Lab 6 – Functional Arguments

1. In class, we wrote two versions of the `any` function, shown below, which takes a predicate function and an input list, and returns true if *any* element in the list satisfies the predicate. The first version uses a `cond` expression, while the second version uses logical expressions (`and/or/not`) instead of a `cond`.

```
(define any
  (lambda (predicate? input-list)
   (cond
     [(null? input-list) #f]
     [(predicate? (car input-list)) #t]
     [else (any predicate? (cdr input-list))])))

(define any
  (lambda (predicate? input-list)
    (and (not (null? input-list))
         (or (predicate? (car input-list))
             (any predicate? (cdr input-list))))))
```

   We also wrote the `all` function, which takes a predicate function and returns true if *all* elements in the list satisfy the predicate. Rewrite the definition of **`all`** shown below using only logical expressions, without `cond`. Hint: this function returns true if *either* the input list is empty, *or* if the car element satisfies the predicate *and* all other elements in the list do so as well. Use the auto-tester program to test your `all` function by typing `(test: all)` at the Scheme prompt. You should use the auto-tester for the rest of your lab exercises as well, to verify that your functions are working correctly.

```
(define all
  (lambda (predicate? input-list)
    (cond
      [(null? input-list) #t]
      [(predicate? (car input-list)) (all predicate? (cdr input-list))]
      [else #f])))
```

2. In class, we wrote the function `keep`, which takes a predicate function and an input list, and returns a new list containing just the elements of the original input list that satisfy the predicate function. The next few exercises ask you to define new functions using `keep` as a helper.

   Write the function **`keep-odd`**, which takes a list of numbers and returns a new list containing just the odd numbers, by filling in the blank in the template below.

```
(define keep-odd
  (lambda (input-list)
    (keep _________________ input-list)))

(keep-odd '(2 3 4 5 7)) → (3 5 7)
(keep-odd ' (1 1 2 2 3 3 4 4 5 5)) → (1 1 3 3 5 5)
```

3. Write the function **`keep-even`**, which keeps just the even numbers, by filling in the template below.

```
(define keep-even
  (lambda (input-list)
    (keep _________________ input-list)))

(keep-even '(2 3 4 5 7)) → (2 4)
(keep-even ' (1 1 2 2 3 3 4 4 5 5)) → (2 2 4 4)
```

4. Write the function **`keep-big`**, which keeps only numbers bigger than 100, by filling in the template below using a lambda expression of the form `(lambda (x) _____ )`.

```
(define keep-big
  (lambda (input-list)
    (keep _________________ input-list)))
```

5. Write the function **`remove-all`**, which takes a symbol and an input list, and removes all occurrences of the specified symbol from the list, by filling in the template below with a lambda expression of the form `(lambda (x) _____ )`.

```
(define remove-all
  (lambda (symbol input-list)
    (keep _________________ input-list)))

(remove-all 'x '(a b x a b x x b x)) → (a b a b b)
(remove-all 'b '(a b x a b x x b x)) → (a x a x x x)
```

6. Write the function **`drop`**, which takes a predicate function and an input list, and removes all elements from the list that satisfy the given predicate function, by filling in the template below with a lambda expression of the form `(lambda (x) _____ )`.

```
(define drop
  (lambda (predicate? input-list)
    (keep _________________ input-list)))

(drop number? '(1 2 a b 3 c 4 d e)) → (a b c d e)
(drop symbol? '(1 2 a b 3 c 4 d e)) → (1 2 3 4)
(drop even? '(1 2 3 4 5)) → (1 3 5)
```

7. The next few exercises ask you to define new functions using `map` as a helper. Recall that `map` takes a one-argument function `f` and applies it to each element of a list, returning a new list containing all of the results. Write the function **`double-each`**, which takes a list of numbers and doubles each number, by filling in the template below.

```
(define double-each
  (lambda (input-list)
    (map (lambda (n) _________________ ) input-list)))

(double-each '(1 2 3 4 5)) → (2 4 6 8 10)
(double-each '(3 3 3 3)) → (6 6 6 6)
```

8. Write the function **`reciprocals`**, which takes a list of numbers and returns a new list of the reciprocals of each number, by filling in the template below.

```
(define reciprocals
  (lambda (input-list)
    (map (lambda (n) _________________ ) input-list)))

(reciprocals '(2 3 4 5 6)) → (1/2 1/3 1/4 1/5 1/6)
(reciprocals '(3 3 3 3)) → (1/3 1/3 1/3 1/3)
```

9. Write the function **`x-all`**, which takes a list of numbers and replaces each number by the symbol `x`, by filling in the template below.

```
(define x-all
  (lambda (input-list)
    (map (lambda (n) _________________ ) input-list)))

(x-all '(1 2 3 4 5)) → (x x x x x)
(x-all '(3 3 3 3)) → (x x x x)
```

10. Write the function **`x-odd`**, which takes a list of numbers and replaces each *odd number* by the symbol `x`, but leaves the even numbers as they are. Hint: use an `if` or `cond` expression in your lambda function.

```
(define x-odd
  (lambda (input-list)
    (map (lambda (n) _______________ ) input-list)))

(x-odd '(1 2 3 4 5)) → (x 2 x 4 x)
(x-odd '(3 3 3 3)) → (x x x x)
(x-odd '(2 4 6 8)) → (2 4 6 8)
```

11. Write the function **`classify-nums`**, which takes a list of numbers and replaces each odd number by the symbol `odd` and each even number by the symbol `even`.

```
(define classify-nums
  (lambda (input-list)
    (map (lambda (n) _______________ ) input-list)))

(classify-nums '(1 2 3 4 5)) → (odd even odd even odd)
(classify-nums '(3 3 3 3)) → (odd odd odd odd)
(classify-nums '(2 4 6 8)) → (even even even even)
```

12. Write the function **`swap`**, which takes an *old* symbol, a *new* symbol, and a list of symbols, and replaces each *old* symbol by the *new* symbol, and vice versa. Other symbols should remain as they are.

```
(define swap
  (lambda (old new input-list)
    (map (lambda (x) _______________ ) input-list)))

(swap 'red 'blue '(red fish blue fish red)) → (blue fish red fish blue)
(swap 'eggs 'ham '(green eggs and ham and eggs)) → (green ham and eggs and ham)
```

13. Write the function **`pair-up`**, which takes a symbol and an input list, and pairs up the symbol with each element of the input list. Hint: use the `list` function inside your lambda expression.

```
(define pair-up
  (lambda (symbol input-list)
    (map (lambda (x) _______________ ) input-list)))

(pair-up 'x '(a b c d)) → ((x a) (x b) (x c) (x d))
(pair-up 'a '(1 2 3 4 5)) → ((a 1) (a 2) (a 3) (a 4) (a 5))
```

14. Write the function **`firsts`**, which takes a list of 2-element inner lists, and returns a new list containing just the *first* element of each inner list.

```
(define firsts
  (lambda (input-list)
    (map (lambda (x) _______________ ) input-list)))

(firsts '((red hot) (chili dogs))) → (red chili)
(firsts '((spanish paella) (red wine) (salsa beans))) → (spanish red salsa)
```

15. Write the function **`seconds`**, which takes a list of 2-element inner lists, and returns a new list containing just the *second* element of each inner list.

```
(define seconds
  (lambda (input-list)
    (map (lambda (x) _______________ ) input-list)))

(seconds '((red hot) (chili dogs))) → (hot dogs)
(seconds '((spanish paella) (red wine) (salsa beans))) → (paella wine beans)
```

16. In class, we wrote the function `reduce`, shown below, which captures the general pattern of recursion over a list of elements.

```
(define reduce
  (lambda (input-list base-value combiner)
    (cond
      [(null? input-list) base-value]
      [else (combiner (car input-list)
                      (reduce (cdr input-list) base-value combiner))])))
```

The following three functions all share the same underlying pattern. The only difference between them is the value returned in the base case, and the way in which the car element is combined with the result of the recursion on the cdr of the input list. The function **`length`** counts the number of elements in the input list. The function **`add-to-end`** adds a new symbol to the end of a list. The function **`count`** counts the number of occurrences of a symbol in a list.

```
(define length
  (lambda (input-list)
    (cond
      [(null? input-list) 0]
      [else (+ 1 (length (cdr input-list)))])))


(define add-to-end
  (lambda (symbol input-list)
    (cond
      [(null? input-list) (list symbol)]
      [else (cons (car input-list) (add-to-end symbol (cdr input-list)))])))


(define count
  (lambda (symbol input-list)
    (cond
      [(null? input-list) 0]
      [(equal? (car input-list) symbol) (+ 1 (count symbol (cdr input-list)))]
      [else (count symbol (cdr input-list))])))
```

Rewrite each of these functions in terms of `reduce` by filling in the templates below with the appropriate base-case value in slot 1 and an expression in slot 2 that will combine the car element with the result of the recursion in an appropriate way. In the lambda expressions, the `x` parameter represents the car element, and the `r` parameter represents the result of calling the recursion on the cdr of the input list.

```
(define length
  (lambda (input-list)
    (reduce input-list  ______1______  (lambda (x r) ________2________ ))))


(define add-to-end
  (lambda (symbol input-list)
    (reduce input-list  ______1______  (lambda (x r) ________2________ ))))


(define count
  (lambda (symbol input-list)
    (reduce input-list  ______1______  (lambda (x r) ________2________ ))))
```

17. **EXTRA CREDIT CHALLENGE:** Define new versions of the functions `map`, `keep`, and `drop`, called **`map2`**, **`keep2`**, and **`drop2`**, in terms of `reduce` by filling in the templates below appropriately.

```
(define map2
  (lambda (f input-list)
    (reduce input-list  _______________  (lambda (x r) ___________________ ))))

(define keep2
  (lambda (predicate? input-list)
    (reduce input-list  _______________  (lambda (x r) ___________________ ))))

(define drop2
  (lambda (predicate? input-list)
    (reduce input-list  _______________  (lambda (x r) ___________________ ))))
```