Clearly print your name as it appears on Canvas:

Clearly print your netid:

EECS-111 Exam 1 DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO	Part1	
	Part2	
	Part3	
	Total	

Part 1

Give the TYPE of the value returned by each of the following expressions. If more than one expression is given, assume that each one is executed, in order, and give the type of the value for the **last** expression. You may assume that the image procedures, including **iterated-overlay** and **iterated-beside**, have been defined.

- If it is a **primitive type** such as a number, string, Boolean or image (picture), just give the name of the type. So if the result is a number, just say "number."
- If it is a **record type (a struct)**, just give the name of the record type. For example, if it's an album object, just say "album"
- If it is a **list**
  - If all the elements of the list are the same type, say "(listof *type*)" where *type* is the type of data in the list. For example (list 1 2 3) is a (listof number).
  - If it is a list with different types of data, say (listof any)
  - If you know the result is specifically the empty list, which has no elements and therefore no element type, just say "empty list".
  - If you know the result is a list but you don't know the type of data in it, just say "list" and we will give you partial credit.
- If the result is a **procedure**, give its type signature, i.e. its argument and return types. In particular, write the type(s) of its argument(s) followed by an arrow and the type of its result. If the procedure accepts any type of value for an argument, just say "any". For example:
  - The type of the **abs** procedure is
    $number \rightarrow number$
  - The type of the **integer?** procedure is:
    $any \rightarrow Boolean$
  - The type of the **<** procedure is:
    $number\ number \rightarrow Boolean$
  - The type of the **square** procedure is
    $number\ string\ color \rightarrow image$
- If you know the expression's value is a **procedure**, but don't know its argument or return types, just say "procedure", and we will give you partial credit.
- If executing it would produce an **exception**, say "Exception". You do not have to explain what type of exception or why.

Clearly print your name as it appears on Canvas:
Clearly print your netid:

1.
```
(define mystery
       (lambda (x) (+ x 1)))
 (mystery 7)
```

number

2.
```
(+ (3) (+ 1 2))
```

exception

3.
```
; An album is...
; - (make-album string string string)
(define-struct album [title artist genre])
(define lib (list
               (make-album "Abbey Road"
                           "The Beatles"
                           "Rock")
               (make-album "Let It Be"
                           "The Beatles"
                           "Rock")))
(map album-title lib)
```

listof strings

4.
```
(lambda (r c)
     (circle r "outline" c))
```

number string → image
OR
number color → image

5.
```
(empty? (filter (lambda (y) (= y 47))
                (list 1 2 3)))
```

boolean

6.
```
(filter (lambda (p) (or (< (posn-x p) 10)
                        (< (posn-y p) 10)))
      (make-posn 20 30))
```

exception

7.
```
(iterated-overlay
   (lambda (x)
      (rotate x
            (square x "solid" "purple")))
   100)
```

image/picture

8.
```
(lambda (z)
   (ormap odd? z))
```

listof numbers → boolean

9.
```
; A rabbit is...
; - (make-rabbit number string)
(define-struct rabbit [weight food])
(make-rabbit 10 "apples")
```

rabbit
struct
record

10.
```
(rest (map odd? (list 1 2 3)))
```

listof boolean

Clearly print your name as it appears on Canvas:
Clearly print your netid:

Part 2

Each of the following questions shows some code being executed at the Racket prompt, along with the output or error it generated, and the intended output that the programmer wanted. Give the **correction** to the code to produce the desired result.

- Fix the code that's there; **don't rewrite it from scratch**. In grading, we're looking for evidence that you understand the bug in that particular code, not that you understand how to write new code.
- You do not need to provide an explanation, although you are free to do so if you like.
- It is sufficient to **write your correction on top of the existing code**; you **don't need to recopy** it.

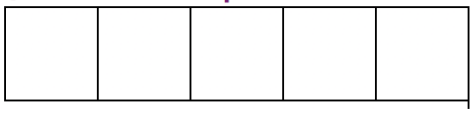
Question 1

Interaction	**Desired output**

> (define (contains-my-num? n lon)
    (if (empty? lon)
        true
        (or (= n (first lon))
            (contains-my-num? n (rest lon)))))
> (contains-my-num? 4 (list 1 2 3))
#true
>
	#false

```
(define (contains-my-num? n lon)
  (if (empty? lon)
      false
      (or (= n (first lon))
          (contains-my-num? n (rest lon)))))
```

Question 2

Interaction	**Desired output**
> (iterated-beside (square 50 "outline" "black") 5) iterated-beside: expects procedure, given #<image> >	

```
(iterated-beside (lambda (n)
                   (square 50 "outline" "black”))
                 5)
```

Clearly print your name as it appears on Canvas:
Clearly print your netid:

Question 3

Interaction	**Desired output**

> (define (sum-list_iterative lon)
    (local [(define (help alon partial_sum)
            (cond [(empty? alon) partial_sum]
                  [else   (help (rest alon))]))]
      (help lon 0)))
> (sum-list_iterative (list 1 2 3))
help: expects 2 arguments, but found only 1
>
	6
	(i.e. the sum of the list)

```
(define (sum-list_iterative lon)
  (local [(define (help alon partial_sum)
                      (cond [(empty? alon) partial_sum]
                            [else (help (rest alon) (+ (first alon) partial_sum]))]
         (help lon 0)))
```

Question 4

Interaction	Desired-output
> (define-struct snake [weight food]) > (define mysnakes (list (make-snake 10 "mice") (make-snake 5 "carrots") (make-snake 7 "grass"))) > (foldl 0 + (map snake-weight mysnakes)) foldl: first argument must be a function that expects two arguments, given 0 >	22 (i.e. the sum of the weights of the snakes)

```
(foldl + 0 (map snake-weight mysnakes))
```

Clearly print your name as it appears on Canvas:
Clearly print your netid:

Part 3

Each of the following questions shows some a procedure definition. In the space below the procedure definition, provide one valid test (check-expect) of the procedure.

Question 1

```
;; add2: Number -> Number
;; adds 2 to the given number
(define add2
    (lambda (x)
        (+ x 2)))
```

```
(check-expect (add2 5) 7)
```

Question 2

```
; a dillo is a
; - (make-dillo number boolean)
(define-struct dillo [weight dead?])

;; feed-dillo : dillo -> dillo
;; feeds a dillo a 2lb meal if its alive
(define (feed-dillo d)
   (make-dillo (if (dillo-dead? d)
                    (dillo-weight d)
                    (+ 2 (dillo-weight d)))
                (dillo-dead? d)))
```

```
(check-expect (feed-dillo (make-dillo 10 #false)) (make-dillo 12 #false))
```

```
(check-expect (feed-dillo (make-dillo 10 #false)) (make-dillo (+ 10 2)  #false))

(check-expect (feed-dillo (make-dillo 10 #true)) (make-dillo 10 #true))
```