Racket Programming Assignment #3: Lambda and Basic Lisp

By: Miguel Cruz

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 programs 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.

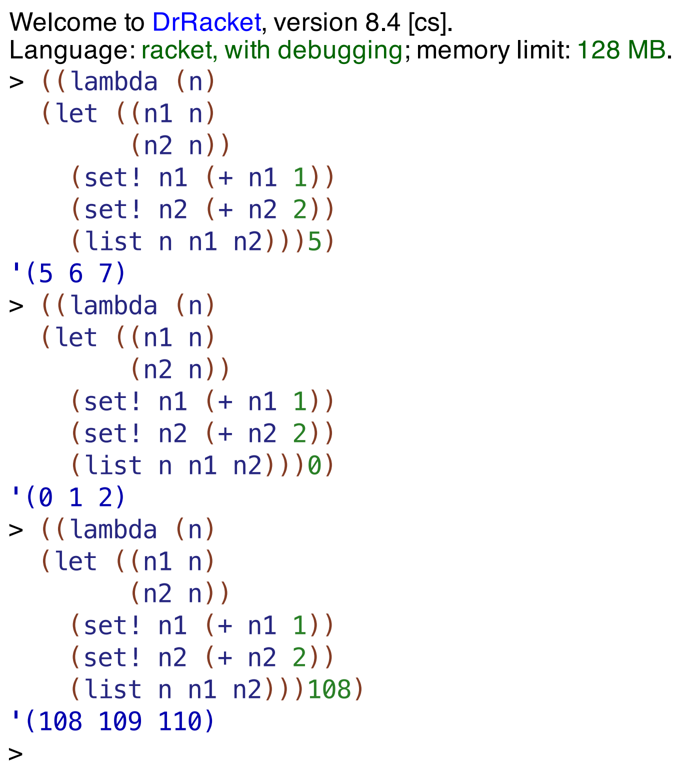
Task 1 - Lambda

Task 1a - Three ascending integers

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”.

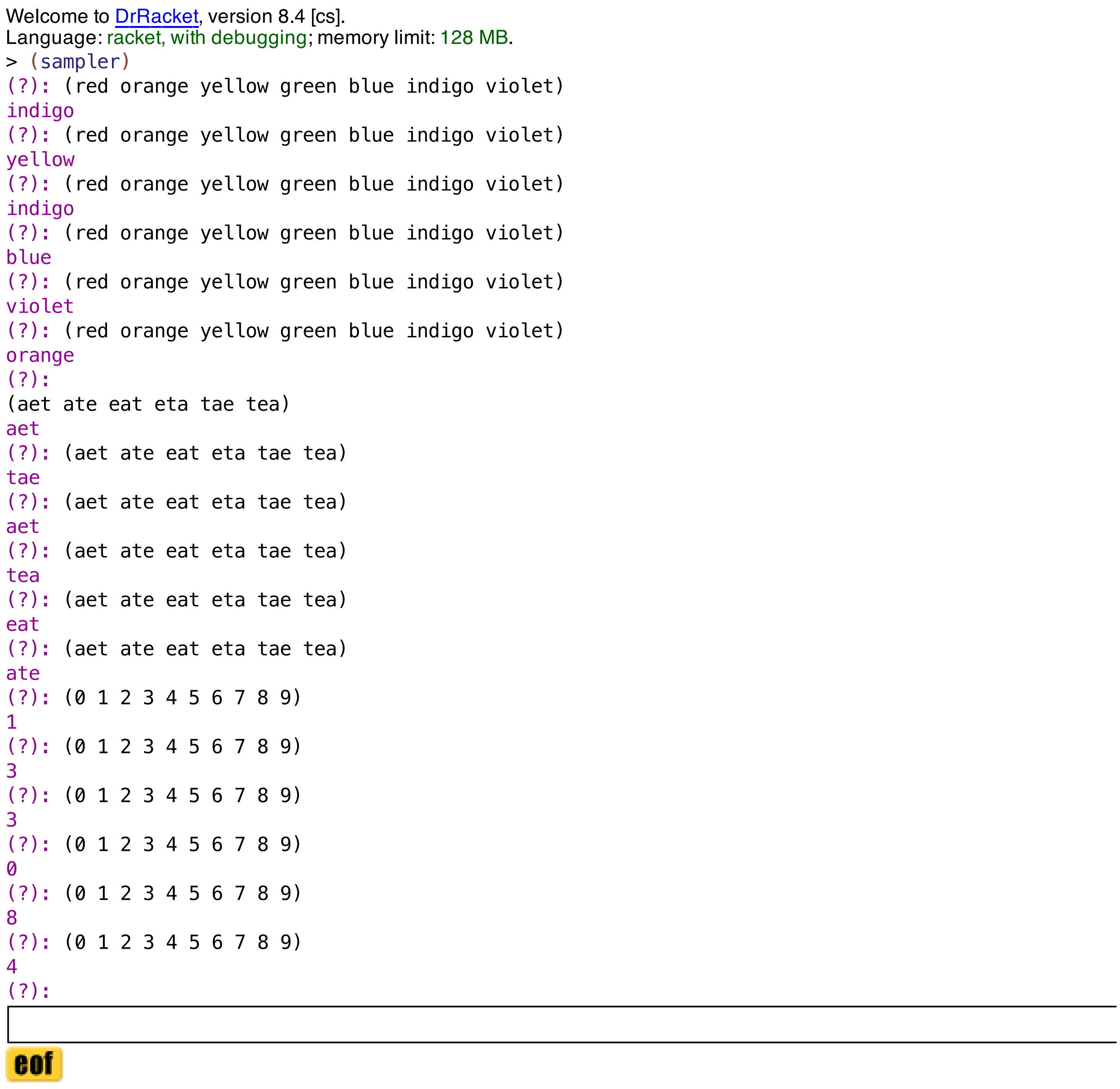
```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define languages '(racket prolog haskell rust))
> languages
'(racket prolog haskell rust)
> ( quote languages)
'languages
> ( car languages)
'racket
> ( cdr languages)
'(prolog haskell rust)
> ( car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

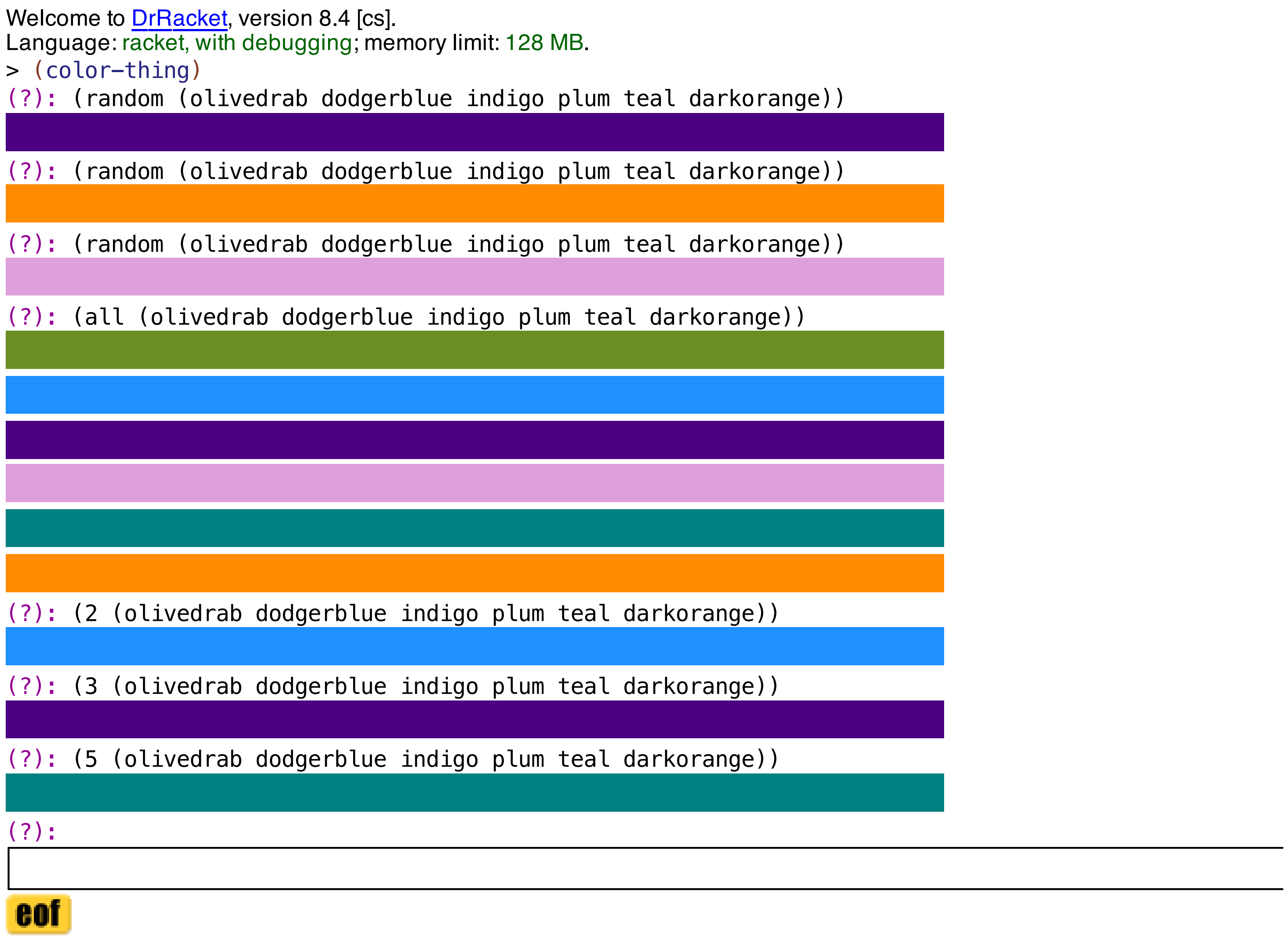
1. In a file called `sampler.rkt`, establish the “sampler” program that can be found in the penultimate section of Lesson 6.
2. Generate a demo by mimicking the demo which accompanies the “sampler” program in Lesson 6.



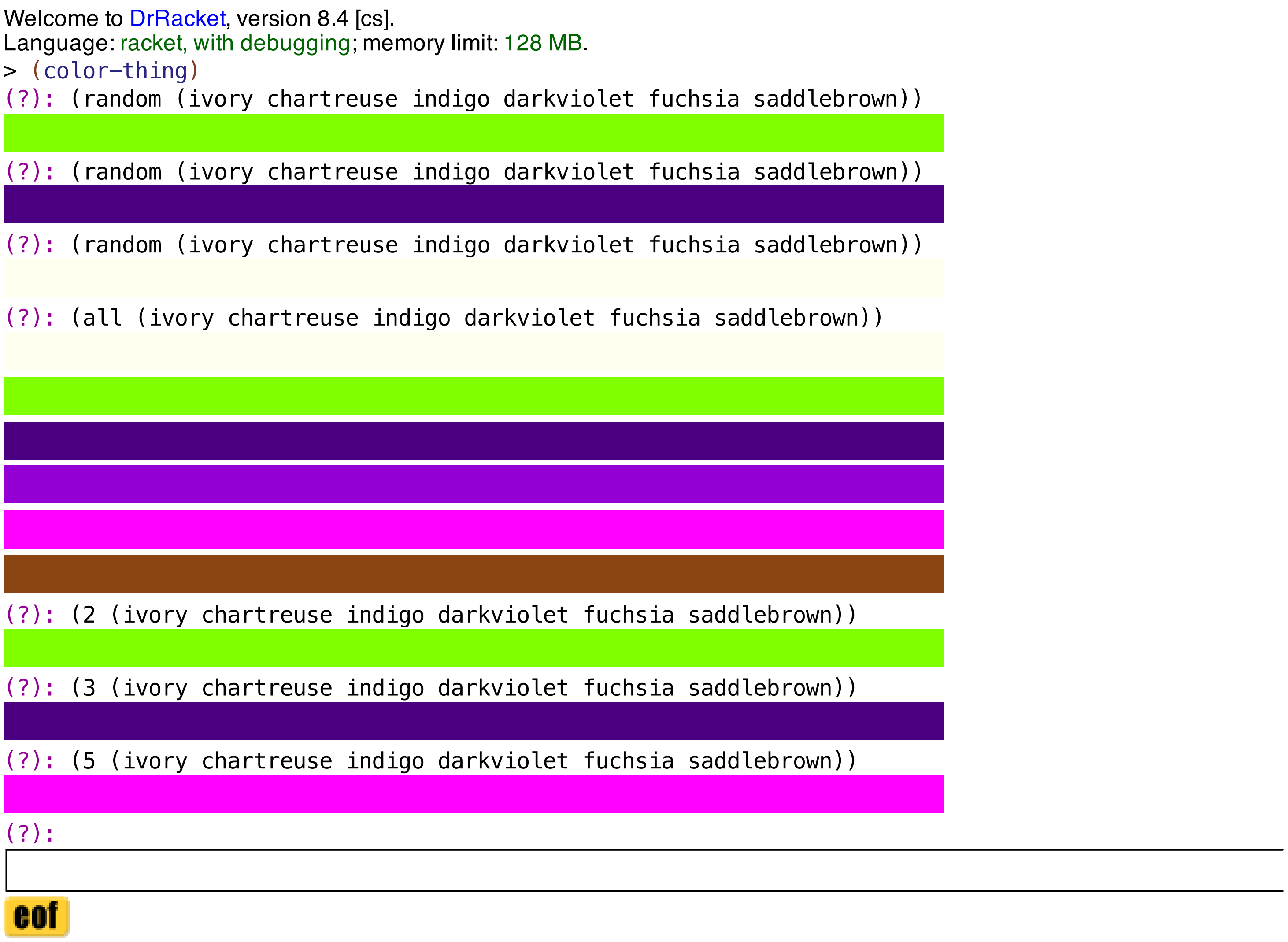
```
Welcome to DrRacket, version 8.4 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (sampler)
(?): (red orange yellow green blue indigo violet)
indigo
(?): (red orange yellow green blue indigo violet)
yellow
(?): (red orange yellow green blue indigo violet)
indigo
(?): (red orange yellow 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))
(?):
```



```
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
```

```racket
#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

1. Please establish a file called `cards.rkt` and place the code given within the section titled “Basic List Processing Example - Playing Cards” of Lesson 6 into the file.
2. Recreate the demo provided in that same part of the lesson.

Task 4b - Two Card Poker Classifier, IR Version

In this part of the assignment you will be lead in a step by step manner to program a two card poker hand classifier.

1. Please closely consider the the accompanying demo called "UR classifier demo", from which you should be able to infer the big picture of what you are expected to do for this part of your assignment.
2. Please establish a file called `classifier_ur.rkt`, copy and paste the code from your "cards.rkt" file into the newly established file, save it, and run it, by which mean just make sure that the code still works. Note: You will make use of some, but not all, of the functionality that you copied into this file as you proceed to develop the classifier program.
3. Within the classifier file, define a parameterless function called `pick-two-cards` which returns a list of two cards from a virgin deck of cards, assuring that the two cards are distinct. One way to do this would be to call `pick-a-card` twice, compare the resulting cards, return them in the form of a list if they are different, or recursively call this function if the should happen to be the same. Please see the accompanying demo called "Pick two cards demo" for assurance that you understand what this function is all about.
4. Recreate the pick two cards demo (except for the fact that your two card results will differ due the to randomness).
5. Within the classifier file, define a function called `higher-rank` taking two parameters, each one being a card, which returns the "highest rank" of the cards. There are lots of ways to do this. Pick one! Please see the accompanying demo called "Higher rank demo" for assurance that you understand what this function is all about.
6. Recreate the accompanying higher rank demo. Note: In order to do this you will need add a couple of extra lines of code to your file. Please add `( require racket/trace )` just after the language specification statement. Please add `( trace higher-rank )` at the end of the file. Do the demo. After you copy the demo, "comment out" the trace form at the end of the file.
7. Within the classifier file, define a function called `classify-two-cards-ur` taking one parameter, a list of two lists, each representing one card, which displays the two cards, followed by a colon, followed by the appropriate classification of the two cards. Note that there are five basic ways to classify two cards: as a pair, as a straight flush, as a straight, as a flush, and as a mere high card. The syntax of the classification expressions can be found in the demo. Please adhere to it, noting that the maximal rank of the pair leads off each classification expression. Please, once again, refer to the accompanying demo called "Higher rank demo" for details and clarification.
8. Please "recreate" the accompanying uniform representation classifier demo, subject to the fact that randomness will cause different pairs of cards to appear throughout the demo. That is, generate a demo which calls this function 20 times.

UR classifier demo

```
> ( classify-two-cards-ur ( pick-two-cards ) )
((J D) (4 C)):  J high
> ( classify-two-cards-ur ( pick-two-cards ) )
((Q C) (X S)):  Q high
> ( classify-two-cards-ur ( pick-two-cards ) )
((Q H) (J S)):  Q high straight
> ( classify-two-cards-ur ( pick-two-cards ) )
```

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

Pick two cards demo

```
> ( pick-two-cards )
'((9 D) (X S))
> ( pick-two-cards )
'((J S) (A H))
> ( pick-two-cards )
'((X D) (K H))
> ( pick-two-cards )
'((X C) (3 D))
> ( pick-two-cards )
'((9 D) (J S))
>
```

Higher rank demo

```
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(5 C) '(A D))
<'A
'A
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(6 D) '(8 H))
<8
8
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(4 S) '(5 D))
<5
5
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(8 H) '(A H))
<'A
'A
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(J H) '(X S))
<'J
'J
>
```

Task 4c - Two Card Poker Classifier

In this part of the assignment you are asked to change the classifier so that the output is a bit more natural.

1. Please closely consider the the accompanying demo called "Classifier demo", from which you should be able to infer the big picture of what you are expected to do for this part of your assignment.
2. Please establish a file called `classifier.rkt`, copy and paste the code from your `classifier_ur.rkt` file into the newly established file, save it, and run it, by which I mean just make sure that the code still works.
3. Within the newly created classifier file, define a function called `classify-two-cards` taking one parameter, a list of two lists, each representing one card, which displays the two cards, followed by a colon, followed by the more natural classification of the two cards. The method of classification is the same as before. The only difference is that the output will appear a bit more natural. The syntax of the more natural classification expressions can be found in the demo. Please adhere to it. Note: You can make a mess of this, or you can do it in a fairly clean manner. Please strive for the latter!
4. Please "recreate" the accompanying classifier demo, with the more expressive representation of the classifications. That is, generate a demo which calls this function 20 times.

Classifier demo

```
> ( classify-two-cards ( pick-two-cards ) )
((4 D) (J S)):  jack high
> ( classify-two-cards ( pick-two-cards ) )
((Q H) (8 C)):  queen high
> ( classify-two-cards ( pick-two-cards ) )
((2 H) (8 C)):  eight high
```

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

Additional Notes on your Solution Document

Craft a nicely structured solution document that contains:

1. A nice title, indicating that this is your second Racket assignment.
2. A nice learning abstract, which foreshadows the tasks that you are asked to complete in this assignment.
3. A section for Task 1, the task involving lambda functions, which contains subsections for each of the three required demos.
4. A section for Task 2, the task involving the referencers and constructors, which contains the demo.
5. A section for Task 3, the task involving the color thing, which contains two main subsections, one for the sampler code and demo, and one for the color thing interpreter code and the two required demos.
6. A section for Task 4, the task involving the two card poker hand classifier, which contains three main subsections. The first will contain code and demo for the "cards" program. The second will contain code and **three** demos for the "ur classifier" program. The third will contain code and demo for the "classifier" program.

Then post your document, in **pdf** format, to you web work site.

Due Date

Friday, March 7, 2022.