Racket Programming Assignment #3: Lambda and Basic Lisp

Learning Abstract

This programming assignment, as the name suggests, Lambda functions and Basic Lisp programming. The first two sections show the creation of a few basic lambda functions and lisp functions such as car and cdr. And the second two sections put those skills to action creating a color processor and poker representation.

Task 1: Lambda

Task 1a - Three ascending integers

Three ascending integers demo

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( ( lambda ( x ) ( cons x ( cons ( + x 1 ) ( cons ( + x 2 ) '( ) )))) 5 )
'(5 6 7)
> ( ( lambda ( x ) ( cons x ( cons ( + x 1 ) ( cons ( + x 2 ) '( ) )))) 0 )
'(0 1 2)
> ( ( lambda ( x ) ( cons x ( cons ( + x 1 ) ( cons ( + x 2 ) '( ) )))) 108 )
'(108 109 110)
>
```

Task 1b - Make list in reverse order

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( ( lambda ( x )
        ( cons ( car ( cdr ( cdr x )))
               ( cons ( car ( cdr x ))
                      ( cons (car x) '() ))))
    '( red yellow blue ) )
'(blue yellow red)
> ( ( lambda ( x )
        ( cons ( car ( cdr ( cdr x )))
               ( cons ( car ( cdr x ))
                      ( cons (car x) '() ))))
    '( 10 20 30 ) )
'(30 20 10)
> ( ( lambda ( x )
        ( cons ( car ( cdr ( cdr x )))
               ( cons ( car ( cdr x ))
                      ( cons (car x) '() ))))
    '( "Professor Plum" "Colonel Mustard" "Miss Scarlet" ) )
'("Miss Scarlet" "Colonel Mustard" "Professor Plum")
>
```

Task 1c - Random number generator

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
4
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
3
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
3
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
5
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
4
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
5
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
3
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
4
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
3
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    3 5 )
3
>
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
12
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
17
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
16
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
16
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
11
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
17
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
17
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
11
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
12
> ( ( lambda ( x y )
       ( + x ( random  ( + ( - y x ) 1 ) ) ) )
    11 17 )
15
>
```

Task 2: List Processing References and Constructors

Redacted Racket Session Featuring Referencers and Constructors Demo:

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

This task extends the sampler demo from Lesson #6 "Basic Lisp Programming." The simple sampler programming asks for a list and returns a random element. The extension of this program takes two parameters: a command and a list.

Sampler Demo:

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( sampler )
(?): ( red orange yellow green blue indigo violet )
red
(?): ( red orange yellow green blue indigo violet )
indigo
(?): ( red orange yellow green blue indigo violet )
yellow
(?): ( red orange yellow green blue indigo violet )
green
(?): ( red orange yellow green blue indigo violet )
yellow
(?): ( red orange yellow green blue indigo violet )
blue
(?): ( aet ate eat ate tae tea )
aet
(?): ( aet ate eat ate tae tea )
tae
(?): ( aet ate eat ate tae tea )
aet
(?): ( aet ate eat ate tae tea )
( aet ate eat ate tae tea )
ate
(?): ( aet ate eat ate tae tea )
eat
(?): tae
```

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

Sampler code:

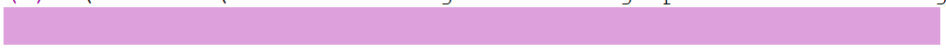
```
#lang racket

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

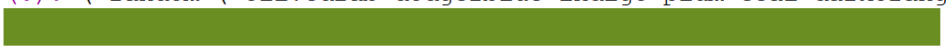
color-thing Demo 1:

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

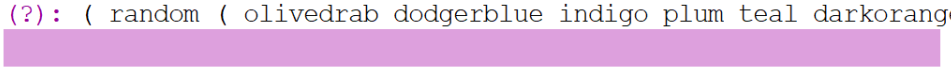
```
> ( color-thing )
(?): ( random ( olivedrab dodgerblue indigo plum teal darkorange ) )
```



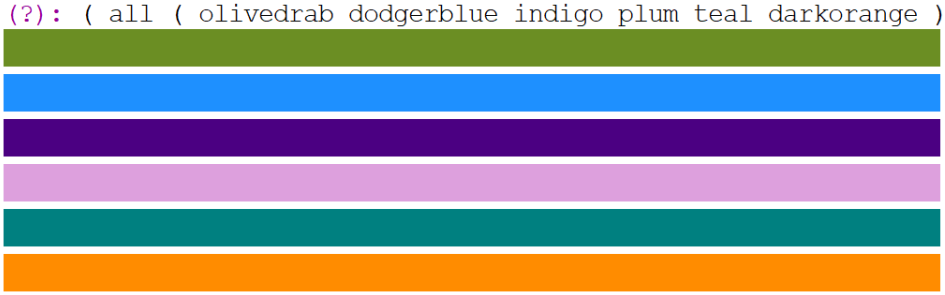
```
(?): ( random ( olivedrab dodgerblue indigo plum teal darkorange ) )
```



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



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



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

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

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

(?): eof

color-thing Demo 2:

color-thing Code:

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

;----------
;Test phrase
;( olivedrab dodgerblue indigo plum teal darkorange )

;----------
; get-random displays a random color from the list
( define ( get-random list )
   ( define the-color ( list-ref list ( random ( length list ))))
   (display ( rectangle 500 20 "solid" the-color ))( display "\n")
   )

;-----------
; all prints all colors in the list in order recursively
```

```
( define ( all list )
      ( cond (( not (null?  ( cdr list )) )
           ( define the-color ( car list ) )
           ( display ( rectangle 500 20 "solid" the-color ))( display "\n")
           ( all ( cdr list ) )
           )
          (( null?  ( cdr list ) )
           ( display ( rectangle 500 20 "solid" ( car list ) ))( display "\n")
           )
      )
   )

;----------
; get-number displays the nth color of the list
( define ( get-number index list )
   ( display "entered number\n" )
   ( define the-color ( list-ref list ( - index 1 ) ) )
   ( display ( rectangle 500 20 "solid" the-color ))( display "\n")
   )

;----------
; sampler code edited to fit the color-thing name )
( define ( color-thing )
   ( display "(?): " )
   ( define input ( read ) )
   ( define command  ( car input ))
   ( define the-list ( append ( car (cdr input )) ( cdr (cdr input ))))

   ( cond (( equal? command 'random )
           ( get-random the-list ))

          (( equal? command 'all )
           ( all the-list ))

          (( integer? command )
           ( get-number command the-list ))
          )
   ( color-thing )
)
```

Task 4: Two Card Poker

Card Demo:

```
Welcome to DrRacket, version 8.3 [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
> ( rank c2 )
'Q
> ( suit c1 )
'C
> ( suit c2 )
'H
> ( red? c1 )
#f
> ( red? c2 )
#t
> ( black? c1 )
#t
> ( black? c2 )
```

```
> ( 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 )
'(8 D)
> ( pick-a-card )
'(A D)
> ( pick-a-card )
'(2 S)
> ( pick-a-card )
'(5 D)
> ( pick-a-card )
'(7 D)
> ( pick-a-card )
'(A H)
>
```

cards 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 )
     )
   )
```

pick-two-cards Demo:

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( pick-two-cards )
((J S) (9 S))
> ( pick-two-cards )
((3 S) (7 D))
> ( pick-two-cards )
((8 H) (8 D))
> ( pick-two-cards )
((6 C) (9 D))
> ( pick-two-cards )
((9 D) (4 S))
>
```

higher-rank Demo:

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(6 H) '(9 D))
<9
9
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(J H) '(3 D))
<'J
'J
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(4 H) '(J S))
<'J
'J
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(6 C) '(9 D))
<9
9
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(A S) '(6 S))
<'A
'A
>
```

classifier-ur Demo:

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

classifier-ur 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 )
     )
   )

;----------
;pick-two-cards accepts no parameters and
;returns two different cards in a list

( define ( pick-two-cards )
   ( define new-deck ( deck ) )
   ( define c1 ( list-ref new-deck ( random ( length new-deck ) ) ))
   ( define c2 ( list-ref new-deck ( random ( length new-deck ) ) ))

   ( cond
      (( equal? c1 c2 )
       ( pick-two-cards )
       )
      ( else
        ( define list-of-two ( append ( cons c1 '()) ( cons c2 '()) ))
        list-of-two
        )
      )
   )

;----------
;higher-rank accepts two parameters and
;returns one card

( define ( rank-val card )
   ( cond
      (( equal? ( rank card ) 'A )
       14
       )
      (( equal? ( rank card ) 'K )
       13
```

```
       )
      (( equal? ( rank card ) 'Q )
       12
       )
      (( equal? ( rank card ) 'J )
       11
       )
      (( equal? ( rank card ) 'X )
       10
       )
      ( else
        ( rank card )
       )
      )
   )

( define ( higher-rank c1 c2 )
   ( cond
      (( > ( rank-val c1 ) ( rank-val c2 ) )
       ( rank c1 )
       )
      (( > ( rank-val c2 ) ( rank-val c1 ) )
       ( rank c2 )
       )
      (else
       ( rank c1 )
       )
      )
   )

;( trace higher-rank )

;----------;----------;----------;----------;----------
; straight? checks if two cards are next to one another ie. 2,3 J,Q
( define ( straight? c1 c2 )
   ( cond
      (( or ( = 1 ( - ( rank-val c1 ) ( rank-val c2 ) ) )
                ( = -1 ( - ( rank-val c1 ) ( rank-val c2 ) ) ) )
       #t
       )
      (( = -1 ( - ( rank-val c1 ) ( rank-val c2 ) ) )
       #t
       )
      ( else #f )
      )
   )

;-----
; flush? checks if two cards share the same suit
```

```
( define ( flush? c1 c2 )
   ( cond
      (( equal? ( suit c1 ) ( suit c2 ) )
       #t
       )
      ( else #f )
      )
   )

;------
; pair? checks if the rank of two cards are equal
( define ( pair? c1 c2 )
   ( cond
      (( = ( rank-val c1 ) ( rank-val c2 ) )
       #t
       )
      ( else #f )
      )
   )
;----------
;classify-two-cards-ur picks two cards and categorizes them as
; pair, high, straight, flush, or straight flush

( define ( classify-two-cards-ur card-list )
   ( define c1 ( car card-list ) )
   ( define c2 ( car ( cdr card-list ) ) )

   ( display card-list ) ( display ":  " )
   ( display ( higher-rank c1 c2 ) ) ( display " high " )

   ( cond
      (( straight? c1 c2 )
       ( display "straight " )
       )
      )
   ( cond
      (( flush? c1 c2 )
       ( display "flush " )
       )
      )

   ( cond
      (( pair? c1 c2 )
       ( display "pair " )
       )
      )
   )
```

classifier Demo:

```
> ( classify-two-cards ( pick-two-cards ) )
((A C) (5 D)):  ace high
> ( classify-two-cards ( pick-two-cards ) )
((7 D) (A S)):  ace high
> ( classify-two-cards ( pick-two-cards ) )
((8 C) (Q D)):  queen high
> ( classify-two-cards ( pick-two-cards ) )
((5 C) (K S)):  king high
> ( classify-two-cards ( pick-two-cards ) )
((K S) (2 H)):  king high
> ( classify-two-cards ( pick-two-cards ) )
((9 D) (Q D)):  queen high flush
> ( classify-two-cards ( pick-two-cards ) )
((2 D) (7 C)):  seven high
> ( classify-two-cards ( pick-two-cards ) )
((7 C) (5 H)):  seven high
> ( classify-two-cards ( pick-two-cards ) )
((4 C) (J S)):  jack high
> ( classify-two-cards ( pick-two-cards ) )
((9 C) (8 H)):  nine high straight
> ( classify-two-cards ( pick-two-cards ) )
((6 S) (2 D)):  six high
> ( classify-two-cards ( pick-two-cards ) )
((K S) (3 D)):  king high
> ( classify-two-cards ( pick-two-cards ) )
((A H) (8 C)):  ace high
> ( classify-two-cards ( pick-two-cards ) )
((J H) (9 D)):  jack high
> ( classify-two-cards ( pick-two-cards ) )
((X D) (2 H)):  ten high
> ( classify-two-cards ( pick-two-cards ) )
((J S) (Q D)):  queen high straight
> ( classify-two-cards ( pick-two-cards ) )
((8 H) (7 D)):  eight high straight
> ( classify-two-cards ( pick-two-cards ) )
((7 D) (3 C)):  seven high
> ( classify-two-cards ( pick-two-cards ) )
((Q H) (9 H)):  queen high flush
> ( classify-two-cards ( pick-two-cards ) )
((A C) (X C)):  ace high flush
>
```

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 )
     )
   )

;----------
;pick-two-cards accepts no parameters and
;returns two different cards in a list

( define ( pick-two-cards )
   ( define new-deck ( deck ) )
   ( define c1 ( list-ref new-deck ( random ( length new-deck ) ) ))
   ( define c2 ( list-ref new-deck ( random ( length new-deck ) ) ))

   ( cond
      (( equal? c1 c2 )
       ( pick-two-cards )
       )
      ( else
        ( define list-of-two ( append ( cons c1 '()) ( cons c2 '()) ))
        list-of-two
        )
      )
   )

;----------
;higher-rank accepts two parameters and
;returns one card

( define ( rank-val card )
   ( cond
      (( equal? ( rank card ) 'A )
       14
       )
      (( equal? ( rank card ) 'K )
       13
```

```
       )
      (( equal? ( rank card ) 'Q )
       12
       )
      (( equal? ( rank card ) 'J )
       11
       )
      (( equal? ( rank card ) 'X )
       10
       )
      ( else
        ( rank card )
       )
      )
   )

( define ( higher-rank c1 c2 )
   ( cond
      (( > ( rank-val c1 ) ( rank-val c2 ) )
       ( rank c1 )
       )
      (( > ( rank-val c2 ) ( rank-val c1 ) )
       ( rank c2 )
       )
      (else
       ( rank c1 )
       )
      )
   )

;( trace higher-rank )

;----------;----------;----------;----------;----------
; straight? checks if two cards are next to one another ie. 2,3 J,Q
( define ( straight? c1 c2 )
   ( cond
      (( or ( = 1 ( - ( rank-val c1 ) ( rank-val c2 ) ) )
                ( = -1 ( - ( rank-val c1 ) ( rank-val c2 ) ) ) )
       #t
       )
      (( = -1 ( - ( rank-val c1 ) ( rank-val c2 ) ) )
       #t
       )
      ( else #f )
      )
   )

;-----
; flush? checks if two cards share the same suit
```

```
( define ( flush? c1 c2 )
   ( cond
      (( equal? ( suit c1 ) ( suit c2 ) )
       #t
       )
      ( else #f )
      )
   )

;------
; pair? checks if the rank of two cards are equal
( define ( pair? c1 c2 )
   ( cond
      (( = ( rank-val c1 ) ( rank-val c2 ) )
       #t
       )
      ( else #f )
      )
   )

;-----
; word-rank takes one rank and makes it into an english
; representation ie. 1 => one
( define ( word-rank card )
   ( define rank-list '( one two three four five six seven eight nine ten jack
queen king ace ) )
   ( cond
      (( integer? card )
       ( list-ref rank-list ( - card 1 ) )
       )
      (( equal? 'X card )
       ( list-ref rank-list 9 )
       )
      (( equal? 'J card )
       ( list-ref rank-list 10 )
       )
      (( equal? 'Q card )
       ( list-ref rank-list 11 )
       )
      (( equal? 'K card )
       ( list-ref rank-list 12 )
       )
      (( equal? 'A card )
       ( list-ref rank-list 13 )
       )
      )
   )

;----------
```

```
;classify-two-cards-ur picks two cards and catagorizes them as
; pair, high, straight, flush, or straight flush

( define ( classify-two-cards-ur card-list )
   ( define c1 ( car card-list ) )
   ( define c2 ( car ( cdr card-list ) ) )

   ( display card-list ) ( display ":  " )
   ( display( word-rank ( higher-rank c1 c2 ) ) ) ( display " high " )

   ( cond
      (( straight? c1 c2 )
       ( display "straight " )
       )
      )
   ( cond
      (( flush? c1 c2 )
       ( display "flush " )
       )
      )

   ( cond
      (( pair? c1 c2 )
       ( display "pair " )
       )
      )
   )

( define ( classify-two-cards card-list )
   ( classify-two-cards-ur card-list )
   )
```