Practice Exercise: Racket 2

For this exercise, you may use **ONLY** the expressions included in the slides (both sets 1 and 2), unless otherwise indicated, or you may create your own helper functions.

1. Let ***b*** be a **non-null list** containing at least two atoms. Write a function ***double-second*** that evaluates to a new list obtained from the list ***b*** by 'doubling' the second atom in ***b***.
   Example:
   (double-second '(1 2)) => '(1 2 2)
   (double-second '(1 2 3 4)) => '(1 2 2 3 4)

```
(define double-second
  (lambda (b)
   (cons (first b)
      (cons (first (rest b)) (rest b)))))
```

2. Let ***x*** and ***y*** be **lists**. Write a function ***func1*** that evaluates to the first element of the list ***x*** if the list ***x*** is non-null, or to the cons of ***x*** onto the list ***y*** otherwise.
   Example:
   (func1 '() '(1 2 3)) => '(() 1 2 3)
   (func1 '(1 2 3) '(4 5 6 7)) => 1

```
(define func1
  (lambda (x y)
   (cond
    [(empty? x) (cons x y)]
    [else (first x)])))
```

3. Let ***s*** be a **list**. Write a function ***func2*** that evaluates to the list of the first three elements (in order) if ***s*** contains three or more elements, and evaluates to the null list otherwise.
   Example:
   (func2 '()) => '()
   (func2 '(1)) => '()
   (func2 '(1 2)) => '()
   (func2 '(1 2 3)) => '(1 2 3)
   (func2 '(1 2 3 4 5 6)) => '(1 2 3)

```
(define func2
 (lambda (s)
  (cond
   [(empty? s) '()]
   [(empty? (rest s)) '()]
   [(empty? (rest (rest s))) '()]
   [else (cons (first s)
            (cons (first (rest s))
               (cons (first (rest (rest s))) '())))])))
```

4. Let ***w*** be a **non-null list** containing at least three elements. Write a function ***new-list*** that evaluates to a new list obtained from ***w*** by exchanging its first and third elements.
Example:
(new-list '(1 2 3)) => '(3 2 1)
(new-list '(1 2 3 4)) => '(3 2 1 4)
(new-list '((1 2) (3) (4))) => '((4) (3) (1 2))

```
(define new-list
  (lambda (w)
    (cons (first (rest (rest w)))
      (cons (first (rest w))
         (cons (first w) (rest (rest (rest w))))))))
```

5. Define a function ***third-element*** that takes a **list *m*** and returns its third element. If there is no third element, return the empty list.
Example:
(third-element '()) => '()
(third-element '(1)) => '()
(third-element '(1 2)) => '()
(third-element '(1 2 3 4)) => 3
(third-element '(1 2 3 4 5)) => 3
(third-element '((1 2) (3 4) (5 6) (7 8))) => '(5 6)

```
(define third-element
 (lambda (lis)
  (cond
   [(empty? lis) '()]
   [(empty? (rest lis)) '()]
   [(empty? (rest (rest lis))) '()]
   [else (first (rest (rest lis)))])))
```

6. Write a **recursive** function ***dupla*** that takes two inputs, a **data expression *a*** and a **list *s***, and outputs a list that contains the data expression repeated as many times as the number of elements in the list ***s***.
Example:
(dupla 'a '()) => '()
(dupla 'a '(one)) => '(a)
(dupla 'a '(one (two))) => '(a a)
(dupla 'a '(one () (three))) => '(a a a)
(dupla '() '(1 2 3 4 5 6 7 8)) => '(() () () () () () () ())

```
(define dupla
 (lambda (a s)
  (cond
   [(empty? s) '()]
   [else (cons a (dupla a (rest s)))])))
```

7. Write a **recursive** function ***double*** that takes two inputs, a **data expression *a*** and a **list *s***, and doubles the first occurrence of the data expression ***a*** in the list ***s***.
   Example:
   (double 'b '(a b c)) => '(a b b c)
   (double '((2 3)) '((1) ((2 3)) 4)) => '((1) ((2 3)) ((2 3)) 4)

```
(define double
 (lambda (a s)
  (cond
   [(empty? s) '()]
   [(equal? (first s) a) (cons a s)]
   [else (cons (first s) (double a (rest s)))])))
```

8. Define a **recursive** function ***cons-to-end*** that accepts two inputs, the first being any **data expression *a*** and the second being any **list *s***, and output a list that is the second input with the first input inserted as the last data expression.
   Example:
   (cons-to-end 'a '()) => '(a)
   (cons-to-end 'a '(b c d)) => '(b c d a)
   (cons-to-end '(a (b) c) '(x y z)) => '(x y z (a (b) c))

```
(define cons-to-end
 (lambda (a s)
  (cond
   [(empty? s) (cons a s)]
   {else (cons (first s) (cons-to-end a (rest s)))})))
```

9. Write a definition for the **recursive** function ***occur*** that takes a **data expression *a*** and a **list *s*** and returns the number of times that the data expression ***a*** appears in the list ***s***.
   Example:
   (occur '() '(1 () 2 () () 3)) =>3
   (occur 1 '(1 2 1 ((3 1)) 4 1)) => 3 (note that it only looks at whole elements in the list)
   (occur '((2)) '(1 ((2)) 3)) => 1

```
(define occur
 (lambda (a s)
  (cond
   [(empty? s) 0]
   [(equal? a (first s)) (+ 1 (occur a (rest s)))]
   [else (occur a (rest s))])))
```

10. (This is similar to the function above, but it looks inside the sublists as well) Write a **recursive** function ***atom-occur?***, which takes two inputs, an **atom *a*** and a **list *s***, and outputs the Boolean **true** if and only if ***a*** appears **somewhere** within ***s***, either as one of the data expressions in ***s***, or as one of the data expression in one of the data expression in ***s***, or…, and so on.
    Example:
    (atom-occur? 'a '((x y (p q (a b) r)) z)) => #t
    (atom-occur? 'm '(x (y p (1 a (b 4)) z))) => #f

```
(define atom?
  (lambda (b)
    (cond
      [(list? b) false]
      [else true])))

(define atom-occur?
  (lambda (a b)
    (cond
      [(empty? b) false]
      [(and (atom? (first b))  (equal? a (first b))) true]
      [(atom? (first b)) (atom-occur? a (rest b))]
      [else (or (atom-occur? a (first b)) (atom-occur? a (rest b)))])))
```

Save your file as **a250_r2_yourlastname_yourfirstname** and drop it in the **Q drive**, **DO NOT ZIP THE FILE**.