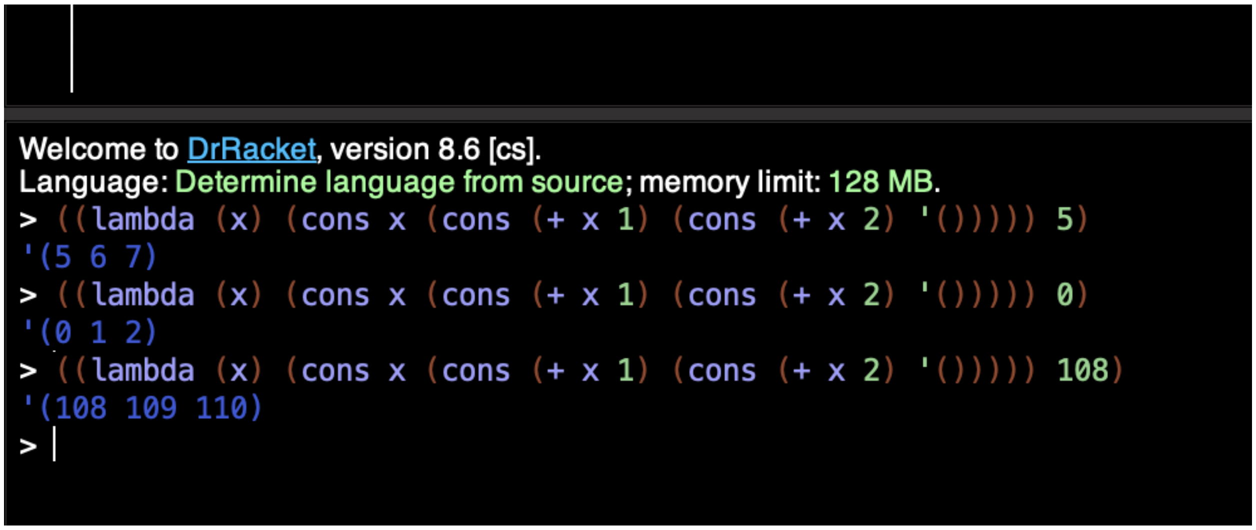
# Racket Programming Assignment #3: Lambda and Basic Lisp

## Abstract:

The first two tasks are highly constrained. One of these pertains to lambda functions, and the other to basic list processing operations in Lisp. The second two tasks, which are considerably more involved, ask you to extend pro- grams that are presented in Lesson 6 “Basic Lisp Programming”. These two tasks will afford you enough freedom to make some mistakes along the way, and to see opportunities to revise your code to make it better, even after you get your programs to do what they are supposed to do.

## Task 1 Lambda :

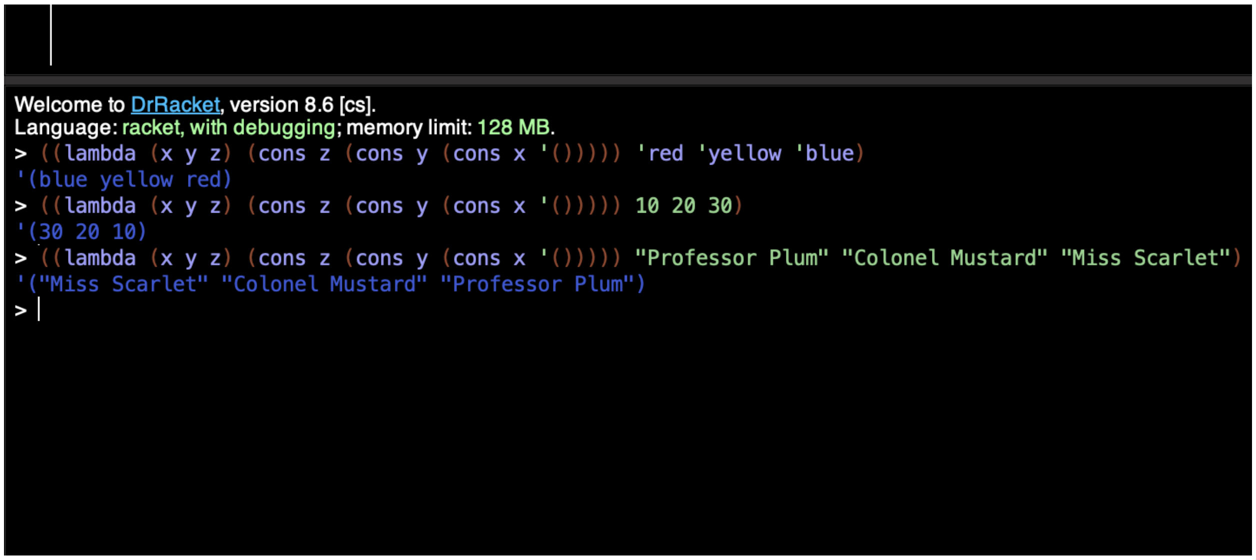
### Task 1a: Three Ascending Integers



```
Welcome to DrRacket, version 8.6 [cs].
Language: Determine language from source; 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 1 Lambda :

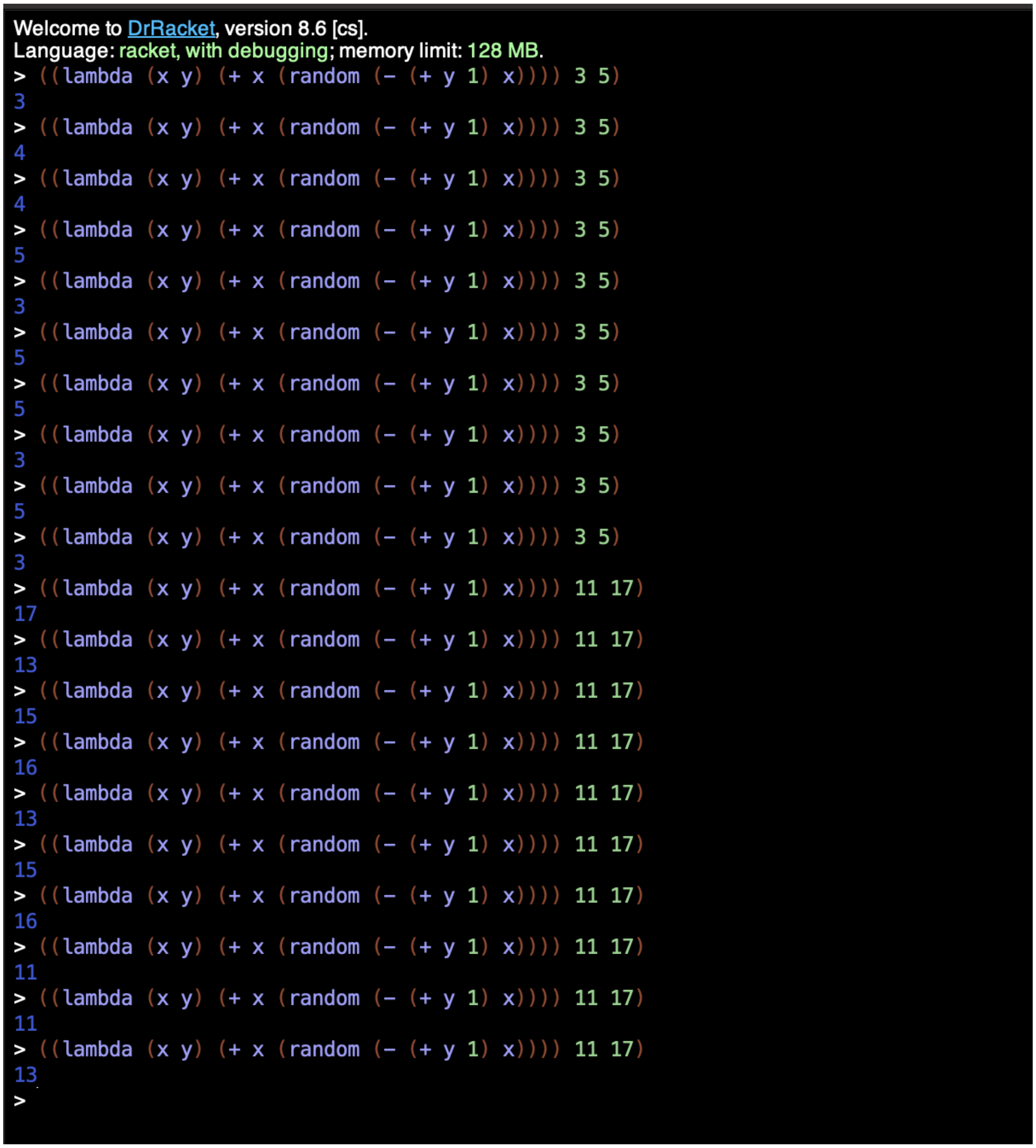
## Task 1b: Make List in Reverse Order



```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ((lambda (x y z) (cons z (cons y (cons x '())))) 'red 'yellow 'blue)
'(blue yellow red)
> ((lambda (x y z) (cons z (cons y (cons x '())))) 10 20 30)
'(30 20 10)
> ((lambda (x y z) (cons z (cons y (cons x '())))) "Professor Plum" "Colonel Mustard" "Miss Scarlet")
'("Miss Scarlet" "Colonel Mustard" "Professor Plum")
> 
```

## Task 1 Lambda :

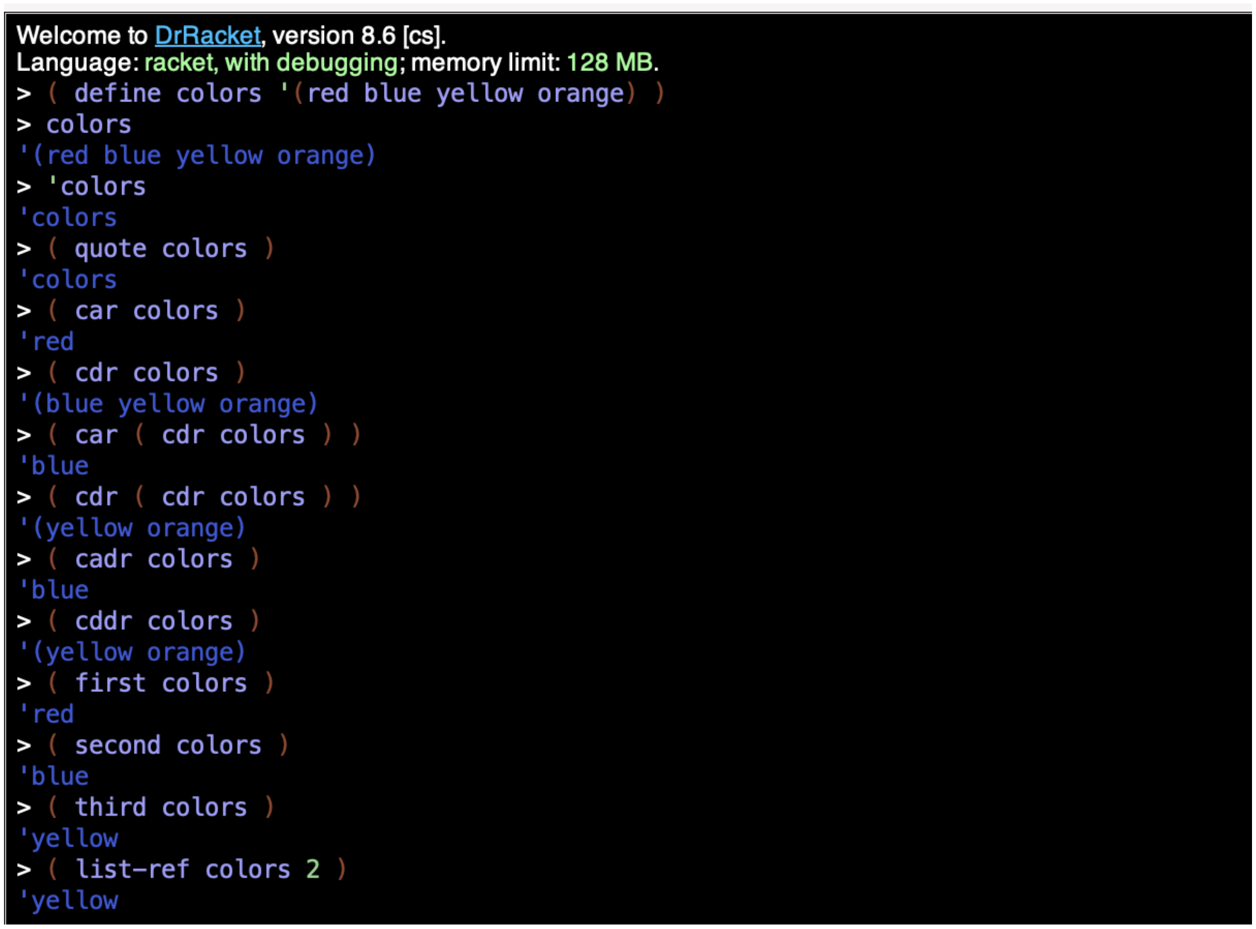
## Task 1c: Random number generator



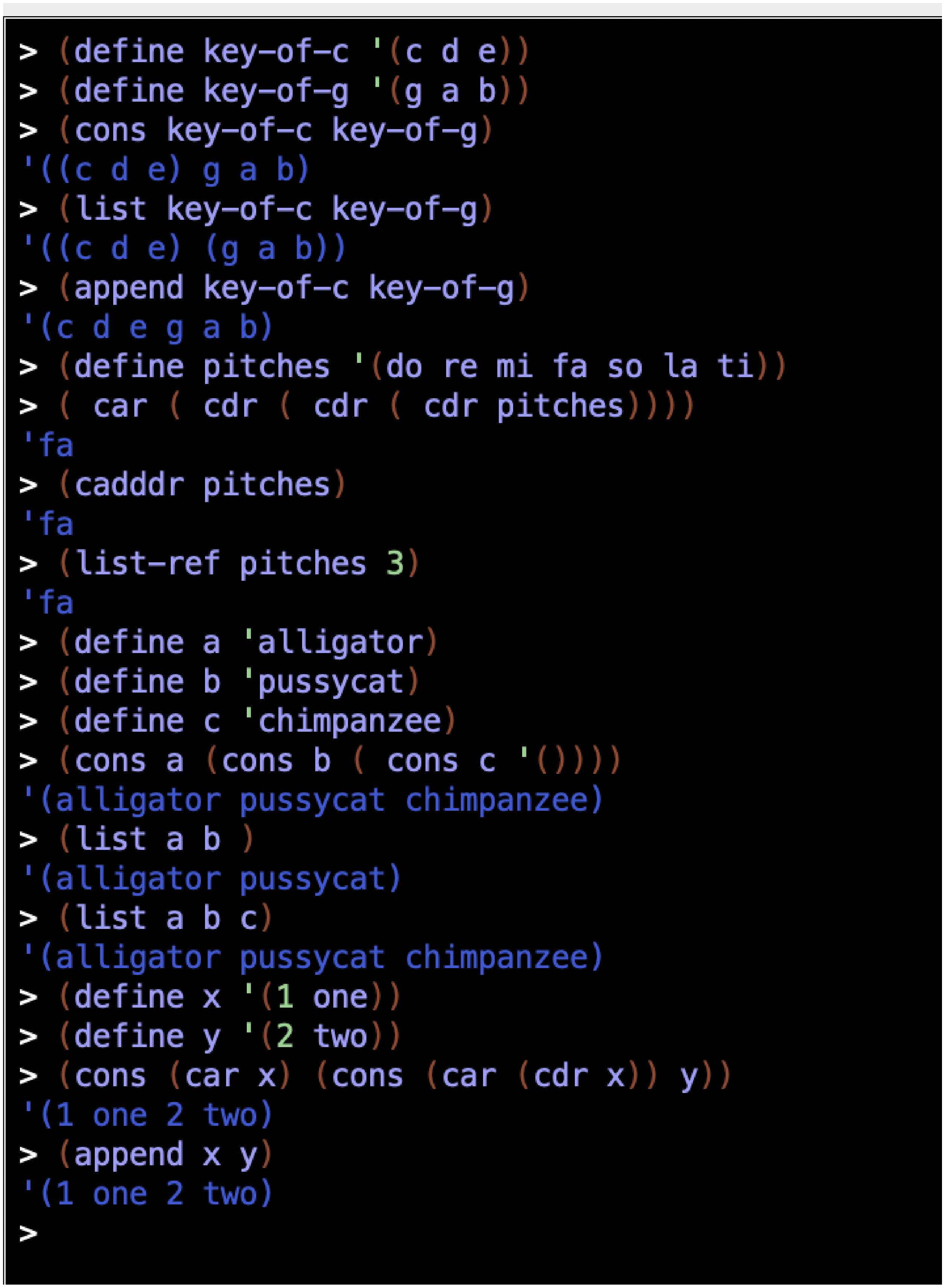
```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
3
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
4
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
4
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
5
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
3
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
5
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
5
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
3
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
5
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 3 5)
3
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
17
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
13
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
15
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
16
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
13
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
15
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
16
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
11
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
11
> ((lambda (x y) (+ x (random (- (+ y 1) x)))) 11 17)
13
>
```

## Task 2 : List Processing Referencers and Constructors

Demo



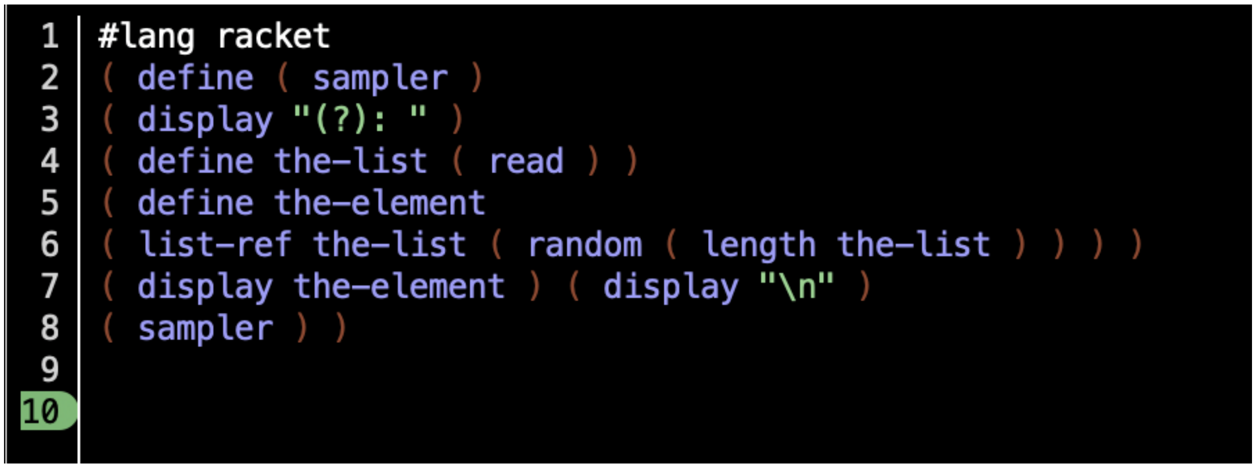
```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define colors '(red blue yellow orange) )
> colors
'(red blue yellow orange)
> 'colors
'colors
> ( quote colors )
'colors
> ( car colors )
'red
> ( cdr colors )
'(blue yellow orange)
> ( car ( cdr colors ) )
'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
```



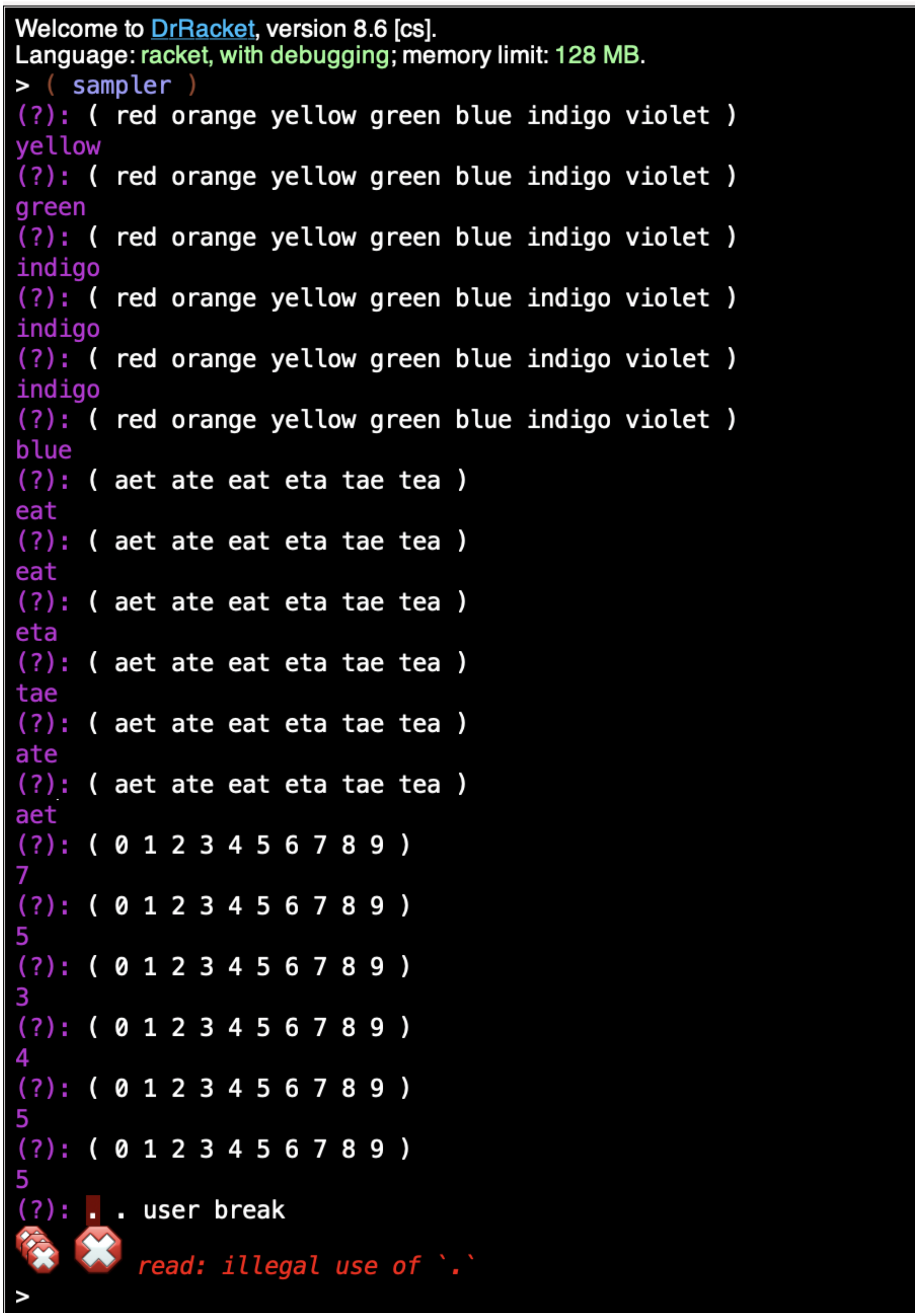
```
> (define key-of-c '(c d e))
> (define key-of-g '(g a b))
> (cons key-of-c key-of-g)
'((c d e) g a b)
> (list key-of-c key-of-g)
'((c d e) (g a b))
> (append key-of-c key-of-g)
'(c d e g a b)
> (define pitches '(do re mi fa so la ti))
> ( car ( cdr ( cdr ( cdr pitches))))
'fa
> (cadddr pitches)
'fa
> (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 )
'(alligator pussycat)
> (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



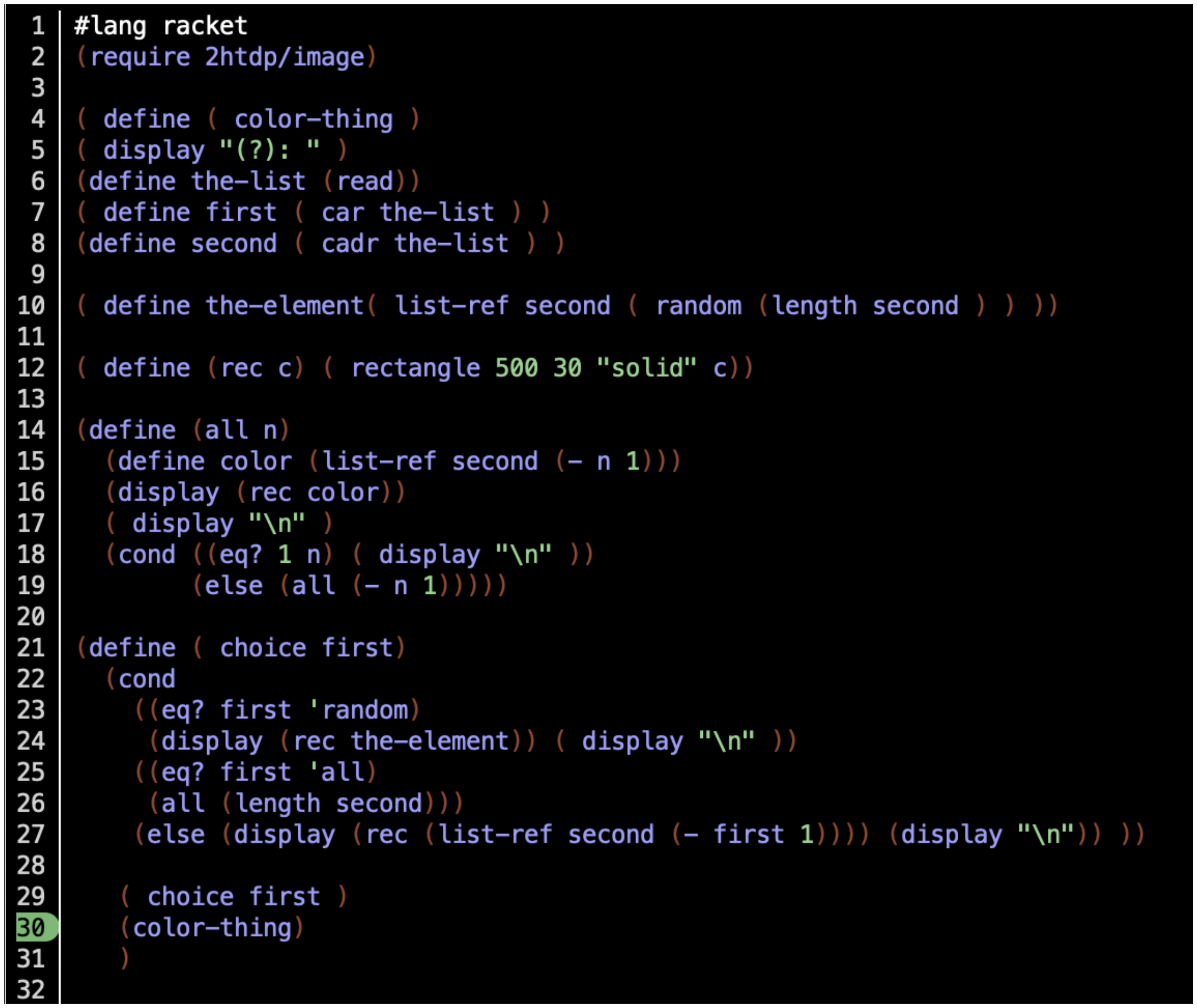
```
#lang racket
( define ( sampler )
( display "(?): " )
( define the-list ( read ) )
( define the-element
( list-ref the-list ( random ( length the-list ) ) ) )
( display the-element ) ( display "\n" )
( sampler ) )
```



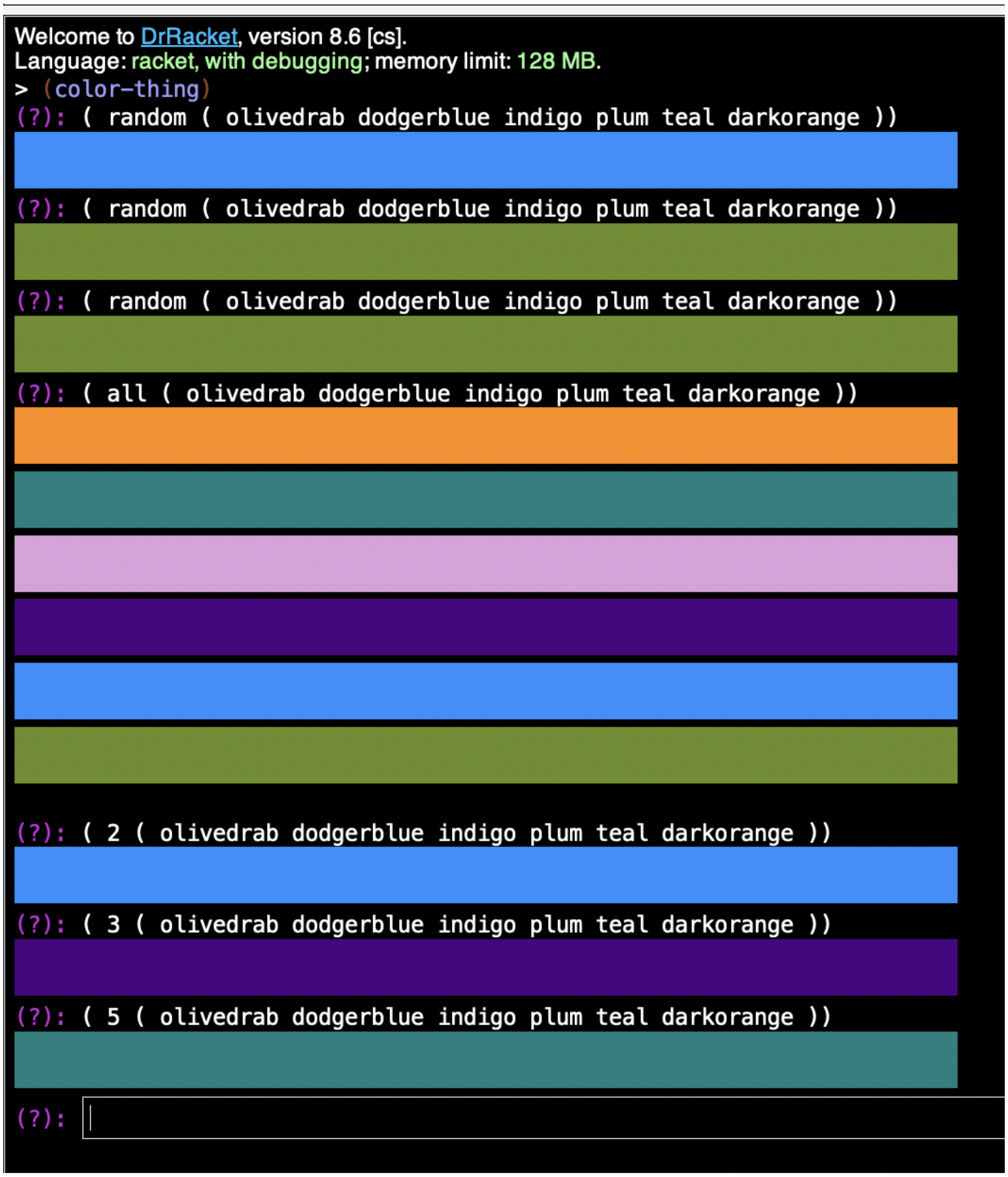
```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( sampler )
(?): ( red orange yellow green blue indigo violet )
yellow
(?): ( red orange yellow green blue indigo violet )
green
(?): ( red orange yellow green blue indigo violet )
indigo
(?): ( red orange yellow green blue indigo violet )
indigo
(?): ( red orange yellow green blue indigo violet )
indigo
(?): ( red orange yellow green blue indigo violet )
blue
(?): ( aet ate eat eta tae tea )
eat
(?): ( aet ate eat eta tae tea )
eat
(?): ( aet ate eat eta tae tea )
eta
(?): ( aet ate eat eta tae tea )
tae
(?): ( aet ate eat eta tae tea )
ate
(?): ( aet ate eat eta tae tea )
aet
(?): ( 0 1 2 3 4 5 6 7 8 9 )
7
(?): ( 0 1 2 3 4 5 6 7 8 9 )
5
(?): ( 0 1 2 3 4 5 6 7 8 9 )
3
(?): ( 0 1 2 3 4 5 6 7 8 9 )
4
(?): ( 0 1 2 3 4 5 6 7 8 9 )
5
(?): ( 0 1 2 3 4 5 6 7 8 9 )
5
(?): . . user break
read: illegal use of `.`
>
```

## Task 3 : Little color interpreter

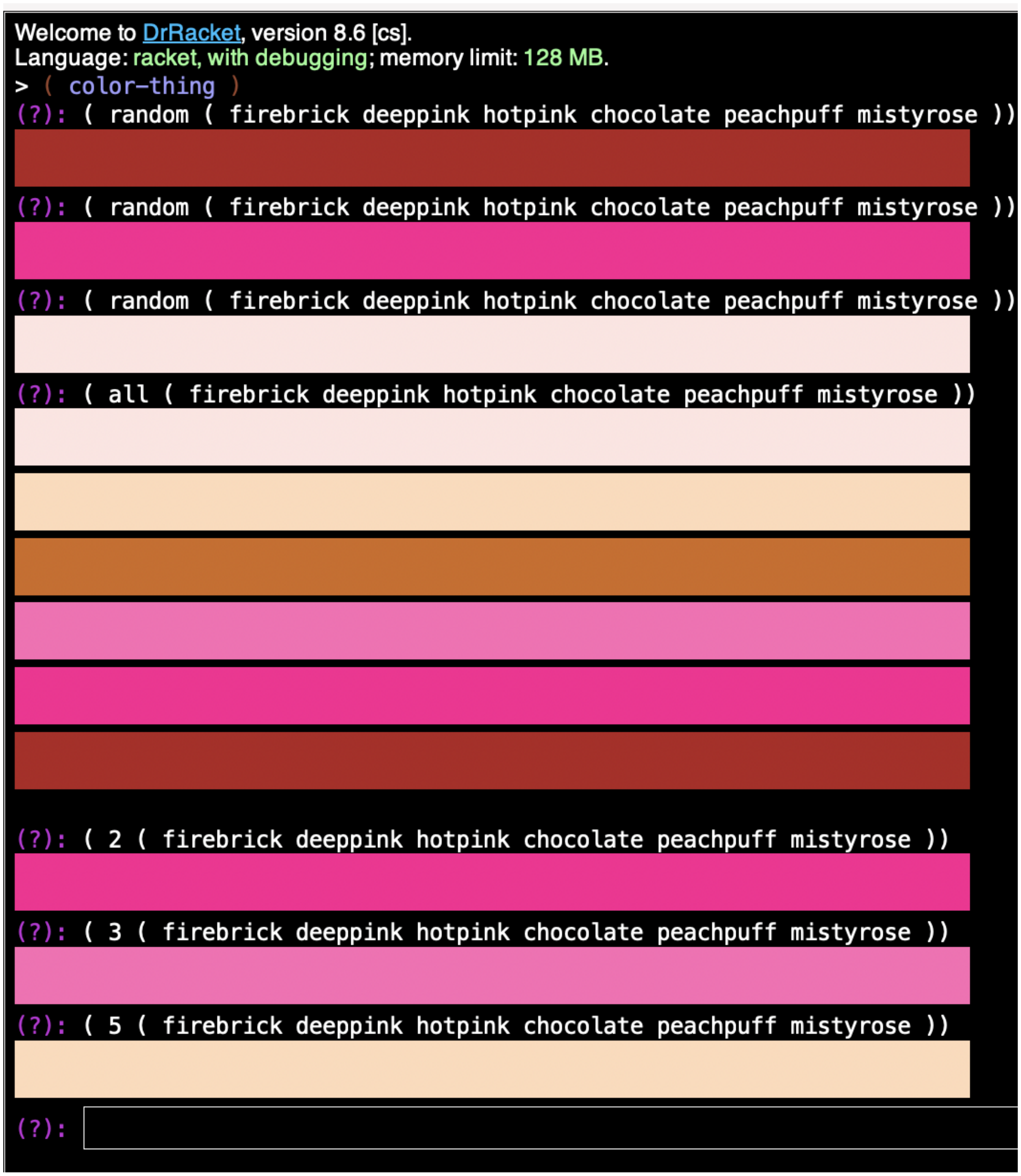
## Task 3b : Color thing interpreter



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

( define ( color-thing )
( display "(?): " )
(define the-list (read))
( define first ( car the-list ) )
(define second ( cadr the-list ) )

( define the-element( list-ref second ( random (length second ) ) ))

( define (rec c) ( rectangle 500 30 "solid" c))

(define (all n)
  (define color (list-ref second (- n 1)))
  (display (rec color))
  ( display "\n" )
  (cond ((eq? 1 n) ( display "\n" ))
        (else (all (- n 1)))))

(define ( choice first)
  (cond
    ((eq? first 'random)
     (display (rec the-element)) ( display "\n" ))
    ((eq? first 'all)
     (all (length second)))
    (else (display (rec (list-ref second (- first 1)))) (display "\n")) ))

   ( choice first )
   (color-thing)
   )

```



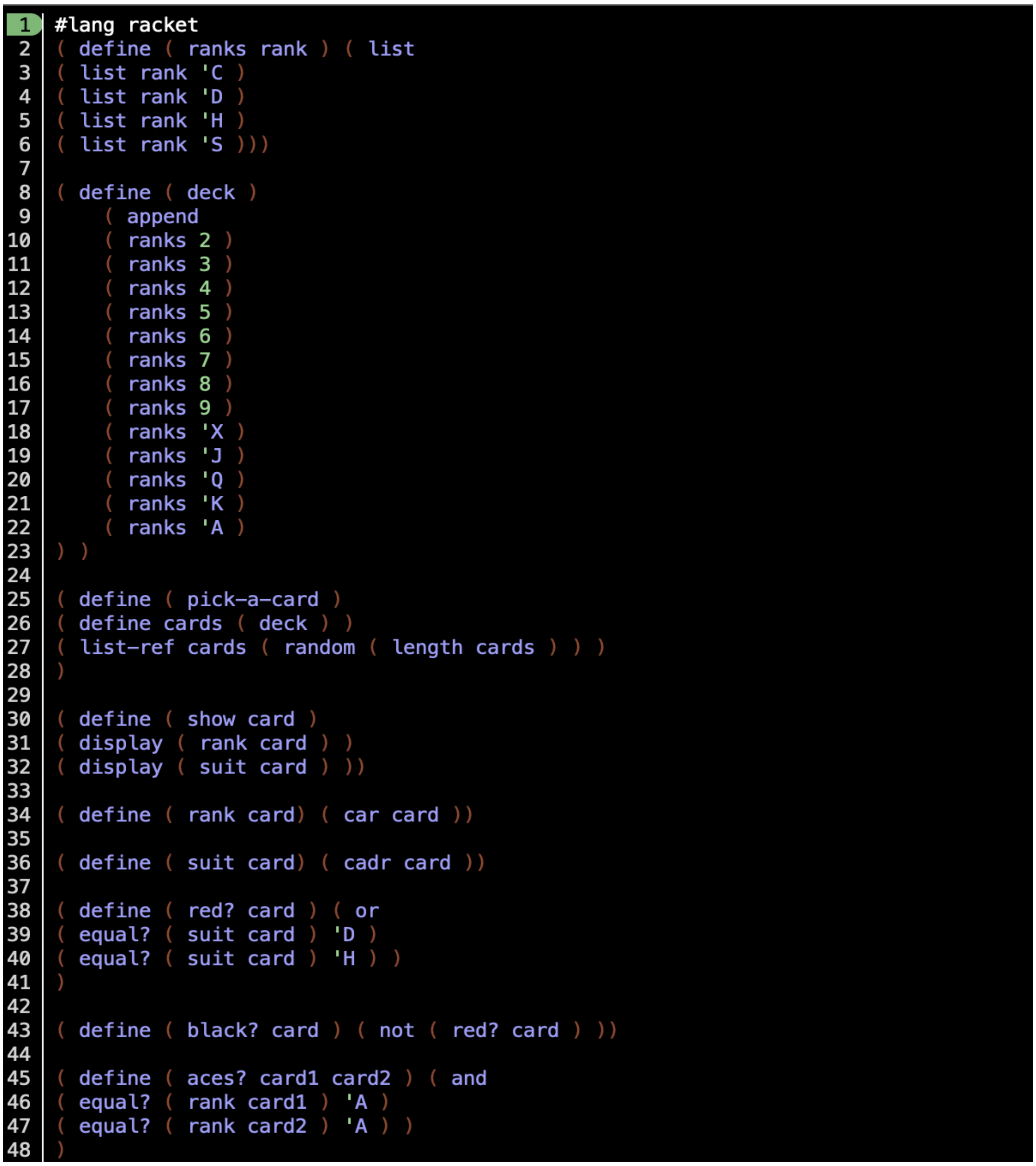
```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (color-thing)
(?): ( random ( olivedrab dodgerblue indigo plum teal darkorange ))
(?): ( random ( olivedrab dodgerblue indigo plum teal darkorange ))
(?): ( random ( olivedrab dodgerblue indigo plum teal 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 ))
(?): 
```



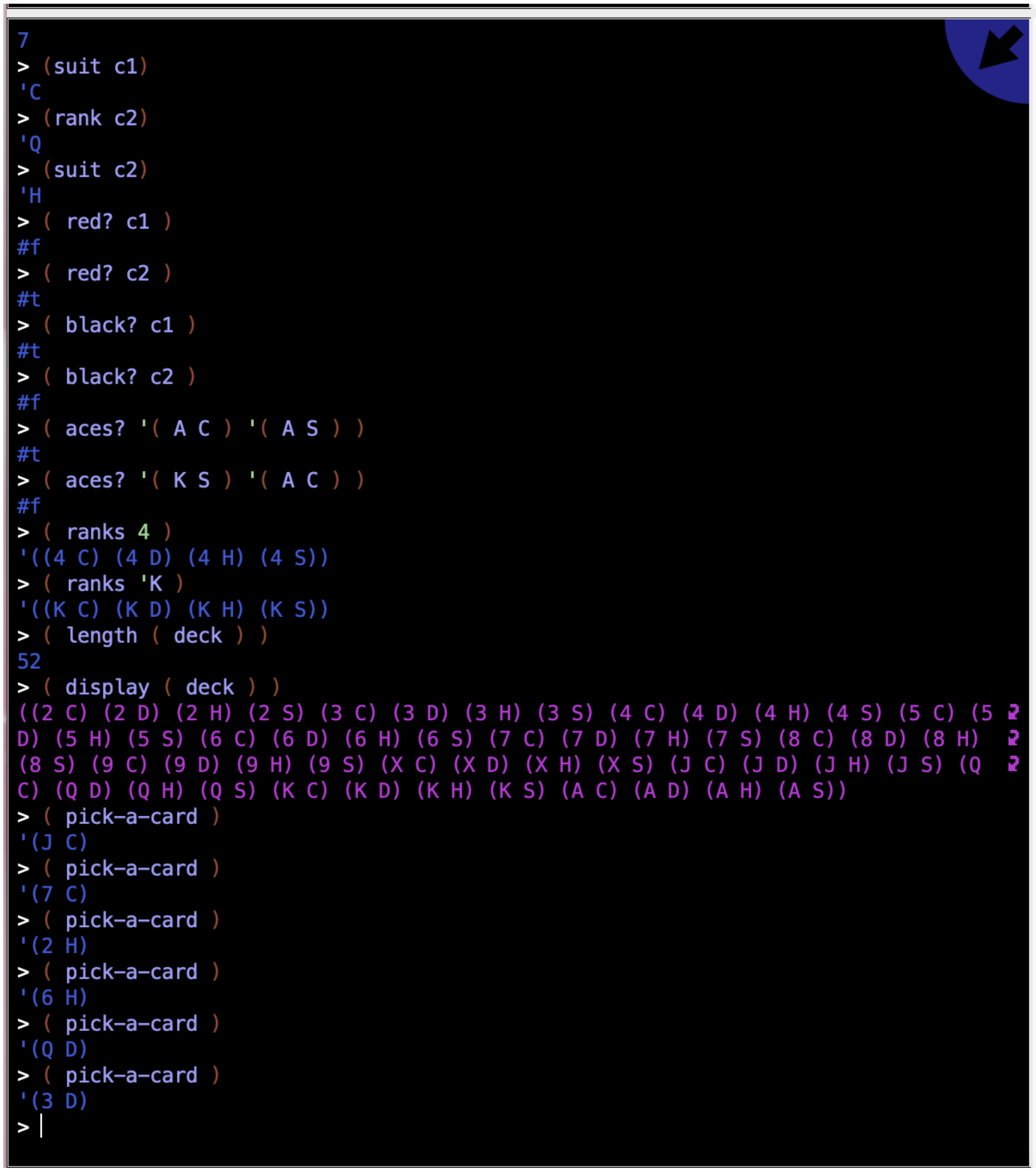
```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( color-thing )
(?): ( random ( firebrick deeppink hotpink chocolate peachpuff mistyrose ))
(?): ( random ( firebrick deeppink hotpink chocolate peachpuff mistyrose ))
(?): ( random ( firebrick deeppink hotpink chocolate peachpuff mistyrose ))
(?): ( all ( firebrick deeppink hotpink chocolate peachpuff mistyrose ))
(?): ( 2 ( firebrick deeppink hotpink chocolate peachpuff mistyrose ))
(?): ( 3 ( firebrick deeppink hotpink chocolate peachpuff mistyrose ))
(?): ( 5 ( firebrick deeppink hotpink chocolate peachpuff mistyrose ))
(?):
```

Task 4 : Two card poker

Task 4a : Establishing the card code from Lesson 6



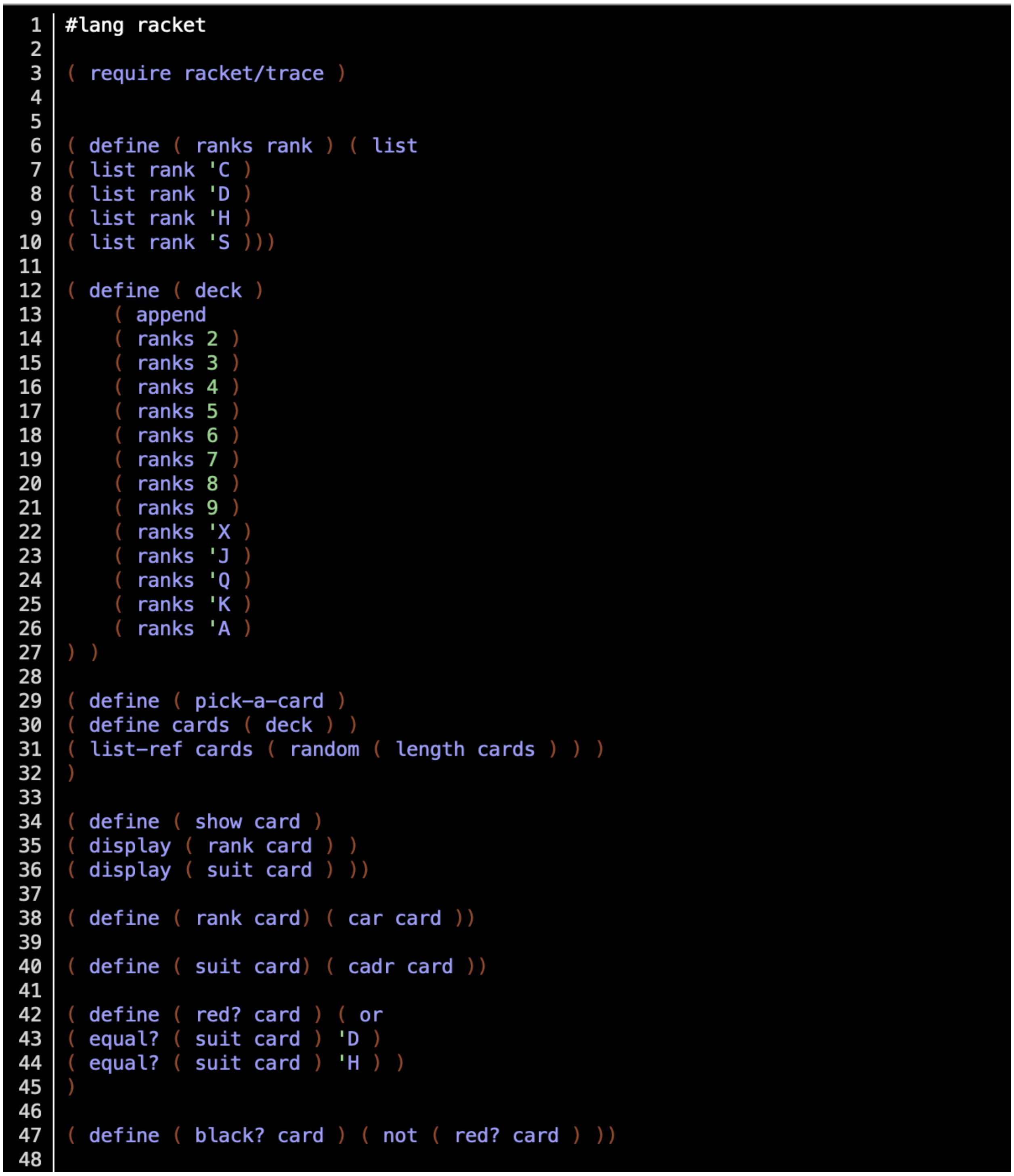
```
#lang racket
( define ( ranks rank ) ( list
( list rank 'C )
( list rank 'D )
( list rank 'H )
( list rank 'S )))

( 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 ) )
)
```



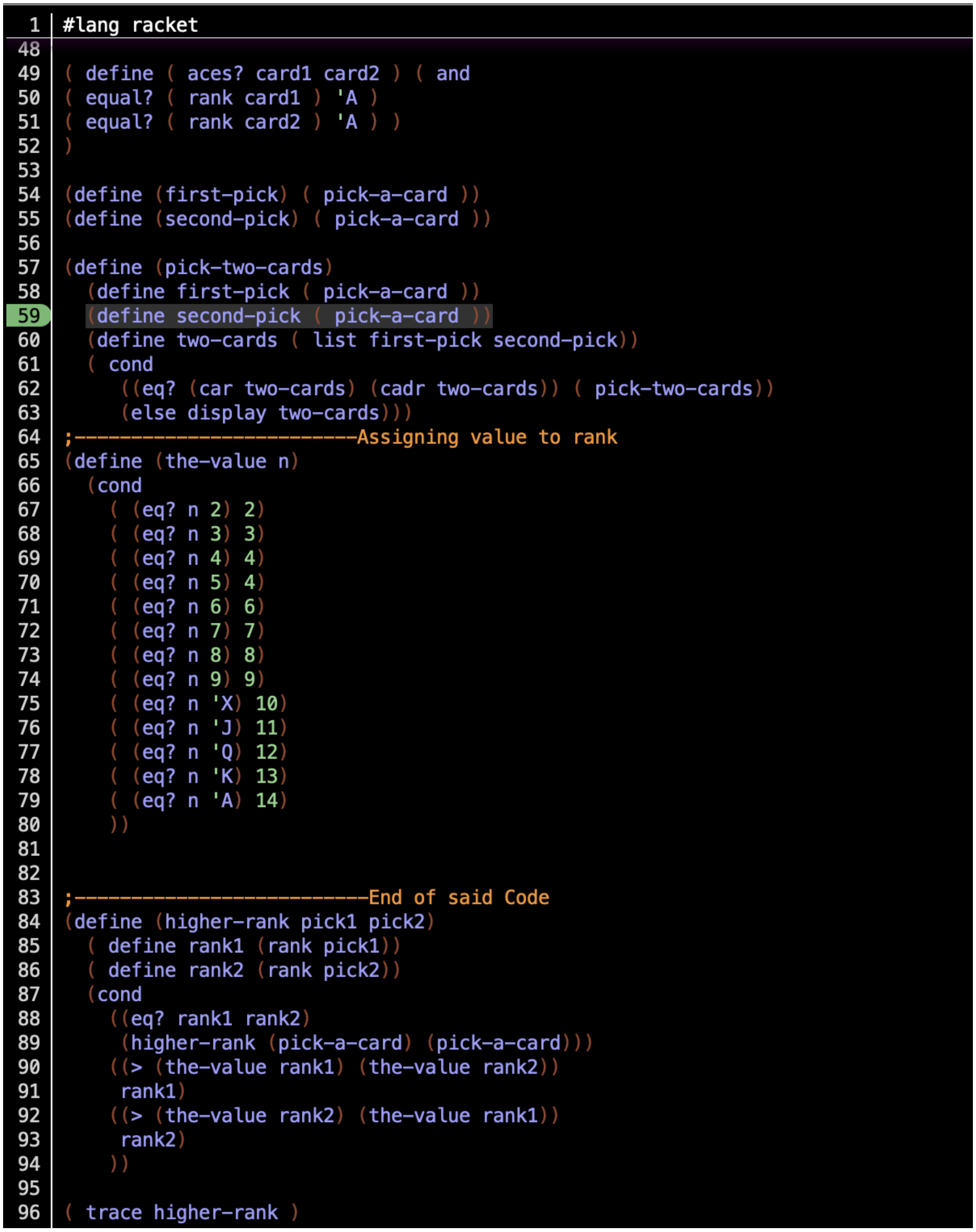
```
7
> (suit c1)
'C
> (rank c2)
'Q
> (suit c2)
'H
> ( red? c1 )
#f
> ( red? c2 )
#t
> ( black? c1 )
#t
> ( black? c2 )
#f
> ( aces? '( A C ) '( A S ) )
#t
> ( aces? '( K S ) '( A C ) )
#f
> ( ranks 4 )
'((4 C) (4 D) (4 H) (4 S))
> ( ranks 'K )
'((K C) (K D) (K H) (K S))
> ( length ( deck ) )
52
> (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

Task 4 : Two card poker

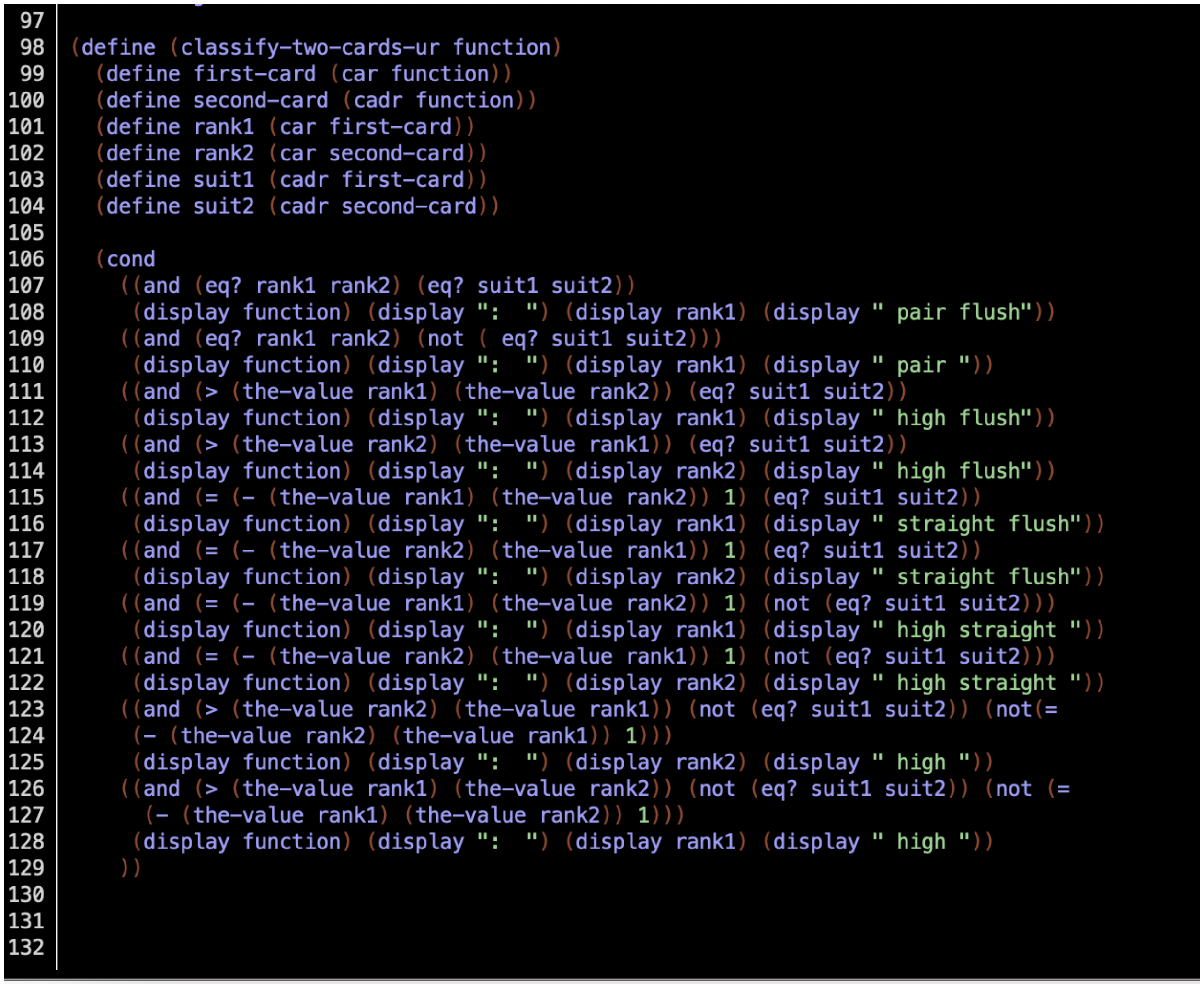
Task 4b : Two Card Poker Classifier, IR Version



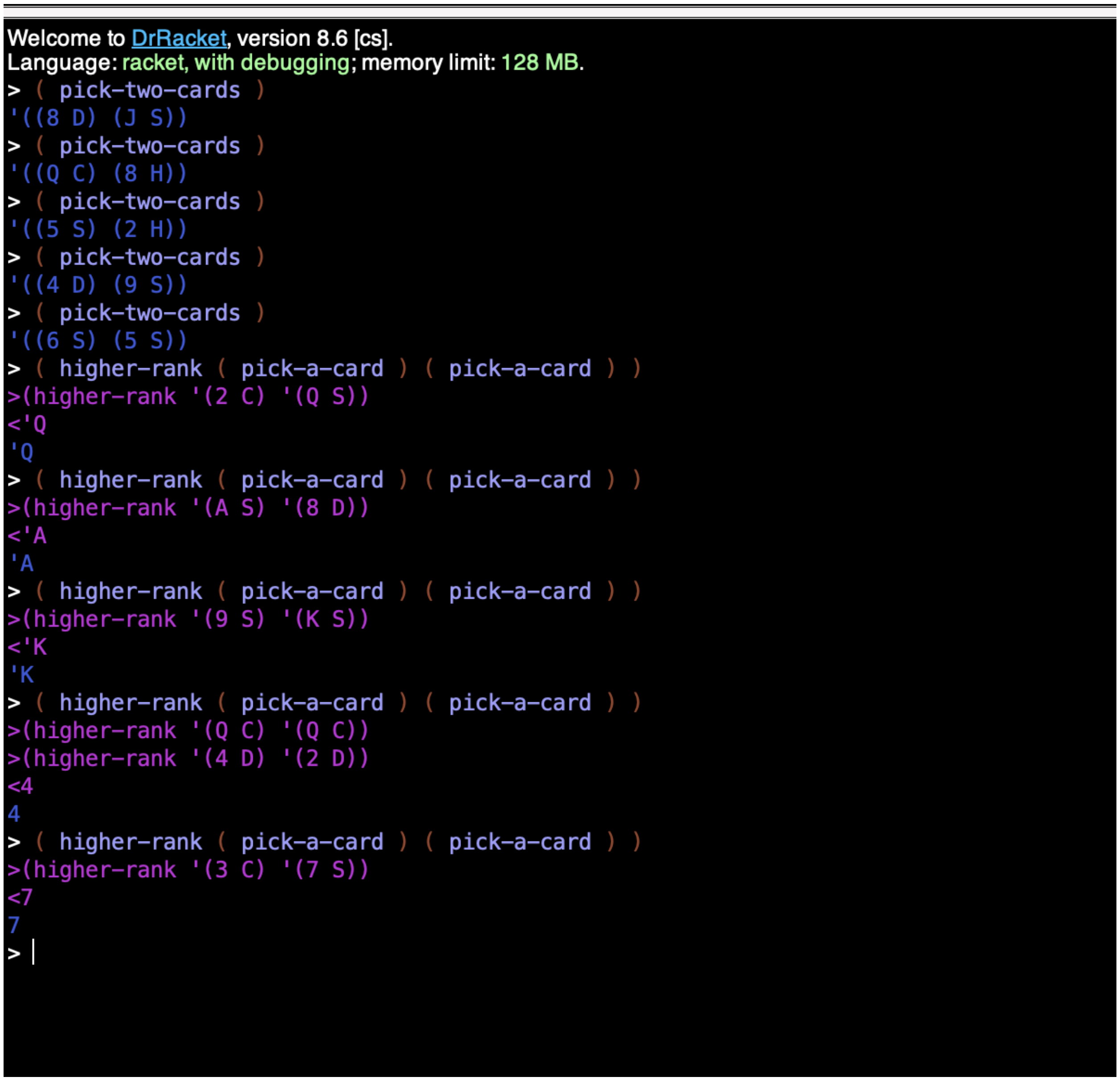
```racket
#lang racket

( 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 ) ))

```



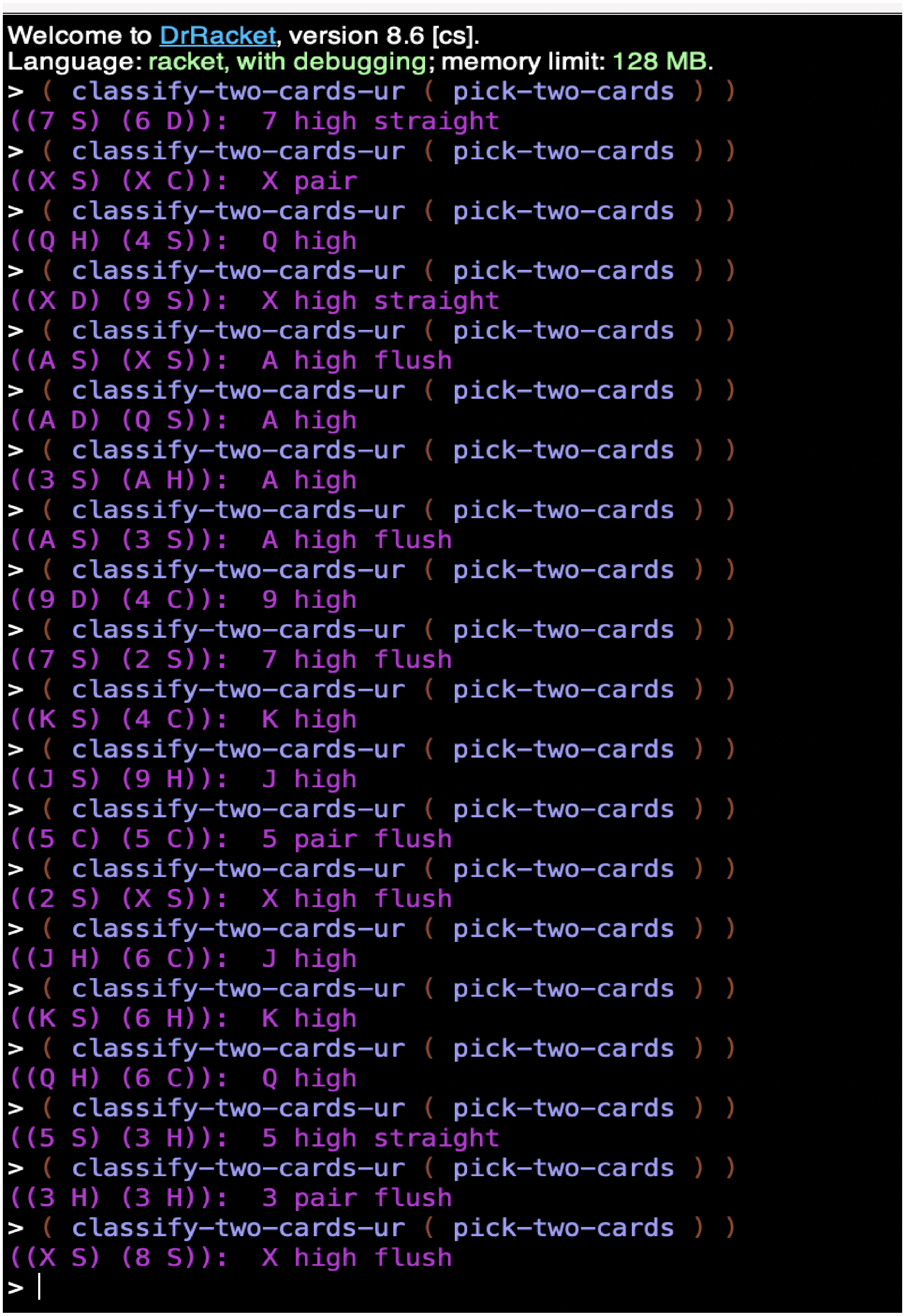
```racket
#lang racket

( define ( aces? card1 card2 ) ( and
( equal? ( rank card1 ) 'A )
( equal? ( rank card2 ) 'A ) )
)

(define (first-pick) ( pick-a-card ))
(define (second-pick) ( pick-a-card ))

(define (pick-two-cards)
  (define first-pick ( pick-a-card ))
  (define second-pick ( pick-a-card ))
  (define two-cards ( list first-pick second-pick))
  ( cond
     ((eq? (car two-cards) (cadr two-cards)) ( pick-two-cards))
     (else display two-cards)))
;-------------------------Assigning value to rank
(define (the-value n)
  (cond
    ( (eq? n 2) 2)
    ( (eq? n 3) 3)
    ( (eq? n 4) 4)
    ( (eq? n 5) 4)
    ( (eq? n 6) 6)
    ( (eq? n 7) 7)
    ( (eq? n 8) 8)
    ( (eq? n 9) 9)
    ( (eq? n 'X) 10)
    ( (eq? n 'J) 11)
    ( (eq? n 'Q) 12)
    ( (eq? n 'K) 13)
    ( (eq? n 'A) 14)
    ))


;--------------------------End of said Code
(define (higher-rank pick1 pick2)
  ( define rank1 (rank pick1))
  ( define rank2 (rank pick2))
  (cond
    ((eq? rank1 rank2)
     (higher-rank (pick-a-card) (pick-a-card)))
    ((> (the-value rank1) (the-value rank2))
     rank1)
    ((> (the-value rank2) (the-value rank1))
     rank2)
    ))

( trace higher-rank )
```



```scheme

(define (classify-two-cards-ur function)
  (define first-card (car function))
  (define second-card (cadr function))
  (define rank1 (car first-card))
  (define rank2 (car second-card))
  (define suit1 (cadr first-card))
  (define suit2 (cadr second-card))

  (cond
    ((and (eq? rank1 rank2) (eq? suit1 suit2))
     (display function) (display ":  ") (display rank1) (display " pair flush"))
    ((and (eq? rank1 rank2) (not ( eq? suit1 suit2)))
     (display function) (display ":  ") (display rank1) (display " pair "))
    ((and (> (the-value rank1) (the-value rank2)) (eq? suit1 suit2))
     (display function) (display ":  ") (display rank1) (display " high flush"))
    ((and (> (the-value rank2) (the-value rank1)) (eq? suit1 suit2))
     (display function) (display ":  ") (display rank2) (display " high flush"))
    ((and (= (- (the-value rank1) (the-value rank2)) 1) (eq? suit1 suit2))
     (display function) (display ":  ") (display rank1) (display " straight flush"))
    ((and (= (- (the-value rank2) (the-value rank1)) 1) (eq? suit1 suit2))
     (display function) (display ":  ") (display rank2) (display " straight flush"))
    ((and (= (- (the-value rank1) (the-value rank2)) 1) (not (eq? suit1 suit2)))
     (display function) (display ":  ") (display rank1) (display " high straight "))
    ((and (= (- (the-value rank2) (the-value rank1)) 1) (not (eq? suit1 suit2)))
     (display function) (display ":  ") (display rank2) (display " high straight "))
    ((and (> (the-value rank2) (the-value rank1)) (not (eq? suit1 suit2)) (not(=
     (- (the-value rank2) (the-value rank1)) 1)))
     (display function) (display ":  ") (display rank2) (display " high "))
    ((and (> (the-value rank1) (the-value rank2)) (not (eq? suit1 suit2)) (not (=
      (- (the-value rank1) (the-value rank2)) 1)))
     (display function) (display ":  ") (display rank1) (display " high "))
    ))



```



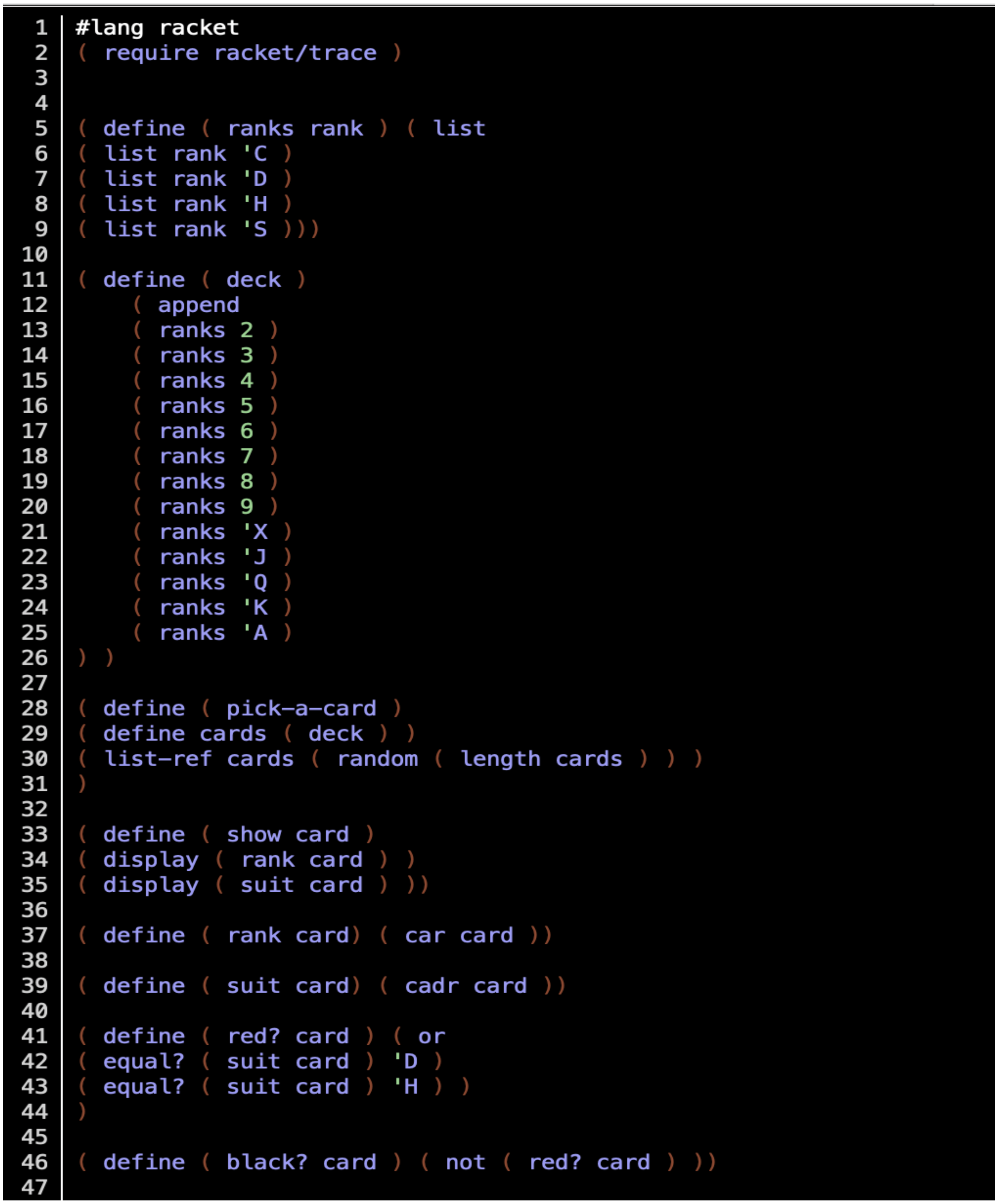
```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( pick-two-cards )
'((8 D) (J S))
> ( pick-two-cards )
'((Q C) (8 H))
> ( pick-two-cards )
'((5 S) (2 H))
> ( pick-two-cards )
'((4 D) (9 S))
> ( pick-two-cards )
'((6 S) (5 S))
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(2 C) '(Q S))
<'Q
'Q
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(A S) '(8 D))
<'A
'A
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(9 S) '(K S))
<'K
'K
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(Q C) '(Q C))
>(higher-rank '(4 D) '(2 D))
<4
4
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(3 C) '(7 S))
<7
7
> 
```



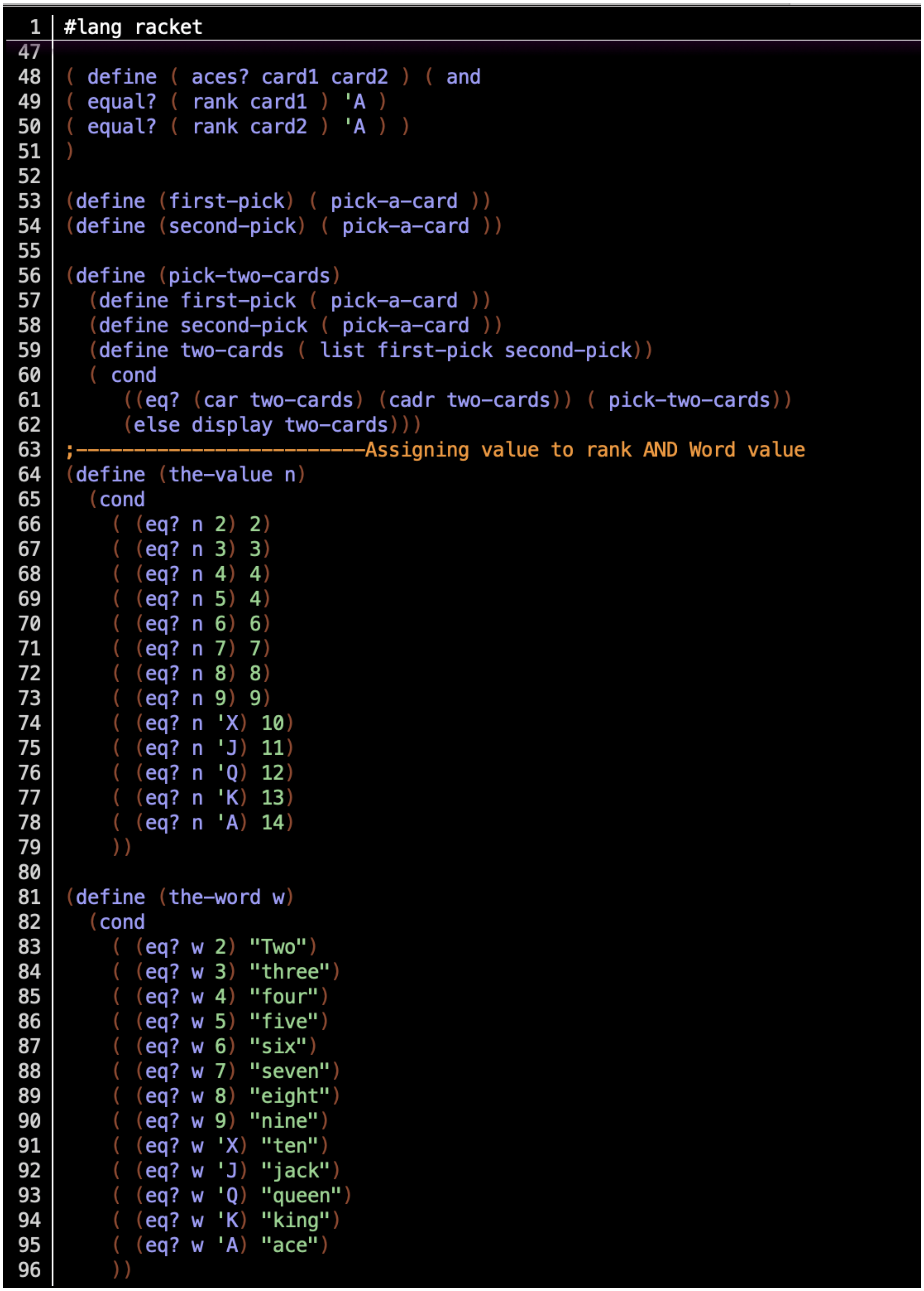
```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( classify-two-cards-ur ( pick-two-cards ) )
((7 S) (6 D)):  7 high straight
> ( classify-two-cards-ur ( pick-two-cards ) )
((X S) (X C)):  X pair
> ( classify-two-cards-ur ( pick-two-cards ) )
((Q H) (4 S)):  Q high
> ( classify-two-cards-ur ( pick-two-cards ) )
((X D) (9 S)):  X high straight
> ( classify-two-cards-ur ( pick-two-cards ) )
((A S) (X S)):  A high flush
> ( classify-two-cards-ur ( pick-two-cards ) )
((A D) (Q S)):  A high
> ( classify-two-cards-ur ( pick-two-cards ) )
((3 S) (A H)):  A high
> ( classify-two-cards-ur ( pick-two-cards ) )
((A S) (3 S)):  A high flush
> ( classify-two-cards-ur ( pick-two-cards ) )
((9 D) (4 C)):  9 high
> ( classify-two-cards-ur ( pick-two-cards ) )
((7 S) (2 S)):  7 high flush
> ( classify-two-cards-ur ( pick-two-cards ) )
((K S) (4 C)):  K high
> ( classify-two-cards-ur ( pick-two-cards ) )
((J S) (9 H)):  J high
> ( classify-two-cards-ur ( pick-two-cards ) )
((5 C) (5 C)):  5 pair flush
> ( classify-two-cards-ur ( pick-two-cards ) )
((2 S) (X S)):  X high flush
> ( classify-two-cards-ur ( pick-two-cards ) )
((J H) (6 C)):  J high
> ( classify-two-cards-ur ( pick-two-cards ) )
((K S) (6 H)):  K high
> ( classify-two-cards-ur ( pick-two-cards ) )
((Q H) (6 C)):  Q high
> ( classify-two-cards-ur ( pick-two-cards ) )
((5 S) (3 H)):  5 high straight
> ( classify-two-cards-ur ( pick-two-cards ) )
((3 H) (3 H)):  3 pair flush
> ( classify-two-cards-ur ( pick-two-cards ) )
((X S) (8 S)):  X high flush
> |
```

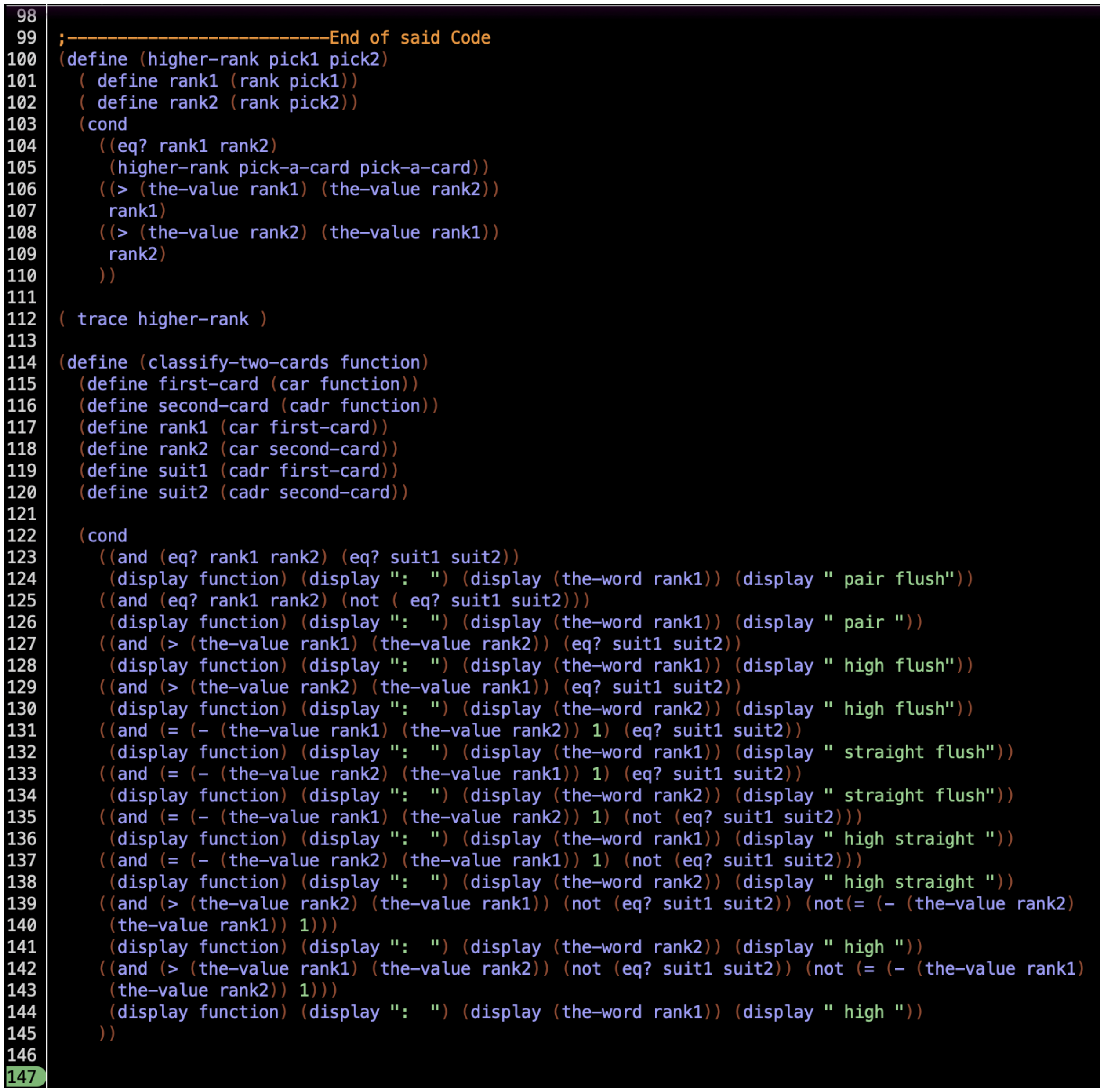
## Task 4 : Two card poker

## Task 4c : Two Card Poker Classifier

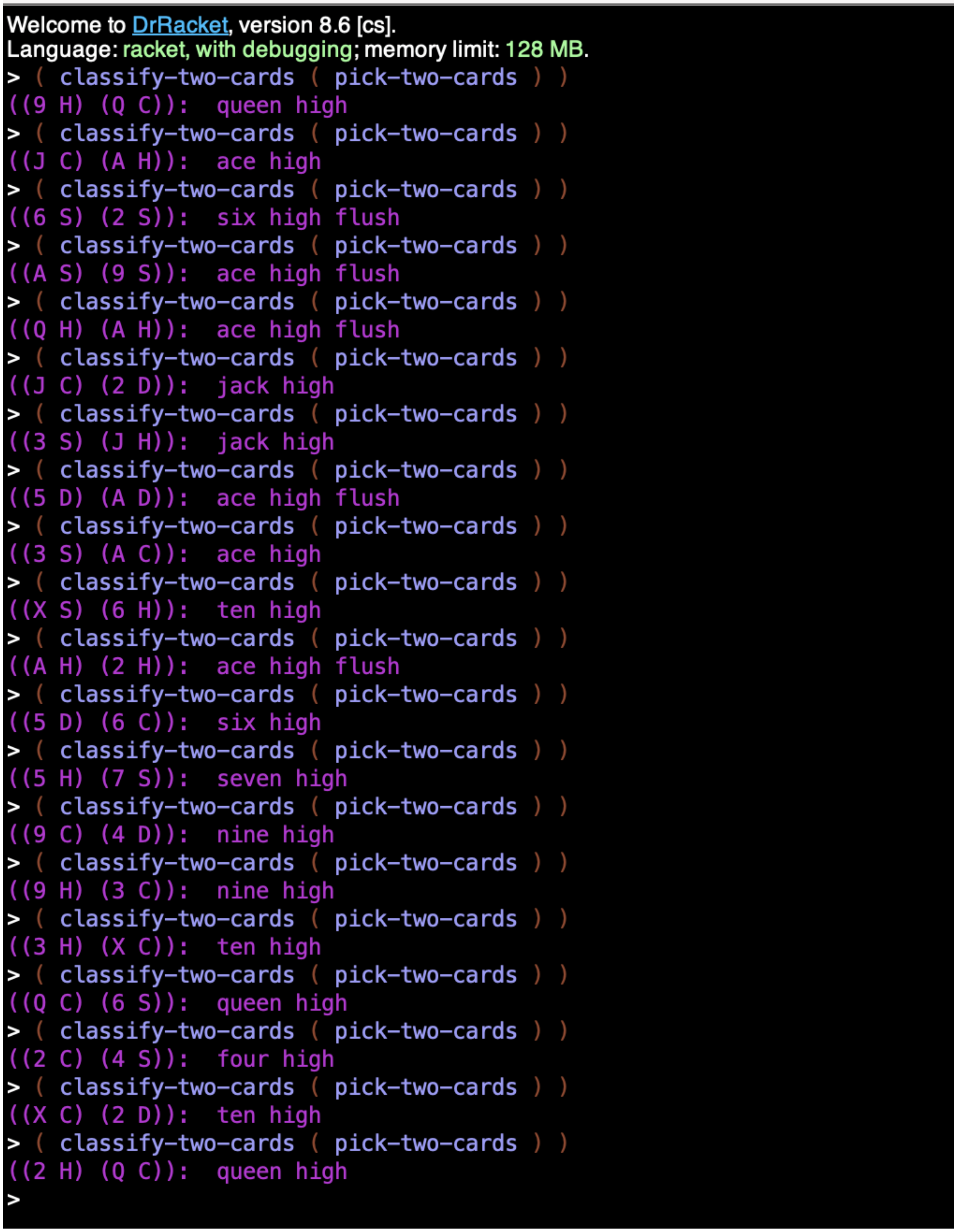


```racket
#lang racket
( require racket/trace )


( 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 ) ))

```


```
#lang racket

( define ( aces? card1 card2 ) ( and
( equal? ( rank card1 ) 'A )
( equal? ( rank card2 ) 'A ) )
)

(define (first-pick) ( pick-a-card ))
(define (second-pick) ( pick-a-card ))

(define (pick-two-cards)
  (define first-pick ( pick-a-card ))
  (define second-pick ( pick-a-card ))
  (define two-cards ( list first-pick second-pick))
  ( cond
     ((eq? (car two-cards) (cadr two-cards)) ( pick-two-cards))
     (else display two-cards)))
;-------------------------Assigning value to rank AND Word value
(define (the-value n)
  (cond
    ( (eq? n 2) 2)
    ( (eq? n 3) 3)
    ( (eq? n 4) 4)
    ( (eq? n 5) 4)
    ( (eq? n 6) 6)
    ( (eq? n 7) 7)
    ( (eq? n 8) 8)
    ( (eq? n 9) 9)
    ( (eq? n 'X) 10)
    ( (eq? n 'J) 11)
    ( (eq? n 'Q) 12)
    ( (eq? n 'K) 13)
    ( (eq? n 'A) 14)
    ))

(define (the-word w)
  (cond
    ( (eq? w 2) "Two")
    ( (eq? w 3) "three")
    ( (eq? w 4) "four")
    ( (eq? w 5) "five")
    ( (eq? w 6) "six")
    ( (eq? w 7) "seven")
    ( (eq? w 8) "eight")
    ( (eq? w 9) "nine")
    ( (eq? w 'X) "ten")
    ( (eq? w 'J) "jack")
    ( (eq? w 'Q) "queen")
    ( (eq? w 'K) "king")
    ( (eq? w 'A) "ace")
    ))
```



```scheme

;---------------------------End of said Code
(define (higher-rank pick1 pick2)
  ( define rank1 (rank pick1))
  ( define rank2 (rank pick2))
  (cond
    ((eq? rank1 rank2)
     (higher-rank pick-a-card pick-a-card))
    ((> (the-value rank1) (the-value rank2))
     rank1)
    ((> (the-value rank2) (the-value rank1))
     rank2)
    ))

( trace higher-rank )

(define (classify-two-cards function)
  (define first-card (car function))
  (define second-card (cadr function))
  (define rank1 (car first-card))
  (define rank2 (car second-card))
  (define suit1 (cadr first-card))
  (define suit2 (cadr second-card))

  (cond
    ((and (eq? rank1 rank2) (eq? suit1 suit2))
     (display function) (display ":  ") (display (the-word rank1)) (display " pair flush"))
    ((and (eq? rank1 rank2) (not ( eq? suit1 suit2)))
     (display function) (display ":  ") (display (the-word rank1)) (display " pair "))
    ((and (> (the-value rank1) (the-value rank2)) (eq? suit1 suit2))
     (display function) (display ":  ") (display (the-word rank1)) (display " high flush"))
    ((and (> (the-value rank2) (the-value rank1)) (eq? suit1 suit2))
     (display function) (display ":  ") (display (the-word rank2)) (display " high flush"))
    ((and (= (- (the-value rank1) (the-value rank2)) 1) (eq? suit1 suit2))
     (display function) (display ":  ") (display (the-word rank1)) (display " straight flush"))
    ((and (= (- (the-value rank2) (the-value rank1)) 1) (eq? suit1 suit2))
     (display function) (display ":  ") (display (the-word rank2)) (display " straight flush"))
    ((and (= (- (the-value rank1) (the-value rank2)) 1) (not (eq? suit1 suit2)))
     (display function) (display ":  ") (display (the-word rank1)) (display " high straight "))
    ((and (= (- (the-value rank2) (the-value rank1)) 1) (not (eq? suit1 suit2)))
     (display function) (display ":  ") (display (the-word rank2)) (display " high straight "))
    ((and (> (the-value rank2) (the-value rank1)) (not (eq? suit1 suit2)) (not(= (- (the-value rank2)
     (the-value rank1)) 1)))
     (display function) (display ":  ") (display (the-word rank2)) (display " high "))
    ((and (> (the-value rank1) (the-value rank2)) (not (eq? suit1 suit2)) (not (= (- (the-value rank1)
     (the-value rank2)) 1)))
     (display function) (display ":  ") (display (the-word rank1)) (display " high "))
    ))

```



```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( classify-two-cards ( pick-two-cards ) )
((9 H) (Q C)):  queen high
> ( classify-two-cards ( pick-two-cards ) )
((J C) (A H)):  ace high
> ( classify-two-cards ( pick-two-cards ) )
((6 S) (2 S)):  six high flush
> ( classify-two-cards ( pick-two-cards ) )
((A S) (9 S)):  ace high flush
> ( classify-two-cards ( pick-two-cards ) )
((Q H) (A H)):  ace high flush
> ( classify-two-cards ( pick-two-cards ) )
((J C) (2 D)):  jack high
> ( classify-two-cards ( pick-two-cards ) )
((3 S) (J H)):  jack high
> ( classify-two-cards ( pick-two-cards ) )
((5 D) (A D)):  ace high flush
> ( classify-two-cards ( pick-two-cards ) )
((3 S) (A C)):  ace high
> ( classify-two-cards ( pick-two-cards ) )
((X S) (6 H)):  ten high
> ( classify-two-cards ( pick-two-cards ) )
((A H) (2 H)):  ace high flush
> ( classify-two-cards ( pick-two-cards ) )
((5 D) (6 C)):  six high
> ( classify-two-cards ( pick-two-cards ) )
((5 H) (7 S)):  seven high
> ( classify-two-cards ( pick-two-cards ) )
((9 C) (4 D)):  nine high
> ( classify-two-cards ( pick-two-cards ) )
((9 H) (3 C)):  nine high
> ( classify-two-cards ( pick-two-cards ) )
((3 H) (X C)):  ten high
> ( classify-two-cards ( pick-two-cards ) )
((Q C) (6 S)):  queen high
> ( classify-two-cards ( pick-two-cards ) )
((2 C) (4 S)):  four high
> ( classify-two-cards ( pick-two-cards ) )
((X C) (2 D)):  ten high
> ( classify-two-cards ( pick-two-cards ) )
((2 H) (Q C)):  queen high
>
```