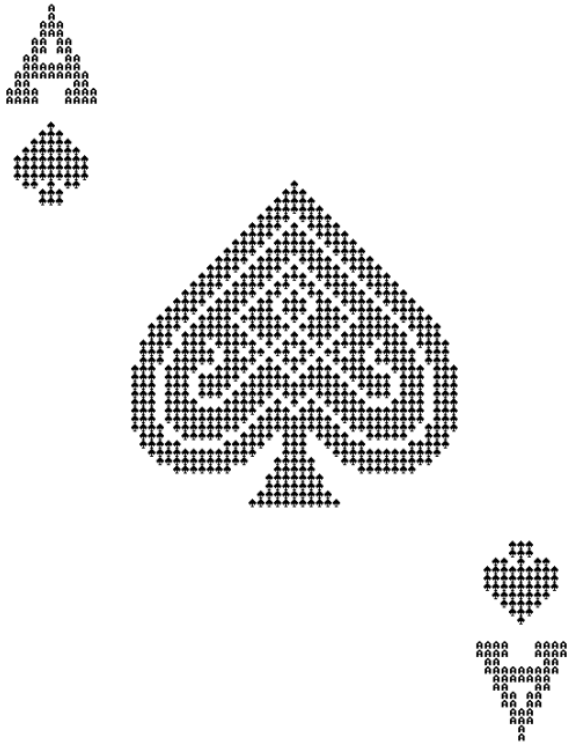
Racket Programming Assignment #3: Lambda and Basic Lisp

By: Miguel Cruz



Abstract

Within this assignment a series of four tasks were done featuring concepts pertaining to lambda functions, basic list processing operations in Lisp, and extending programs that are presented in Lesson 6 "Basic Lisp Programming".

Task 1 - Lambda

Task 1a - Three ascending integers

Consider the following demo:

```
> ( asc 5 )
'(5 6 7)
> ( asc 0 )
'(0 1 2)
> ( asc 108 )
'(108 109 110)
>
```

Your job is to generate this exact demo, except that you must replace the call to the `asc` function in each of the three applications with a **lambda** function. Note that the Definitions area will not come into play in this exercise, nor will you create any named functions. Your demo will simply feature three anonymous function applications, the first with argument 5, the second with argument 0, and the third with argument 108.

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ((lambda (n)
  (let ((n1 n)
        (n2 n))
    (set! n1 (+ n1 1))
    (set! n2 (+ n2 2))
    (list n n1 n2)))5)
'(5 6 7)
> ((lambda (n)
  (let ((n1 n)
        (n2 n))
    (set! n1 (+ n1 1))
    (set! n2 (+ n2 2))
    (list n n1 n2)))0)
'(0 1 2)
> ((lambda (n)
  (let ((n1 n)
        (n2 n))
    (set! n1 (+ n1 1))
    (set! n2 (+ n2 2))
    (list n n1 n2)))108)
'(108 109 110)
>
```

Task 1b - Make list in reverse order

Consider the following demo:

```
> ( mlr 'red 'yellow 'blue )
'(blue yellow red)
> ( mlr 10 20 30 )
'(30 20 10)
> ( mlr "Professor Plum" "Colonel Mustard" "Miss Scarlet" )
'("Miss Scarlet" "Colonel Mustard" "Professor Plum")
```

Your job is to generate this exact demo, except that you must replace the call to the `mlr` function in each of the three applications with a **lambda** function. Note that the Definitions area will not come into play in this exercise, nor will you create any named functions. Your demo will simply feature three anonymous function applications, each using the same three arguments that I used in the given demo.

Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.

```
> ((lambda (a b c)
  (let ((a1 a)
        (a2 b)
        (a3 c))
    (list a3 a2 a1))) 'red 'yellow 'blue)

'(blue yellow red)
> ((lambda (a b c)
  (let ((a1 a)
        (a2 b)
        (a3 c))
    (list a3 a2 a1))) 10 20 30)
'(30 20 10)
> ((lambda (a b c)
  (let ((a1 a)
        (a2 b)
        (a3 c))
    (list a3 a2 a1))) "Professor Plum" "Colonel Mustard" "Miss Scarlet")

'("Miss Scarlet" "Colonel Mustard" "Professor Plum")
```

Task 1c - Random number generator

Consider the following demo:

```
> ( rn 3 5 )
5
> ( rn 3 5 )
3
> ( rn 3 5 )
5
> ( rn 3 5 )
5
> ( rn 3 5 )
3
> ( rn 3 5 )
4
> ( rn 3 5 )
3
> ( rn 3 5 )
3
> ( rn 3 5 )
5
> ( rn 3 5 )
3
> ( rn 11 17 )
17
> ( rn 11 17 )
12
> ( rn 11 17 )
14
> ( rn 11 17 )
12
> ( rn 11 17 )
12
> ( rn 11 17 )
14
> ( rn 11 17 )
16
> ( rn 11 17 )
14
> ( rn 11 17 )
16
> ( rn 11 17 )
13
```

Your job is to generate demo like this one, except that you must replace the call to the `rn` function in each of the three applications with a **lambda** function. Note that the Definitions area will not come into play in this exercise, nor will you create any named functions. Your demo will simply feature ten anonymous function applications with arguments 3 and 5, and ten anonymous function applications with arguments 11 and 17.

Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.

```
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
3
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
4
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
4
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
4
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
4
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
5
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
3
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
3
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
3
> ((lambda (a b)
   (random a (+ b 1))) 3 5)
4
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
11
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
11
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
11
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
14
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
16
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
15
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
17
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
17
> ((lambda (a b)
   (random a (+ b 1))) 11 17)

15
> ((lambda (a b)
   (random a (+ b 1))) 11 17)
17
>
```

Task 2 - List Processing Referencers and Constructors

Simply create the demo, for real, that is presented in redacted form in Lesson 6 “Basic Lisp Programming”, in the section titled “Redacted Racket Session Featuring Referencers and Constructors”.

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define languages '(racket prolog haskell rust))
> languages
'(racket prolog haskell rust)
> ( 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 b ( cons c ' () ) )
'(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)
>
```

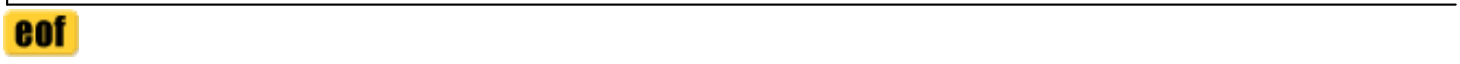
Task 3 - Little Color Interpreter

This task invites you to extend a very simple interpreter into a slightly less simple interpreter.

Task 3a - Establishing the Sampler code from Lesson 6

1. In a file called `sampler.rkt`, establish the "sampler" program that can be found in the penultimate section of Lesson 6.
2. Generate a demo by mimicking the demo which accompanies the "sampler" program in Lesson 6.

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (sampler)
(?): (red orange yellow green blue indigo violet)
indigo
(?): (red orange yellow green blue indigo violet)
yellow
(?): (red orange yellow green blue indigo violet)
indigo
(?): (red orange yellow green blue indigo violet)
blue
(?): (red orange yellow green blue indigo violet)
violet
(?): (red orange yellow green blue indigo violet)
orange
(?):
(aet ate eat eta tae tea)
aet
(?): (aet ate eat eta tae tea)
tae
(?): (aet ate eat eta tae tea)
aet
(?): (aet ate eat eta tae tea)
tea
(?): (aet ate eat eta tae tea)
eat
(?): (aet ate eat eta tae tea)
ate
(?): (0 1 2 3 4 5 6 7 8 9)
1
(?): (0 1 2 3 4 5 6 7 8 9)
3
(?): (0 1 2 3 4 5 6 7 8 9)
3
(?): (0 1 2 3 4 5 6 7 8 9)
0
(?): (0 1 2 3 4 5 6 7 8 9)
8
(?): (0 1 2 3 4 5 6 7 8 9)
4
(?):
```



Task 3b - Color Thing Interpreter

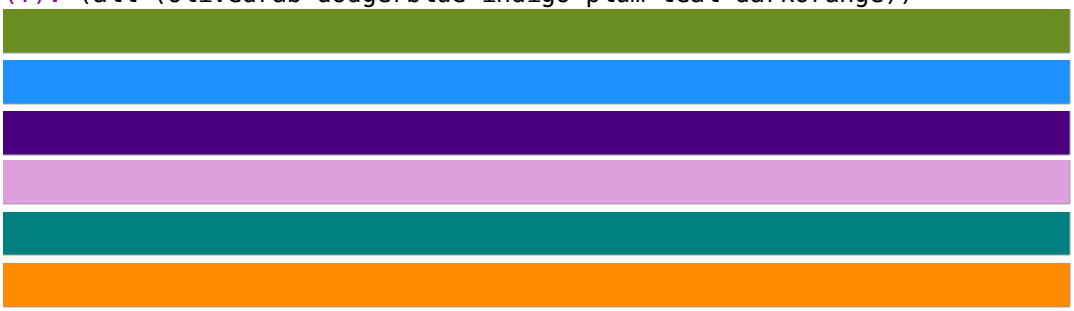
1. In a file called `color_thing.rkt`, copy and paste the code from the `sampler.rkt` file. In the two obvious places, change `sampler` to `color_thing`. Then run the program, just to make sure that everything still works.
2. Change/extend the program so that it will interpret three commands, each of which takes the form of a list of length 2, as suggested in the accompanying demo. You needn't do error checking for this program. Simply assume that the user capably types in a list of length two when required, the first element of which is either `random`, or `all`, or an integer between 1 and the length of the second parameter, which will always be a list of color names that the Racket `2htdp/image` library recognizes. **Constraint: No iterative constructs are allowed.**
3. Create a demo that is just like the accompanying demo, except for the randomness bits.
4. Create another demo, a similar demo, but with different colors than in the accompanying demo. You can find a list of colors known to Racket at:
`https://docs.racket-lang.org/draw/color-database___.html`

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (color-thing)
(?): (random (olivedrab dodgerblue indigo plum teal darkorange))
```

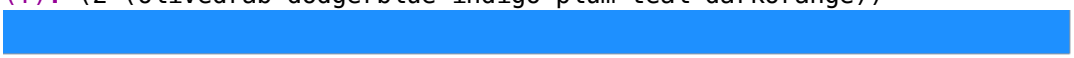
```
(?): (random (olivedrab dodgerblue indigo plum teal darkorange))
```

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

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

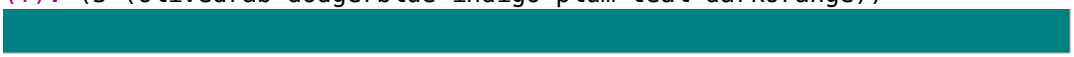


```
(?): (2 (olivedrab dodgerblue indigo plum teal darkorange))
```



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

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



```
(?):
```

eof

Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.

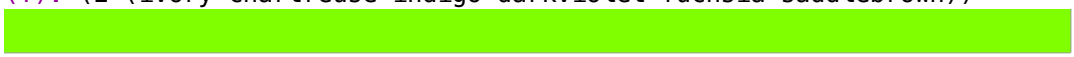
```
> (color-thing)
(?): (random (ivory chartreuse indigo darkviolet fuchsia saddlebrown))
```

```
(?): (random (ivory chartreuse indigo darkviolet fuchsia saddlebrown))
```

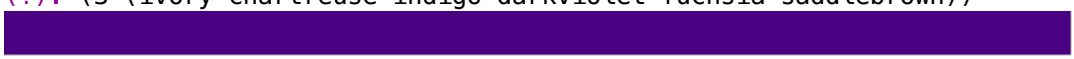
```
(?): (random (ivory chartreuse indigo darkviolet fuchsia saddlebrown))
```

```
(?): (all (ivory chartreuse indigo darkviolet fuchsia saddlebrown))
```

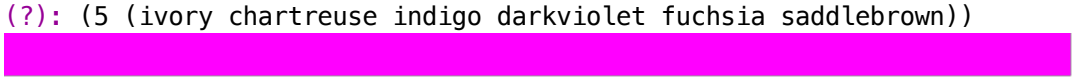
```
(?): (2 (ivory chartreuse indigo darkviolet fuchsia saddlebrown))
```



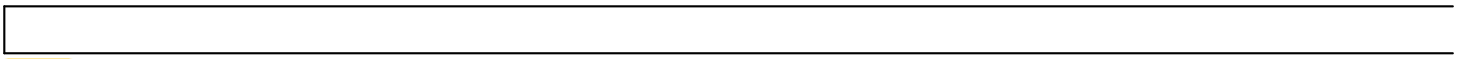
```
(?): (3 (ivory chartreuse indigo darkviolet fuchsia saddlebrown))
```



```
(?): (5 (ivory chartreuse indigo darkviolet fuchsia saddlebrown))
```



```
(?):
```



eof

```
#lang racket

(require 2htdp/image)
(require racket/trace)
;---------
; Programming Assignment 3: Little Color Interpreter

;; Task 3a: Establishing the Sampler code from Lesson 6 (Sampler.rkt)
( 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


( define ( color-thing )
   ( display "(?): " )
   ( define input ( read ) )
   ( define cmd (car input))
   ( define the-list (car (cdr input)))

   (cond
     ((equal? cmd 'random)
      (define result (random (length the-list )))
      (display (rectangle 500 20 "solid" ( list-ref the-list result)))
      (display "\n")
      )
     ((equal? cmd 'all )
      (all the-list)
      )
     (else
      (display (rectangle 500 20 "solid" (list-ref the-list (- cmd 1))))
      (display "\n")
      )
     )
   (color-thing)
   )


(define (all color-list)
  (cond
    ((equal? (length color-list) 0)
        (display "")
        )
    (not (empty? color-list)
         (display (rectangle 500 20 "solid" (car color-list)))
         (display "\n")
         (all (cdr color-list))
         )
    )
  )
```

Task 4 - Two Card Poker

This task invites you to programmably explore the hands that are dealt in a game of two card poker. You will start by merely entering and demoing the card playing code presented in Lesson 6 “Basics Lisp Programming”. You will then write a classifier for the hands in this game, assuming a uniform representational (ur) scheme (output of cards will be represented in the primitive internall representation of cards. As a final step, you will arrange for the output to appear in a more natural fashion.

Task 4a - Establishing the Card code from Lesson 6

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (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? '(A S) '(A C))
#t
> (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)
'(7 D)
> (pick-a-card)
'(3 H)
> (pick-a-card)
'(2 H)
> (pick-a-card)
'(A S)
> (pick-a-card)
'(X H)
> (pick-a-card)
'(7 D)
>
```

Task 4b - Two Card Poker Classifier, IR Version

UR classifier demo

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (pick-two-cards)
'((8 D) (K D))
> (pick-two-cards)
'((4 D) (J C))
> (pick-two-cards)
'((Q S) (K H))
> (pick-two-cards)
'((K C) (Q H))
> (pick-two-cards)
'((2 C) (7 S))
> (pick-two-cards)
'((J C) (K D))
> (higher-rank (pick-a-card) (pick-a-card))
>(higher-rank '(X H) '(9 C))
<'X
'X
> (higher-rank (pick-a-card) (pick-a-card))
>(higher-rank '(5 S) '(9 C))
<9
9
> (higher-rank (pick-a-card) (pick-a-card))
>(higher-rank '(2 D) '(2 D))
<#<void>
> (higher-rank (pick-a-card) (pick-a-card))
>(higher-rank '(3 D) '(3 H))
<#<void>
> (higher-rank (pick-a-card) (pick-a-card))
>(higher-rank '(A H) '(5 D))
<'A
'A
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(X S) '(X H))
<#<void>
((X S) (X H)): #<void>
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(5 S) '(K D))
<'K
((5 S) (K D)): K
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(K H) '(K S))
<#<void>
((K H) (K S)): #<void>
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(J C) '(4 H))
<'J
((J C) (4 H)): J
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(7 D) '(5 D))
<7
((7 D) (5 D)): 7 High Flush
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(6 C) '(3 S))
<6
((6 C) (3 S)): 6
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(6 H) '(3 S))
```

```
<6
((6 H) (3 S)): 6
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(X D) '(K H))
<'K
((X D) (K H)): K
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(Q H) '(6 C))
<'Q
((Q H) (6 C)): Q
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(2 C) '(7 S))
<7
((2 C) (7 S)): 7
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(4 D) '(6 C))
<6
((4 D) (6 C)): 6
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(J H) '(K H))
<'K
((J H) (K H)): K High Flush
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(7 D) '(7 S))
<#<void>
((7 D) (7 S)): #<void>
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(5 S) '(J D))
<'J
((5 S) (J D)): J
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(A D) '(5 S))
<'A
((A D) (5 S)): A
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(3 H) '(8 D))
<8
((3 H) (8 D)): 8
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(J H) '(4 H))
<'J
((J H) (4 H)): J High Flush
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(2 D) '(A C))
<'A
((2 D) (A C)): A
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(6 H) '(4 S))
<6
((6 H) (4 S)): 6
> (classify-two-cards-ur (pick-two-cards))
>(higher-rank '(2 H) '(9 C))
<9
((2 H) (9 C)): 9
>
```

Task 4c - Two Card Poker Classifier

```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (classify-two-cards (pick-two-cards))
>(higher-rank '(3 D) '(9 C))
<9
((3 D) (9 C)): Nine
> (classify-two-cards (pick-two-cards))
>(higher-rank '(9 S) '(2 D))
<9
((9 S) (2 D)): Nine
> (classify-two-cards (pick-two-cards))
>(higher-rank '(4 S) '(4 H))
<#<void>
((4 S) (4 H)): #<void>
> (classify-two-cards (pick-two-cards))
>(higher-rank '(A D) '(A C))
<#<void>
((A D) (A C)): #<void>
> (classify-two-cards (pick-two-cards))
>(higher-rank '(K H) '(2 S))
<'K
((K H) (2 S)): King
> (classify-two-cards (pick-two-cards))
>(higher-rank '(4 S) '(6 C))
<6
((4 S) (6 C)): Six
> (classify-two-cards (pick-two-cards))
>(higher-rank '(5 H) '(6 D))
<6
((5 H) (6 D)): #t
> (classify-two-cards (pick-two-cards))
>(higher-rank '(8 C) '(K S))
<'K
((8 C) (K S)): King
> (classify-two-cards (pick-two-cards))
>(higher-rank '(K D) '(X H))
<'K
((K D) (X H)): King
> (classify-two-cards (pick-two-cards))
>(higher-rank '(9 D) '(2 C))
<9
((9 D) (2 C)): Nine
> (classify-two-cards (pick-two-cards))
>(higher-rank '(X S) '(Q C))
<'Q
((X S) (Q C)): Queen
> (classify-two-cards (pick-two-cards))
>(higher-rank '(7 H) '(9 H))
<9
((7 H) (9 H)): Nine High Flush
> (classify-two-cards (pick-two-cards))
>(higher-rank '(8 D) '(K H))
<'K
((8 D) (K H)): King
> (classify-two-cards (pick-two-cards))
>(higher-rank '(6 D) '(3 C))
<6
((6 D) (3 C)): Six
```

```
> (classify-two-cards (pick-two-cards))
>(higher-rank '(8 H) '(Q D))
<'Q
((8 H) (Q D)): Queen
> (classify-two-cards (pick-two-cards))
>(higher-rank '(9 D) '(X H))
<'X
((9 D) (X H)): #t
> (classify-two-cards (pick-two-cards))
>(higher-rank '(J H) '(2 C))
<'J
((J H) (2 C)): Jack
> (classify-two-cards (pick-two-cards))
>(higher-rank '(9 C) '(Q C))
<'Q
((9 C) (Q C)): Queen High Flush
> (classify-two-cards (pick-two-cards))
>(higher-rank '(X S) '(J H))
<'J
((X S) (J H)): #t
> (classify-two-cards (pick-two-cards))
>(higher-rank '(Q C) '(A H))
<'A
((Q C) (A H)): Ace
>
```

```
#lang racket

(require racket/trace)

;---------
; Programming Assignment 3: Two Card Poker


;; Task 4a
( 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

(define (pick-two-cards)
  (define card1 (pick-a-card))
  (define card2 (pick-a-card))
  (cond
    [(equal? card1 card2)
     (pick-two-cards)]
    (else
     (cons (pick-a-card) (cons (pick-a-card) '()))
     ))
  )


(define (index-rank card)
  (define card-rank (rank card))
  (cond
    [(number? card-rank) card-rank]
    [else (face-ranker card-rank)])
  )

(define (face-ranker card-rank)
  (cond
    [(eq? 'X card-rank) 10]
    [(eq? 'J card-rank) 11]
    [(eq? 'Q card-rank) 12]
    [(eq? 'K card-rank) 13]
    [(eq? 'A card-rank) 14]
    [else 0]))

(define (card-name card)
  (cond
    [(equal? 1 card) "One"]
    [(equal? 2 card) "Two"]
    [(equal? 3 card) "Three"]
    [(equal? 4 card) "Four"]
    [(equal? 5 card) "Five"]
    [(equal? 6 card) "Six"]
    [(equal? 7 card) "Seven"]
    [(equal? 8 card) "Eight"]
    [(equal? 9 card) "Nine"]
    [(equal? 'X card) "Ten"]
    [(equal? 'J card) "Jack"]
    [(equal? 'Q card) "Queen"]
    [(equal? 'K card) "King"]
    [(equal? 'A card) "Ace"]
    )
  )

;; Display the higher rank card
(define (higher-rank card1 card2)
```

```
  (define card1-rank (index-rank card1))
  (define card2-rank (index-rank card2))

  (cond
    [(< card1-rank card2-rank) display (rank card2)]
    [(> card1-rank card2-rank) display (rank card1)]
    )
  )

(trace higher-rank)


;; Classify-two-cards-ur
( define ( classify-two-cards-ur card-pair )

   ( define card1 ( car card-pair ) )
   ( define card2 ( cadr card-pair ) )
   ( define card1-rank (index-rank card1))
   ( define card2-rank (index-rank card2))
   ( define card1-suit (suit card1))
   ( define card2-suit (suit card2))
   ( define high ( higher-rank card1 card2 ) )
   ( display card-pair )
   ( display ": " )
   ( cond
      ( ( equal? card1-suit card2-suit )
        ( cond
           ( ( or
               ( = 1 ( - card1-rank card2-rank))
               ( = 1 ( - card2-rank card1-rank)))
             ( display high ( display " High Straight Flush " ) ) )
           ( else
             ( display high ) ( display " High Flush " ) ) ) )
      ( else
        ( cond
           ( ( or
               ( = 1 ( - card1-rank card2-rank))
               ( = 1 ( - card2-rank card1-rank))
               ( display high ) ( display " High Straight " ) ) )
           ( else
             ( cond
                ( ( equal? ( car card1 ) ( car card2 ) )
                  ( display " Pair of " ) ( display ( car card1 ) ) ( display "'s" ) )
                ( else
                  ( display high ) ( display " High " ) ) ) ) ) ) ) )

 )



(trace higher-rank)


;; Classify-two-cards
(define (classify-two-cards card-pair)
   ( define card1 ( car card-pair ) )
   ( define card2 ( cadr card-pair ) )
   ( define card1-rank (index-rank card1))
```

```
   ( define card2-rank (index-rank card2))
   ( define card1-suit (suit card1))
   ( define card2-suit (suit card2))
   ( define higher ( higher-rank card1 card2 ) )
   ( define higher-name (card-name higher))
   ( display card-pair )( display ": " )
   ( cond
      ( ( equal? card1-suit card2-suit )
        ( cond
           ( ( or
               ( = 1 ( - card1-rank card2-rank))
               ( = 1 ( - card2-rank card1-rank)))
             (classify-two-cards (pick-two-cards))( display higher-name ( display "
High Straight Flush " ) ) )
           ( else
             ( display higher-name ) ( display " High Flush " ) ) ) )
      ( else
        ( cond
           ( ( or
               ( = 1 ( - card1-rank card2-rank))
               ( = 1 ( - card2-rank card1-rank))
               ( display higher-name ) ( display " High Straight " ) ) )
           ( else
             ( cond
                ( ( equal? ( car card1 ) ( car card2 ) )
                  ( display " Pair of " ) ( display higher-name ) ( display "'s" ) )
                ( else
                  ( display higher-name ) ( display " High " ) ) ) ) ) ) ) )

)
```