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 can be either a define or define-struct statement, and expression is a Racket expression that uses these definitions.

Clicker Question - Local Definitions
In Intermediate Student, what would this code produce?

```
(define a 10)
(define b
  (local [(define a 5)]
    (add1 a)))
```

( + a b)
A 11
B 15
C 16
D 21
E An error
Clicker Question - Local Definitions
In Intermediate Student, what would this code produce?

```scheme
(define (fn alon)
  (local [(define (fn-helper alon n)
            (cond [(empty? alon) n]
                  [else (fn-helper (rest alon) (+ (first alon) n))]))
         (fn-helper alon 0)))(fn-helper (list 2 3 5 7) 0)
```

A 0  
B 2  
C 7  
D 17 
E An error

Clicker Question - Local Definitions
In Intermediate Student, what would this function produce?

```scheme
(define (f a b)
  (local [(define (f c) (+ (* a c) (* b c)))] f))
```

A A number  
B A function that doesn't consume anything  
C A function that consumes one number  
D A function that consumes two numbers  
E An error

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 recompute 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))]])
```

Group Problem - Stepping with Local

In a text editor or on paper, do a full trace of the following code. When renaming local definitions, append ".0" if possible, or else ".1", ".2", etc. Do not recopy any line that is already in simplest form.

```scheme
(define (foo x)
  (local [(define a 1) (define b 2)] (+ x a b)))
(+ (foo 3) 5)
```

Group Problem - above-average

```scheme
(define-struct student (name grade))
;; A Student is a (make-student Sym Num)

Write a function 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 purpose and contract for your function.

;; Example
(define above-cs-mt-avg? (above-average 68.5))
(check-expect (above-cs-mt-avg? (make-student 'Turing 73)) true)
```