PS3 [Answers to the Questions]

Practitioner: Mobin Kheibary [994421017]

Supervisor: Dr. Ehsan Shoja

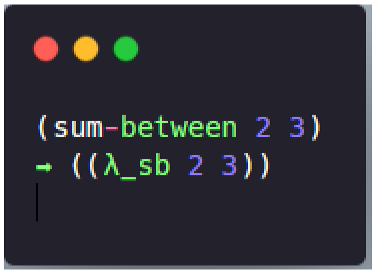
Step-by-step Solution for *Section 1*:

Lois's semantics for `if` indeed change the behavior of the language significantly. In the standard semantics of Racket (and most other programming languages), `if` expressions use short-circuit evaluation. This means that only the condition is evaluated first, and based on that, only one of the then and else expressions are evaluated. This is an important feature that allows programs to avoid unnecessary computations, or computations that would result in an error.

If we were to switch to Lois's semantics, the sum-between function would not work correctly. The problem lies in the recursive nature of sum-between. In Lois's semantics, before any `if` statement can be reduced, all three of its arguments must be evaluated. This means that each recursive call in the else clause would need to be evaluated before the `if` could be reduced, leading to infinite recursion.

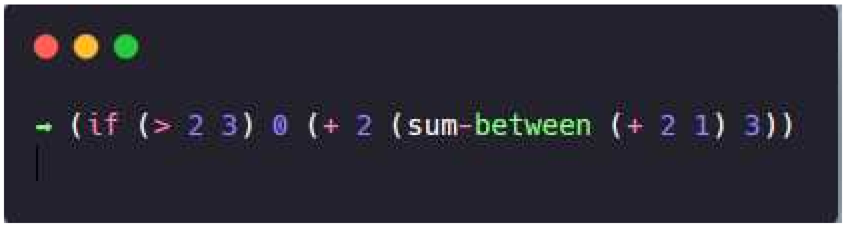
Let's demonstrate this by taking an example of `(sum-between 2 3)`:

Using Lois's semantics for `if`,



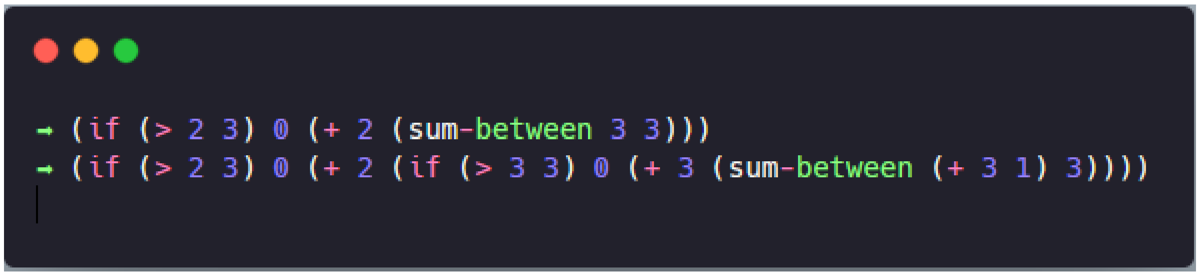
```
(sum-between 2 3)
→ ((λ_sb 2 3))
```

`λ_sb` is `(lambda (lo hi) (if (> lo hi) 0 (+ lo (sum-between (+ lo 1) hi))))`, so



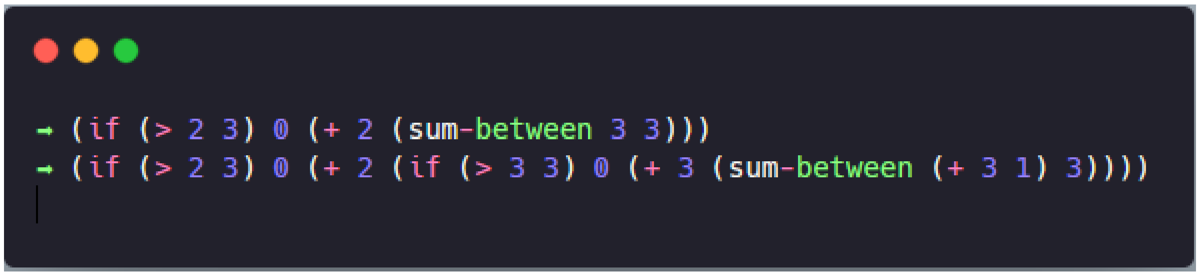
```
→ (if (> 2 3) 0 (+ 2 (sum-between (+ 2 1) 3))
```

According to Lois's rules, we have to evaluate `(sum-between (+ 2 1) 3)` before reducing the `if` expression.



```
→ (if (> 2 3) 0 (+ 2 (sum-between 3 3)))
→ (if (> 2 3) 0 (+ 2 (if (> 3 3) 0 (+ 3 (sum-between (+ 3 1) 3)))))
```

Again, the recursive call `(sum-between (+ 3 1) 3)` must be evaluated.



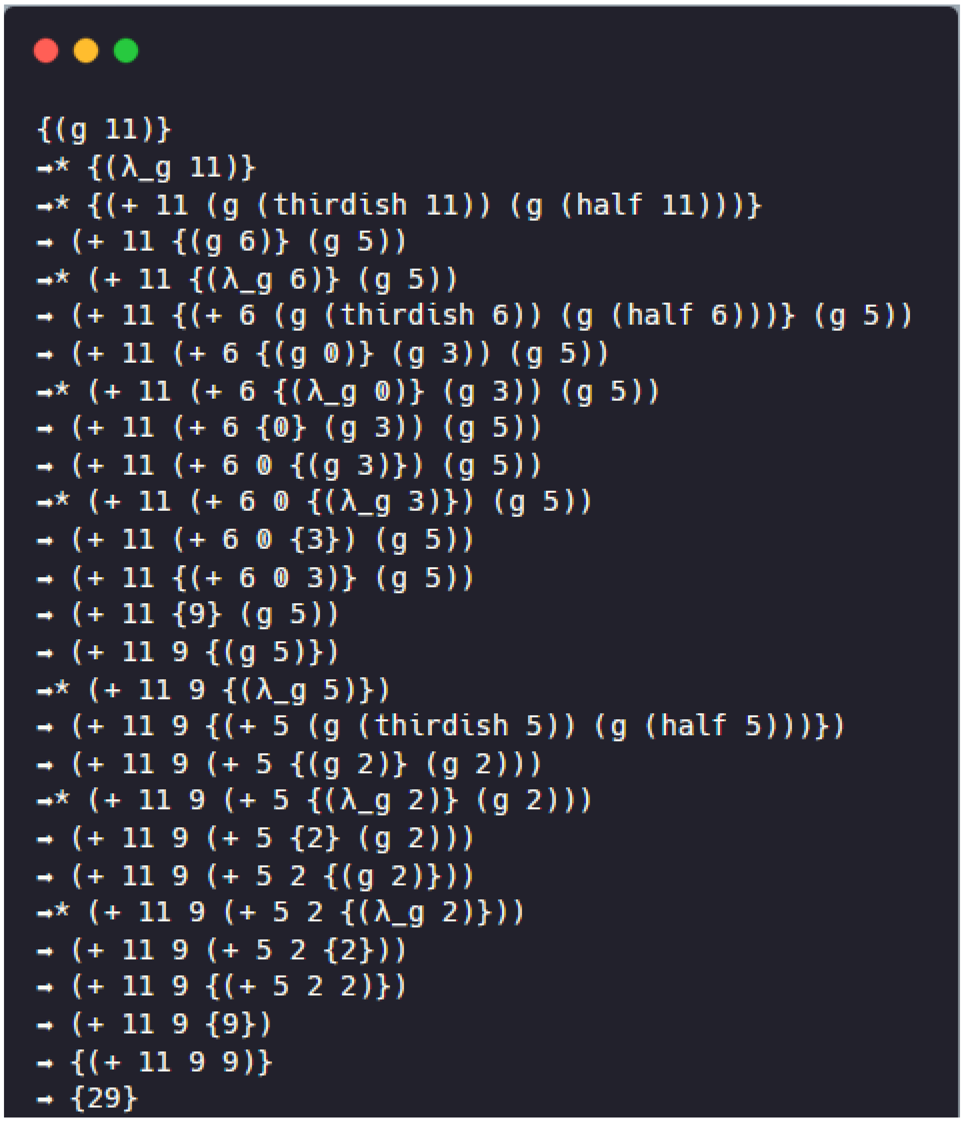
```
→ (if (> 2 3) 0 (+ 2 (sum-between 3 3)))
→ (if (> 2 3) 0 (+ 2 (if (> 3 3) 0 (+ 3 (sum-between (+ 3 1) 3)))))
```

And this will keep on going infinitely because we are forced to evaluate the `else` part of the `if` expression, which in turn calls `sum-between` recursively. This demonstrates the problem with Lois’s semantics for `if`. Thus, Lois's `if` semantics are not suitable for functions that utilize recursion in the way `sum-between` does.

Step-by-step Solution for *Section 2*:

1. Small-step semantics derivation for (g 11)

Let's abbreviate the function `g` as `λ_g`. Also, `thirdish(11)` results in 6, and `half(11)` results in 5. Using these, we can proceed with the derivation:



```
{(g 11)}
→* {(λ_g 11)}
→* {(+ 11 (g (thirdish 11)) (g (half 11)))}
→ (+ 11 {(g 6)} (g 5))
→* (+ 11 {(λ_g 6)} (g 5))
→ (+ 11 {(+ 6 (g (thirdish 6)) (g (half 6)))} (g 5))
→ (+ 11 (+ 6 {(g 0)} (g 3)) (g 5))
→* (+ 11 (+ 6 {(λ_g 0)} (g 3)) (g 5))
→ (+ 11 (+ 6 {0} (g 3)) (g 5))
→ (+ 11 (+ 6 0 {(g 3)}) (g 5))
→* (+ 11 (+ 6 0 {(λ_g 3)}) (g 5))
→ (+ 11 (+ 6 0 {3}) (g 5))
→ (+ 11 {(+ 6 0 3)} (g 5))
→ (+ 11 {9} (g 5))
→ (+ 11 9 {(g 5)})
→* (+ 11 9 {(λ_g 5)})
→ (+ 11 9 {(+ 5 (g (thirdish 5)) (g (half 5)))})
→ (+ 11 9 (+ 5 {(g 2)} (g 2)))
→* (+ 11 9 (+ 5 {(λ_g 2)} (g 2)))
→ (+ 11 9 (+ 5 {2} (g 2)))
→ (+ 11 9 (+ 5 2 {(g 2)}))
→* (+ 11 9 (+ 5 2 {(λ_g 2)}))
→ (+ 11 9 (+ 5 2 {2}))
→ (+ 11 9 {(+ 5 2 2)})
→ (+ 11 9 {9})
→ {(+ 11 9 9)}
→ {29}
```

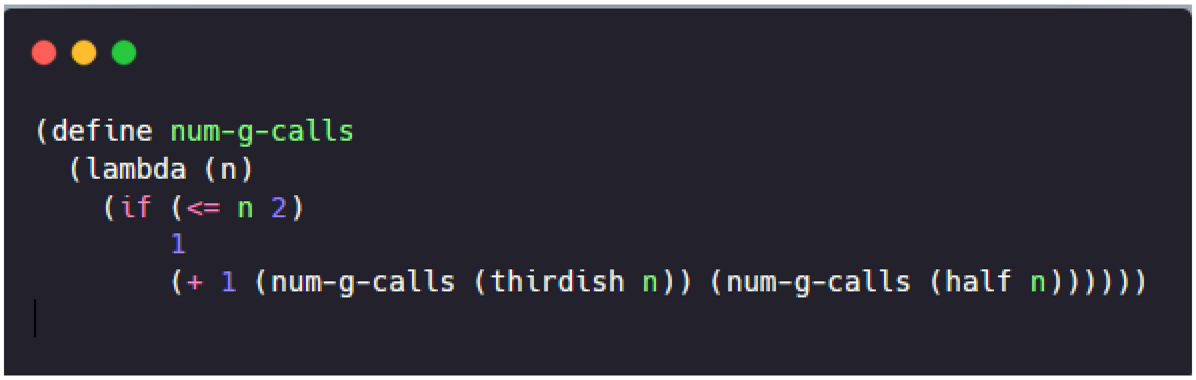
2. The number of g calls in the evaluation of (g 11)

The total number of `g` calls during this evaluationn is 7. The calls are as follows: `g 11`, `g 6`, `g 0`, `g 3`, `g 5`, `g 2` and `g 2` (again).

3. The maximum stack depth in the evaluation of (g 11)

The maximum stack depth, as measured by the number of nested `+` operations, is 3 during the evaluation of `(+ 11 (+ 6 0 3) (g 5))`.

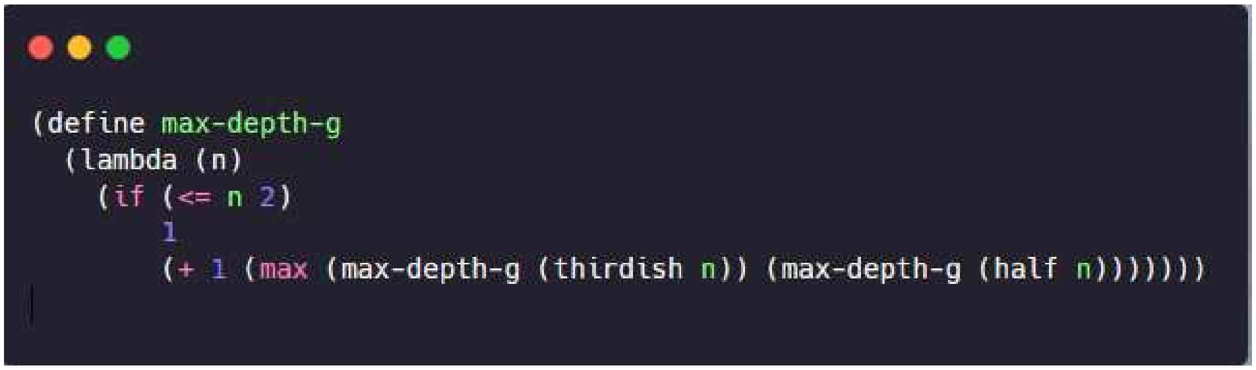
4. Recursive function to calculate number of g calls



```
(define num-g-calls
  (lambda (n)
    (if (<= n 2)
        1
        (+ 1 (num-g-calls (thirdish n)) (num-g-calls (half n))))))
```

Now, we can use the above `num-g-calls` function to get the number of times `g` is called when `n` is 11.

5. Recursive function to calculate maximum stack depth



```
(define max-depth-g
  (lambda (n)
    (if (<= n 2)
        1
        (+ 1 (max (max-depth-g (thirdish n)) (max-depth-g (half n)))))))
```

Now, we can use the above `max-depth-g` function to get the maximum stack depth when `n` is 11.

Step-by-step Solution for *Section 3*:

Part (a):

The expressions to extract each number from the list a using car and cdr are:

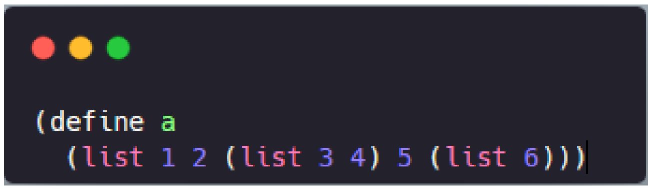
1. (car a) or (car (cdr (cdr (cdr a)))) or (car (cdr (cdr (cdr (cdr (cdr a))))))
2. (car (cdr a)) or (car (cdr (cdr (cdr (cdr a))))))
3. (car (cdr (cdr a))) or (car (cdr (cdr (cdr (cdr (cdr a))))))
4. (car (cdr (cdr (cdr a)))) or (car (cdr (cdr (cdr (cdr (cdr (cdr a)))))))
5. (car (cdr (cdr (cdr (cdr a)))))
6. (car (cdr (cdr (cdr (cdr (cdr a))))))

Part (b):

The printed representation of a would be (1 2 (3 4) 5 (6)).

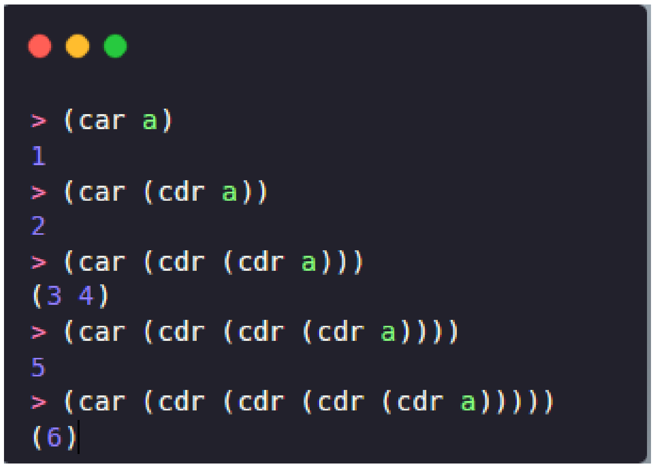
Part (c):

Here's a Racket expression using cons, list, and the numbers 1 through 6 to create the structure depicted in the diagram:



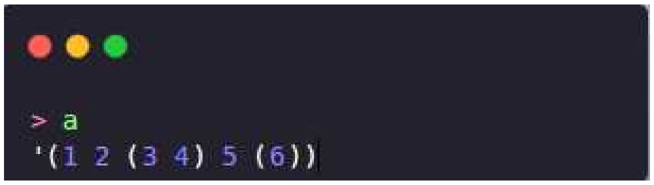
```
(define a
  (list 1 2 (list 3 4) 5 (list 6)))
```

Note that we have used list to create the nested lists (3 4) and(6) for readability, while using cons would also be possible. We can now test the expressions from Part (a) and verify that they extract the correct numbers from the list a:



```
> (car a)
1
> (car (cdr a))
2
> (car (cdr (cdr a)))
(3 4)
> (car (cdr (cdr (cdr a))))
5
> (car (cdr (cdr (cdr (cdr a)))))
(6)
```

We can also verify that the printed representation of a matches the original box-and-pointer diagram:

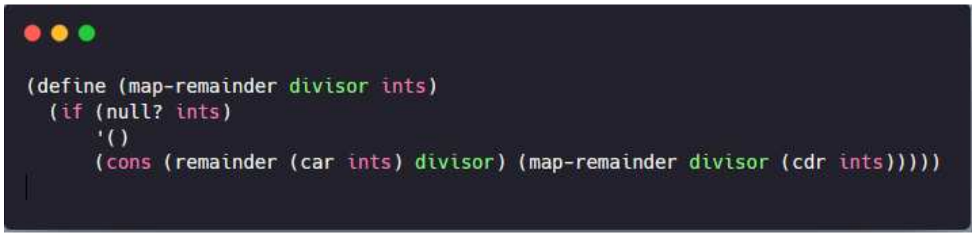


```
> a
'(1 2 (3 4) 5 (6))
```

Step-by-step Solution for *Section 4*:

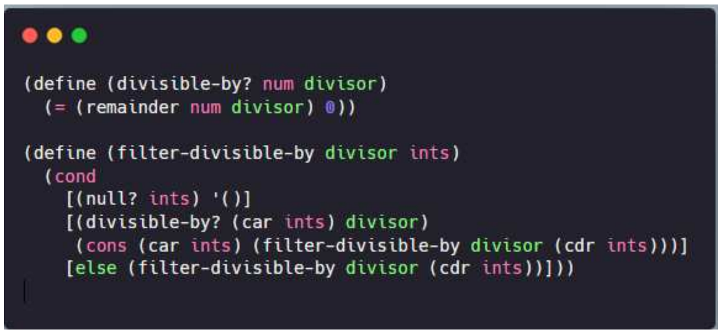
Sure, let's define these Racket functions one by one:

1. `map-remainder` function:



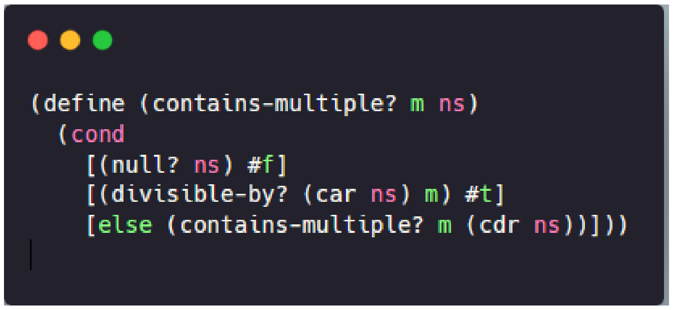
```
(define (map-remainder divisor ints)
  (if (null? ints)
      '()
      (cons (remainder (car ints) divisor) (map-remainder divisor (cdr ints)))))
```

2. `filter-divisible-by` function:



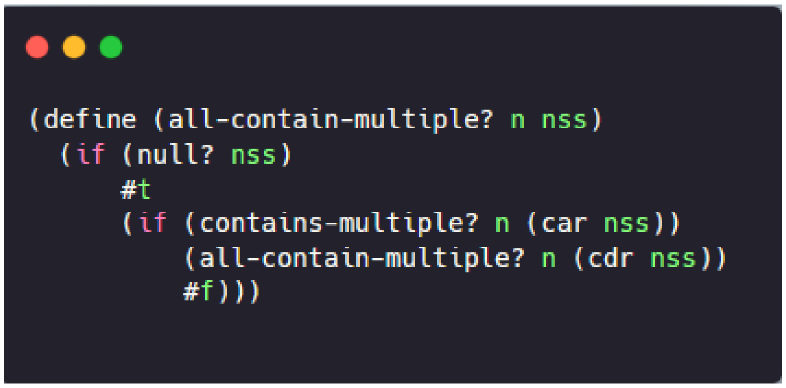
```
(define (divisible-by? num divisor)
  (= (remainder num divisor) 0))

(define (filter-divisible-by divisor ints)
  (cond
    [(null? ints) '()]
    [(divisible-by? (car ints) divisor)
     (cons (car ints) (filter-divisible-by divisor (cdr ints)))]
    [else (filter-divisible-by divisor (cdr ints))]))
```

3. `contains-multiple?` function:



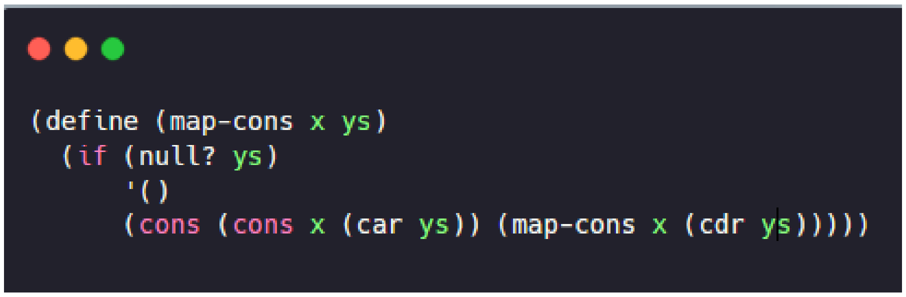
```
(define (contains-multiple? m ns)
  (cond
    [(null? ns) #f]
    [(divisible-by? (car ns) m) #t]
    [else (contains-multiple? m (cdr ns))]))
```

4. `all-contain-multiple?` function:



```
(define (all-contain-multiple? n nss)
  (if (null? nss)
      #t
      (if (contains-multiple? n (car nss))
          (all-contain-multiple? n (cdr nss))
          #f)))
```

5. `map-cons` function:



```
(define (map-cons x ys)
  (if (null? ys)
      '()
      (cons (cons x (car ys)) (map-cons x (cdr ys)))))
```

Step-by-step Solution for *Section 5*:

Let's tackle them one by one.

Problem 1: Cartesian Product

In this problem, we are tasked to create a function `my-cartesian-product` which will produce a Cartesian product of two lists. A Cartesian product of two sets A and B, denoted by A × B, is the set of all ordered pairs (a, b) where a is in A and b is in B.

Suppose L is '("a" "b" "c") in the function call (my-cartesian-product '("a" "b" "c") '(3 4))

i. (my-cartesian-product '("a" "b" "c") '(3 4)) should return '(("a" . 3) ("a" . 4) ("b" . 3) ("b" . 4) ("c" . 3) ("c" . 4))

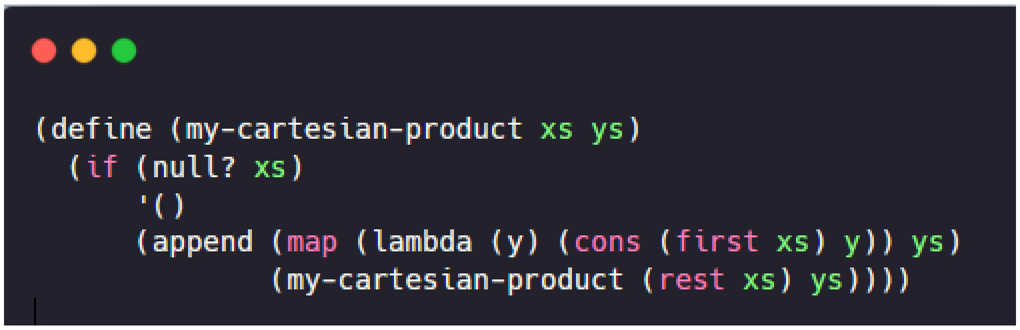
ii. (rest L) is '("b" "c"), and (my-cartesian-product '("b" "c") '(3 4)) should return '(("b" . 3) ("b" . 4) ("c" . 3) ("c" . 4))

iii. (first L) is "a". We need to create a pair of "a" with every element in '(3 4) which yields '(("a" . 3) ("a" . 4)). The way to combine this result with the result of ii is to use Racket's `append` function: (append '(("a" . 3) ("a" . 4)) '(("b" . 3) ("b" . 4) ("c" . 3) ("c" . 4)))

iv. For (my-cartesian-product xs ys), the generalization of (iii) is the general case (append (map (lambda (y) (cons (first xs) y)) ys) (my-cartesian-product (rest xs) ys))

(my-cartesian-product '() '(3 4)) should return '(). In general (my-cartesian-product '() ys) should return '().

Now, let's combine these results into the function definition:



```
(define (my-cartesian-product xs ys)
  (if (null? xs)
      '()
      (append (map (lambda (y) (cons (first xs) y)) ys)
              (my-cartesian-product (rest xs) ys))))
```

Problem 2: Alts

In this problem, we are tasked to create a function `alts` that will return a two-element list, one with the elements at even positions and the other with the elements at odd positions.

Suppose L is '(1 2 3 4 5 6) in the function call (alts '(1 2 3 4 5 6)):

i. (alts '(1 2 3 4 5 6)) should return '((1 3 5) (2 4 6)).

ii. (rest L) is '(2 3 4 5 6), and (alts '(2 3 4 5 6)) should return '((2 4 6) (3 5)).

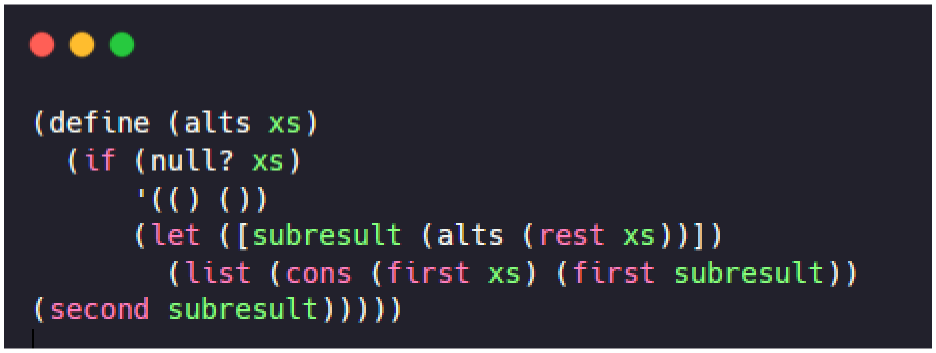
iii. (first L) is 1. We have to add 1 to the list of even-indexed elements and

keep the list of odd-indexed elements the same. The result is '((1 2 4 6) (3 5)). This can be achieved by using `cons` to add 1 to the list of even-indexed elements and `list` to create the final two-element list.

iv. For (alts xs), the generalization of (iii) is the general case (list (cons (first xs) (first (alts (rest xs)))) (second (alts (rest xs))))

(alts '()) should return '(() ()). In general, (alts '()) should return '(() ()).

Now, let's combine these results into the function definition:

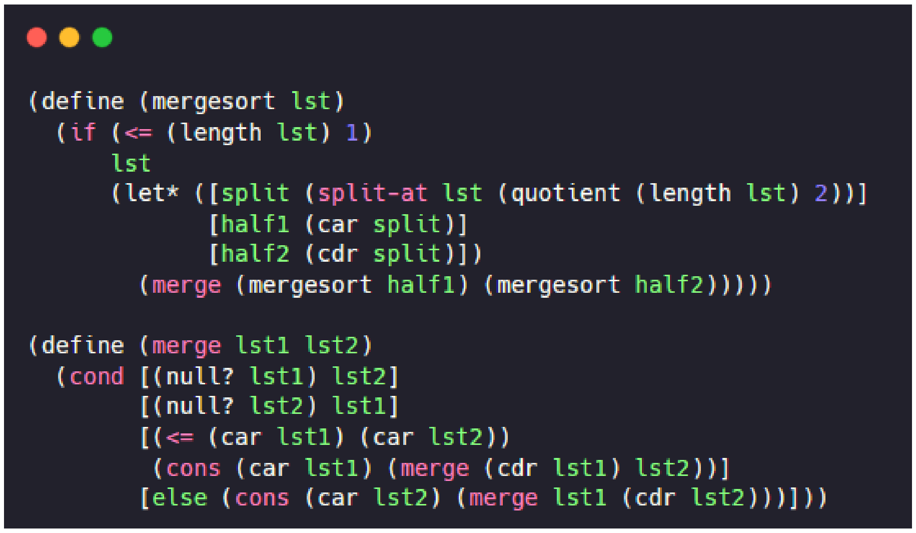


```
(define (alts xs)
  (if (null? xs)
      '(() ())
      (let ([subresult (alts (rest xs))])
        (list (cons (first xs) (first subresult))
(second subresult)))))
```

Problem 3: Mergesort

Mergesort is a divide and conquer algorithm that divides the unsorted list into n sublists, each containing one element (a list of one element is considered sorted), and then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining. This results in a sorted list.

Here's an implementation of mergesort in Racket:

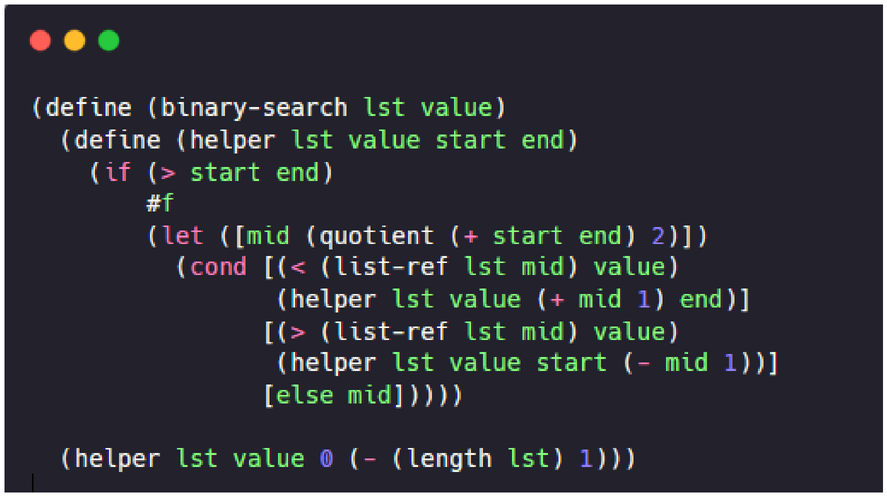


```
(define (mergesort lst)
  (if (<= (length lst) 1)
      lst
      (let* ([split (split-at lst (quotient (length lst) 2))]
             [half1 (car split)]
             [half2 (cdr split)])
        (merge (mergesort half1) (mergesort half2)))))

(define (merge lst1 lst2)
  (cond [(null? lst1) lst2]
        [(null? lst2) lst1]
        [(<= (car lst1) (car lst2))
         (cons (car lst1) (merge (cdr lst1) lst2))]
        [else (cons (car lst2) (merge lst1 (cdr lst2)))]))
```

Problem 4: Binary Search

Binary Search is a search algorithm that finds the position of a target value within a sorted array or list. It compares the target value to the middle element of the array/list and based on the comparison, it decides whether the search continues in the lower half or upper half of the array/list.

Here's an implementation of binary search in Racket:

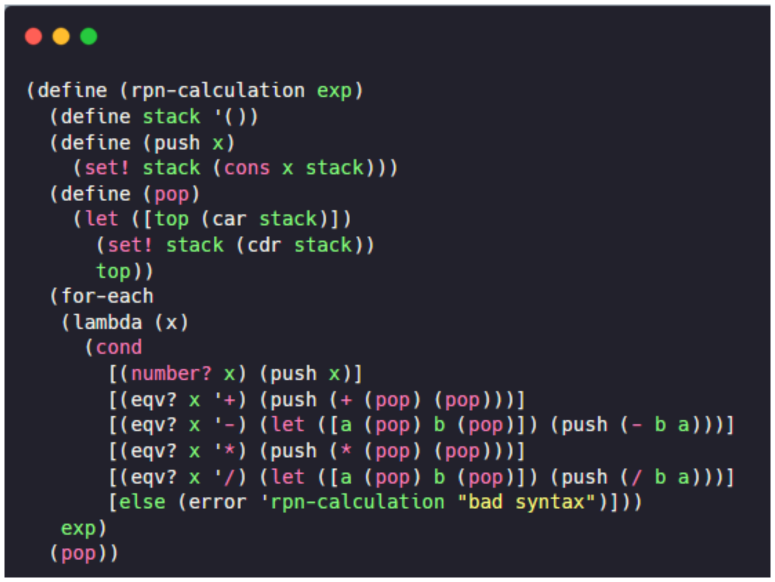


```
(define (binary-search lst value)
  (define (helper lst value start end)
    (if (> start end)
        #f
        (let ([mid (quotient (+ start end) 2)])
          (cond [(< (list-ref lst mid) value)
                 (helper lst value (+ mid 1) end)]
                [(> (list-ref lst mid) value)
                 (helper lst value start (- mid 1))]
                [else mid]))))

  (helper lst value 0 (- (length lst) 1)))
```

Problem 5: RPN Calculator

RPN stands for Reverse Polish Notation. In computer science, it is a mathematical notation in which every operator follows all of its operands.

Here's an implementation of an RPN calculator in Racket:



```
(define (rpn-calculation exp)
  (define stack '())
  (define (push x)
    (set! stack (cons x stack)))
  (define (pop)
    (let ([top (car stack)])
      (set! stack (cdr stack))
      top))
  (for-each
   (lambda (x)
     (cond
       [(number? x) (push x)]
       [(eqv? x '+) (push (+ (pop) (pop)))]
       [(eqv? x '-) (let ([a (pop) b (pop)]) (push (- b a)))]
       [(eqv? x '*) (push (* (pop) (pop)))]
       [(eqv? x '/) (let ([a (pop) b (pop)]) (push (/ b a)))]
       [else (error 'rpn-calculation "bad syntax")]))
   exp)
  (pop))
```

The End.

Special Thanks to Dr. Shoja for all his efforts.