Goals of this tutorial
You should be able to...

• write mutually recursive functions based on the relevant templates.
• work with lists of arbitrary nesting.
• understand the syntax and semantics of local.
• use local when writing your own functions.

Group Problem: Mutual Recursion
For this problem, we will use the following structure and data definition to represent a general tree of people:

(define-struct person (name children))

;; A Person is a (make-person Str (listof Person))
;; requires: the names are unique

According to this data definition, we cannot have an empty tree of people.
Group Problem: Mutual Recursion

Here is what cs135-staff looks like:

```
(define cs135-staff
(make-person "Karen"
(list (make-person "Paul"
 (list (make-person "Troy"
 (list (make-person "Jean" empty)
 (make-person "Sana" empty)))))
(make-person "Byron"
 (list (make-person "Dustin" empty)))
(make-person "Craig" empty)
(make-person "Sandy"
 (list (make-person "Vincent" empty)
 (make-person "Jimmy"
 (list (make-person "Ben" empty)))
 (make-person "Zainab" empty)))))
```

Group Problem: Mutual Recursion

Write templates for each of Person and (listof Person).

```
(define-struct person (name children))
```

;; A Person is a (make-person Str (listof Person))
;; requires: the names are unique
Group Problem: add-person

Using your templates, write a function add-person that consumes two strings, parent-name and new-name, and a Person. It produces the result of inserting a new person with the name new-name as an immediate child of the person with the name parent-name. The new person is inserted at the beginning of the corresponding list of children.

You may assume that new-name does not exist as a name in the consumed tree. If parent-name does not exist, the function should produce the same tree that was originally consumed.

(add-person "Paul" "Ian" cs135-staff) ⇒

(add-person "Vincent" "Mark" cs135-staff) ⇒
Group Problem: add-person

(add-person "Dave" "Josh" cs135-staff) ⇒ cs135-staff

Consider the following data definition for a nested list of integers:

;; A Nested-List-Int is one of:
;;   * empty
;;   * (cons Int Nested-List-Int)
;;   * (cons Nested-List-Int Nested-List-Int)

Write a function remove-odds which consumes a Nested-List-Int and removes all odd integers from the consumed list. However, all empty lists and any nested lists that only contain empty lists after removing the odd integers should be removed as well.

(remove-odds '(1 (2 (3 (5) 7) 6) (8 (4 3)))) ⇒ '((2 6) (8 (4)))

Review: Local Definitions

Recall the special form local which allows us to create local definitions. The syntax for local is as follows:

(local [definition_1 ... definition_n]
  expression)

where each definition is a define statement, and expression is a Racket expression that uses these definitions.
Clicker Question: Local Definitions
In Intermediate Student, what would this expression produce?
(define a 10)
(define b
  (local [(define a 5)]
    (add1 a)))
(+ a b)
A 10
B 15
C 16
D 21
E An error

Clicker Question: Local Definitions
In Intermediate Student, what would the following expression produce?
(define (sum-lon alon)
  (local [(define (sum-lon/acc alon sum-so-far)
           (cond [(empty? alon) sum-so-far]
                  [else (sum-lon/acc (rest alon) (+ (first alon) sum-so-far))])]
        (sum-lon/acc alon 0)))
(sum-lon/acc (list 2 3 5 7) 0)
A 0
B 2
C 7
D 17
E An error

Stepping Problem - Local
Provide a step-by-step evaluation of the following program. When renaming
local definitions, append ",0" if possible, or else ",1", ",2", etc. Do not recopy
any line that is already in its simplest form.
(define (f x y)
  (local
    [(define a (+ x 3))
     (define y 4)
     (define (g x)
       (+ x a))
     (+ 2 (g y))]
  (f 2 6))
(define (f x y)
  (local
    [(define a (+ x 3))
     (define y 4)
     (define (g x)
       (+ x a))]
    (∗ 2 (g y))))
(f 2 6)
(define a 0 5)
(define y 0 4)
(define (g 0 x)
  (+ x a 0))
⇒ (+ 2 (g 0 y 0))
⇒ (+ 2 (g 0 4))
⇒ (+ 2 (+ 4 a 0))
⇒ (+ 2 (+ 4 5))
(define a 0 5)
(define y 0 4)
(define (g 0 x)
  (+ x a 0))
⇒ (∗ 2 (g 0 y 0))
⇒ (∗ 2 (g 0 4))
⇒ (∗ 2 (+ 4 a 0))
⇒ (∗ 2 (+ 4 5))
⇒ (∗ 2 9)

Group Discussion: Using local
Let’s revisit the solution for the remove-odds function. How could you use local constant definitions to avoid exponential blowups and improve readability, without using another helper function?

(define (remove-odds nest-loints)
  (cond
   [(empty? nest-loints) empty]
   [(and (integer? (first nest-loints))
         (odd? (first nest-loints)))
     (remove-odds (rest nest-loints))]
   [(integer? (first nest-loints))
     (cons (first nest-loints) (remove-odds (rest nest-loints)))]
   [(empty? (remove-odds (first nest-loints)))
     (remove-odds (rest nest-loints))]
   [else (cons (remove-odds (first nest-loints))
             (remove-odds (rest nest-loints)))]))