PROCEDURES ARE FIRST-CLASS OBJECTS IN SCHEME. THEY MAY BE PASSED IN AS PARAMETERS, STORED IN VARIABLES, AND RETURNED FROM FUNCTIONS.

Write Scheme expressions with the following names and behaviors:

1. **divide-by**: Given a number, return a procedure that accepts a number and divides it by this first number

2. **square-and-add**: Given a number, return a procedure that accepts a number, squares it, and adds the first number

3. **compose**: Given two procedures, return a procedure which, given an input, applies the second function then the first. Thus \((\text{compose } f \ g) \ 5\) is equivalent to \(f \ (g \ 5)\)

Then make sure you can evaluate this expression:

\((\text{compose } (\text{square-and-add } 42) \ (\text{divide-by } 2)) \ 20\). What do you get?

**consider this**

1. Draw box-and-pointer for the values of the following expressions. Also give the printed representation.
(cons 1 2)
(cons 1 (cons 3 (cons 5 nil)))
(cons (cons (cons 3 2) (cons 1 0)) nil)
(cons 0 (list 1 2))
(list (cons 1 2) (list 4 5) 3)

2. Write expressions whose values will print out like the following.

(1 2 3)
(1 2 . 3)
(((1 2) (3 4) (5 6))

3. Write expressions using car and cdr that will return 4 when the name lst is bound to the following values:

(7 6 5 4 3 2 1)
(((7) (6 5 4) (3 2) 1)
(7 (6 (5 (4 (3 (2 (1))))))
(7 ((6 5 ((4)) 3) 2) 1)

Down for the Count

Write a procedure, list-ref, with type List<A>, non-negative integer -> A, which will return the Nth element of a list. Start counting from 0 like any good computer scientist.

(define list-ref
  (lambda (L n)
    ...

Copy cat

Give a list L, write a procedure copy which produces a new list with fresh new cons cells but contains the same elements. Then, evaluate:

(define L1 (list 1 5 (list 8 9) 'foo (quote bar)))
(eq? L1 (copy L1))
(eq? (copy L1) (copy L1))
(equal? L1 (copy L1))
Got it backwards

Write a procedure reverse which, given a list L, returns a new list where the elements appear in the reverse order. Thus:

(reverse '(1 2 3 4 5)) => (5 4 3 2 1)
(reverse (list (list 1 2) (list 3 4) 5)) => (5 (3 4) (1 2))

A special snowflake

Create a procedure, unique, which given a list returns a new list where each element appears only once:

(unique '(1 2 2 3 4 5 4)) => (1 2 3 4 5 8)
Getting things all set

Suppose you and Ben Bitdiddle are working for the registrar, who has asked you to develop a Scheme system to keep track of each student’s schedule. Each class has a name, start time, and end time. For flexibility, Ben decides to model this as a number of labeled time ranges (where time is just a number):

(define (make-range min max label)
  (list 'range min max label))
(define range-min second)
(define range-max third)
(define range-label fourth)

Getting everything arranged

1. Add a range? predicate to help determine if something is a range.
   (define (range? thing)

2. Write within-range?, which takes a point in time and determines if it is within the specified range. Treat the endpoints of the range as being inside of it.
   (define (within-range? x range)

3. We also need to be able to group together these time ranges into a schedule. Come up with a set abstraction which groups together multiple ranges.
   (define (make-set)

   (define (set? thing)

   (define (add-range-to-set r set)

   (define (set-ranges set)

4. Write within?, which takes a point in time and either a set or a range; if it is a range, use within-range? to check the bounds. If it is a set, return #t if it is within any of the ranges.
   (define (within? x thing)

5. Write labels-at, which takes a point in time and a set and returns a list of the label of every range within the set that overlaps with that point. Use map and filter.
   (define (labels-at x set)

Bonus

What does the following expression evaluate to?

( (lambda (x) (x x)) (lambda (x) (x x)) )