Problems

1. Write list-copy, which takes a list and returns an identical new list (ie do not just return the original list, cons up a new list).

   (list-copy (list 1 2 3))
   ;Value: (1 2 3)

2. Write n-copies, which takes a value and a number of copies, and returns a list with the appropriate number of copies.

   (n-copies 7 5)
   ;Value: (7 7 7 7)
   (n-copies "yay" 1)
   ;Value: ("yay")
   (n-copies 7 0)
   ;Value: () ; or #f
   (n-copies (list 3) 3)
   ;Value: ((3) (3) (3))

3. Write reverse, which takes a list and returns new list with the order of the elements reversed.

   (reverse (list 1 2 3))
   ;Value: (3 2 1)
   (reverse (list 1))
   ;Value: (1)
4. Write `append`, which takes two lists and returns a new list with the elements of the first list and the second list.

   \[(append (list 3 4) (list 1 2))\]
   \[;Value: (3 4 1 2)\]
   \[(append nil (list 1 2))\]
   \[;Value: (1 2)\]

5. Write `list-ref`, which takes a list and an index (starting at 0), and returns the nth element of the list. You may assume that the index is less than the length of the list.

   \[(list-ref (list 17 42 35 "hike") 0)\]
   \[;Value: 17\]
   \[(list-ref (list 17 42 35 "hike") 1)\]
   \[;Value: 35\]
   \[(list-ref (list 17 42 35 "hike") 2)\]
   \[;Value: 35\]

6. Write `list-range`, which takes two numbers \((a, b : a \leq b)\) and returns a list containing the numbers from \(a\) to \(b\), inclusive.

   \[(list-range 1 5)\]
   \[;Value: (1 2 3 4 5)\]
   \[(list-range 2 5)\]
   \[;Value: (2 3 4 5)\]
   \[(list-range 42 42)\]
   \[;Value: (42)\]
   \[(list-range 207 5)\]
   \[;Value: ()\]
7. Write `max-list`, which takes in a list of numbers and returns the maximum element. You may assume that the list is non-empty. (Hint: different base case than normal!)

```
(max-list (list 1))
;Value: 1
(max-list (list 1 3 5))
;Value: 5
(max-list (list 2 56 8 43 21))
;Value: 56
```

---

**Data Abstraction**

1. Derived Type - A user-designated and implemented type.

2. Constructor - Builds entity of the type

3. Selector - Returns one of the values of the type

4. Contract - Specifies the relationship between the constructor(s) and the selector(s).

```
(define (make-point x y)

define (get-x point)

define (get-y point)
```

8. Write `add-points` which takes two points and returns a new point which is the sum of the x and y coordinates.

```
(define result (add-points (make-point 3 4) (make-point 1 2)))
(get-x result)
;Value: 4
(get-y result)
;Value: 6
```
9. Write `left-of?` which takes two points and returns true if the first point is to the left of the second point.

```scheme
(left-of? (make-point 3 4) (make-point 1 2))
;Value: #f
(left-of? (make-point -3 4 (make-point 1 2)))
;Value: #t
```

Stacking Abstractions: Segments

10. Implement an abstraction for line-segments, which are defined by a pair of end-points.

11. Write `segment-length`, which takes a segment and returns it’s length.