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

- understand the syntax and semantics of `local`.
- step through `local` definitions.
- use `local` when writing your own functions.
- step through code involving `lambda`.
- write functions that consume and/or produce other functions.
- write functions using `lambda`.
- use `filter` and `map`.

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 the following expression produce?

```scheme
(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 1 9 1 23) 0))
```

A 1
B 10
C 11
D 34
E An error

CS 135 Winter 2019 Tutorial 9: Local and Lambda

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.

```scheme
(define (f x y)
  (local
    [(define a (+ x 3))
     (define y 4)
     (define (g x)
      (+ x a))]
    (* 2 (g y))))
(f 7 3)
```

CS 135 Winter 2019 Tutorial 9: Local and Lambda
(define a (+ 7 3))
(define y 4)
(define (g x)
  (+ x a))
(* 2 (g y)))

(define a 0 (+ 7 3))
(define y 0 4)
(define (g 0 x)
  (+ x a 0))
(* 2 (g 0 y 0))

(define a 0 10)
(define y 0 4)
(define (g 0 x)
  (+ x a 0))
(* 2 (g 0 y 0))
⇒ (* 2 (g 0 y 0))
⇒ (* 2 (g 0 4))
(define a₀ 10)
(define y₀ 4)
(define (g₀ x)
  (+ x a₀))
⇒ (* 2 (g₀ y₀))
⇒ (* 2 (g₀ 4))
⇒ (* 2 (+ 4 a₀))
⇒ (* 2 (+ 4 10))
⇒ (* 2 14)
(define a 0 10)
(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 10))
⇒ (+ 2 14)
⇒ 28

Clicker Question - Contracts with Function Types
What would be the contract for this function?
(define (f a b)
  (local [(define (f c) (+ (* a c) (* b c)))]
    f))
A ;; f: Num Num → Num
B ;; f: Num Num → Num
C ;; f: Num Num → (Num → Num)
D ;; f: Num Num → Function
E ;; f: Num Num → (fn Num → Num)

Review: Local Definitions
Here are two reasons why we might want to use local expressions:

1. Encapsulation - local allows us to hide parts of our program from each other, since anything defined inside a local expression is not visible from outside the local. For example, we can define a helper function inside the function it is helping, and no function defined outside of that main function will be able to use that helper.

2. Efficiency - We can use local to eliminate repeated computations by storing the result of a computation in a local variable, and using that variable whenever the value is needed. This prevents us from having to repeatedly recomputed the value.
Group Discussion - Efficiency with Local
Given the code below, how could you use local constant definitions to make it more efficient?

```scheme
;; list-min: (listof Num) → Num
;; requires: lon is non-empty
(define (list-min lon)
  (cond
   [(empty? (rest lon)) (first lon)]
   [(<= (first lon) (list-min (rest lon))) (first lon)]
   [else (list-min (rest lon))]))
```

Review: Lambda
(local [(define (name-used-once x1 . . . xn) exp)]
  name-used-once)
can also be written
(lambda (x1 . . . xn) exp)

lambda can be thought of as “make-function”.
It can be used to create a function which we can then use as a value – for example, as the value of the first argument to filter, map, or foldr.

Group Problem - Stepping with Lambda
Step through the following program:
```
((lambda (x y z) (* x 5)) 9 (+ 1 2) 8)
```
Group Problem - above-average

```lisp
(define-struct student (name grade))

;; A Student is a (make-student Sym Num)
```

write a function (twice, once using lambda and once using local) `above-average` which consumes a `Num` representing an average mark in some course. `above-average` will produce a function that consumes a `Student` and determines if that student has a grade that is higher than the average mark. Include a contract.

```lisp
;; Example
(define above-cs-mt-avg? (above-average 67.8))
(check-expect (above-cs-mt-avg? (make-student 'Turing 73)) true)
```

Abstract List Functions
Recall the abstract list functions `filter`, `map`, `foldr`, and `build-list`.

```lisp
;;filter: (X → Bool) (listof X) → (listof X)
;;map: (X → Y) (listof X) → (listof Y)
;;foldr: (X Y → Y) Y (listof X) → Y
;;build-list: Nat (Nat → X) → (listof X)
```

Use abstract list functions to do the following tasks without explicit recursion:

- Add the list of numbers in `(list 1 2 3 4 5)`: ⇒ 15
- Create a list of odd numbers from 1 to 12: ⇒ `(list 1 3 5 7 9 11)`
- Double each element in `(list 1 2 3 4 5)`: ⇒ `(list 2 4 6 8 10)`
- Keep all the elements in `(list 1 2 3 4 5 6 7)` that are divisible by 3: ⇒ `(list 3 6)`

Group Problem - invite-list

```lisp
(define-struct student (name grade))

;; A Student is a (make-student Sym Num)
;; requires: 0 <= grade <= 100
```

The Engineering Faculty of Waterloo regularly holds a faculty dinner to which any student who holds a cumulative average of over 95% is invited. Write a function `invite-list` which consumes a list of `Student`s and produces a list of `Students` who will be invited to the dinner. Include a contract for each function you write. You must use the built-in function `filter`. All helper functions must be encapsulated in a `local` block with purposes and contracts.
Group Problem - map-lofn

Write a function map-lofn which consumes an argument of type Any and a list of functions and produces a list of Any by applying each function from the list to the first argument.

;; (map-lofn item lofn) consumes an item of any type and a list of functions lofn, and produces a list containing the output of each function in lofn applied to item
;; map-lofn: X (listof (X → Any)) → (listof Any)