Evaluation Strategy
So far in this class, we have used **call-by-value** evaluation: function arguments are evaluated to values before being passed into the function body.
Another strategy is to pass in the uninterpreted arguments to the function, and make the function itself handle their evaluation. This is known as call-by-name evaluation.
What is evaluation?

(first (map (lambda (x) (factorial x)) lst) (list 1 2 3))
(first (map (lambda (x) (factorial x)) (list 1 2 3)))
(first (list (factorial 1) (factorial 2) (factorial 3)))
(first (list (factorial 1) (* 2 (factorial 1))(* 3 (factorial 2)))))
(first (list 1 (* 2 1)(* 3 (* 2 (factorial 1)))))
(first (list 1 2 (* 3 (* 2 1))))
(first (list 1 2 6))
1
Throw-back: order of operations

In elementary school, you might have learned a rule about the order of operations for arithmetic:

Please Excuse My Dear Aunt Sally
(parentheses, exponents, multiplication, division, addition, subtraction)

Parentheses specify scope, but the others specify evaluation order.
Parentheses specify scope, but the others specify evaluation order: first evaluate the exponentiation, then the multiplication, then the division...

The evaluation strategy of a programming language tells you what things get done first.
What really happens here?

(define (factorial n)
    (letrec ((helper (lambda (x res)
                       (if (= x n)
                           res
                           (helper (+ 1 x) (* x res))))))
        (helper 1 1)))

(+ (square (* (factorial (+ 1 2)) 5)) 10)
What really happens here?

One option: work from the outside inwards

> (+ (square(* (factorial (+ 1 2)) 5)) 10)
  ((square (* (factorial (+ 1 2)) 5)) + 10)
  ((factorial (+ 1 2)) * 5) * ((factorial (+ 1 2)) 5)) + 10)
  (((factorial (+ 1 2)) * 5) * ((factorial (+ 1 2)) 5)) + 10)
  (((1 (* 2 (+ 1 2))) * 5) * ((1 (* 2 (+ 1 2)) 5)) + 10)
  (((1 (* 2 3)) * 5) * ((1 (* 2 3)) 5)) + 10)
  (((2 3) 1) * 5) * (((2 3) 1) 5)) + 10)
  (((2 3) 1) * 5) * (((2 3) 1) 5)) + 10)
  (((6 1) 5) * ((6 1) 5)) + 10)
  ((6 5) * (6 5)) + 10)
  (30 * 30) + 10)
  (900 + 10)
  910
What really happens here?

Another option: work from the inside outwards

> (+ (square(* (factorial (+ 1 2)) 5)) 10)
   (+ (square (* (factorial (+ 1 2)) 5)) 10)
   (+ (square (* (factorial 3) 5)) 10)
   (+ (square (* (* 1 (* 2 3)) 5)) 10)
   (+ (square (* (* 1 6) 5)) 10)
   (+ (square (* 6 5)) 10)
   (+ (square 30) 10)
   (+ 900 10)
   910
Eager Evaluation

Evaluate expressions as soon as possible
Eager Evaluation

(first (map (lambda (x) (factorial x)) lst) (list 1 2 3))

(first (map (lambda (x) (factorial x)) (list 1 2 3)))

(first (list (factorial 1) (factorial 2) (factorial 3)))

(first (list (factorial 1) (* 2 (factorial 1))(* 3 (factorial 2)))))

(first (list 1 (* 2 1)(* 3 (* 2 1))))

(first (list 1 2 (* 3 (* 2 1))))

(first (list 1 2 6))

1
Lazy evaluation

Evaluate expressions only when needed
Lazy Evaluation

(first (map (lambda (x) (factorial x)) lst) (list 1 2 3))
(first (map (lambda (x) (factorial x)) (list 1 2 3)))
(first (list (factorial 1) (factorial 2) (factorial 3)))
(first (list (factorial 1) (* 2 (factorial 1))(* 3 (factorial 2)))))
(first (list 1 (* 2 1)(* 3 (* 2 (factorial 1)))))
(first (list 1 2 (* 3 (* 2 1))))
(first (list 1 2 6))

1
(first (map (lambda (x) (factorial x)) lst) (list 1 2 3))
(first (map (lambda (x) (factorial x)) (list 1 2 3)))
(first (list (factorial 1) (factorial 2) (factorial 3)))
(factorial 1)
1
Call-by-Need

Wait to evaluate an expression until it is needed, but once it is evaluated, remember its value.
Exercise: endless string list

Exercise: write a function that takes a single string as an argument and creates an endless list of that string.

Call it *endless-strings*.
Evaluation Strategies

- **Eager**
  - Call-by-value (Racket, Java*, C)

- **Lazy**
  - Call-by-need (Haskell, R)
  - Call-by-name (Algol)

*Java objects are complicated*