") (display " high")
       )
      ((eq? res 'K)
       (display "  ") (display "king") (display " high")
```

```
    )
   ((eq? res 'A)
    (display "  ") (display "ace") (display " high")
    )
   )
(check-straight first last)
(check-flush first last)
)
```