and and or are not procedures, but special forms. They have a common feature of “short-circuiting” at a value if their task is completed early, and then they do not need to evaluate all of their arguments.

**Scheme**

1. Special Forms

   (a) **and** – (and arg1 arg2 …) Evaluates arguments from left to right, stopping at the first one that evaluates to false and returning false. Should all arguments evaluate “true-ishly”, returns the values of the last argument.

   Example:

   ```scheme
   (and (not null? lst)
       (= (car lst) 3))
   ```

   If the list lst is not null, then the procedure continues to check if the first element of the list is a three. If it is, the return is true; if not the return is false. However, if the list lst is null, the entire thing has a return of false even though the second argument was not evaluated.

   (b) **or** – (or arg1 arg2 …) Evaluates arguments from left to right, stopping at the first one that evaluates to “true-ish” and returns that value. Should all the arguments evaluate to false, returns false.

   Example:

   ```scheme
   (or (= (car lst) 3)
       (check-3 (cdr lst)))
   ```

   If the first element of the list is three, procedure returns true. If not, it continues to the next, and performs the same check. If true, it returns true; if false, it returns false.

As an aside and reminder, a procedure takes in inputs and returns outputs.

**Higher Order Procedures**

The idea of programming is to capture patterns, as in the summation of n, and n – 1, or the squares of n, and n-1, and use those patterns to form procedures for evaluation.

(The following are de-sugared to show the multiple lambdas.)
(define sum
  (lambda (f x y dx)
    (if (> x y)
        0
        (+ (f x)
            (sum f (+ x dx) y dx))))  
← the argument f in this case represents a function,
which we know because it is listed in the position of
an operation

(define make-adder
  (lambda (amt)
    (lambda (x) (+ x amt)))))

(define add-3 (make-adder 3))

(add-3 5)
;Value: 8

(((make-adder 3) 5)

((((lambda (amt) (lambda (x) (+ x amt))) 3) 5) 5)
((lambda (x) (+ x 3)) (+ 3 5)) → 8

We want to compose two functions, f and g.

(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

f represents the square function
(define square (lambda (x) (* x x)))
g represents the increment function
(define inc (lambda (x) (+ x 1)))

Type Analysis

(define inc-square (compose square inc))
(inc-square 3) → 16

The compose function takes in two procedures and returns a procedure. These are the
“types” involved. For this class, it was notated the following way:

num: number
\rightarrow: \text{procedure (which takes in whatever is to the left of the arrow, and returns whatever is to the right)} \\
\text{Bool: Boolean}

For the compose function:

\text{(num} \rightarrow \text{num)}, \text{(num} \rightarrow \text{num}) \rightarrow \text{(num} \rightarrow \text{num)}

This means you put two procedures into the compose function (the two procedures written to the left of the procedure arrow, separated by commas), each of which takes in a number and returns a number. The return value is a procedure which also takes in a number and returns a number.

\text{inc-square:}
\text{(compose square inc)}

\((\lambda f \ g \ (\lambda x \ (f \ (g \ x)))) \ (\lambda x \ (* \ x \ x)) \ (\lambda x \ (+ \ x \ 1))\) \((\lambda y \ ((\lambda x \ (* \ x \ x)) \ ((\lambda z \ (+ \ z \ 1) \ y))) \ 3)\)

(The variables have been renamed to y and z, so that the difference between them and x is more apparent.)

\(\((\lambda x \ (* \ x \ x)) \ (\lambda z \ (+ \ z \ 1)) \ 3)\)
\(* \ 4 \ 4) \Rightarrow 16 \quad (+ \ 3 \ 1) \Rightarrow 4\)

Reiteration from previous notes:
Types are a powerful tool for analyzing code
- you can analyze code and see why a program isn’t working
- you can use types to help you fill-in-the-blank of what belongs in a code

The following are types of values returned from the listed expressions:

4 \quad \text{returns a number}
(+ 1 1) \quad \text{returns a number}
(\lambda x (+ x 1)) \quad \text{returns a procedure (num} \rightarrow \text{num)}
(\lambda x (= x 1)) \quad \text{returns a procedure (num} \rightarrow \text{boolean)}

Above, we can tell x’s type is number because a number is the only type that will fit with an integer-function (add, equals, etc.).

\text{(lambda} \ x \ y\text{)}
\text{(if y}
\quad (+ x 3)
\quad 7))\)

The above returns a procedure that takes in a number and a Boolean and returns a number. (num, bool} \rightarrow \text{num)