The functions and special forms we’ve seen so far can be arbitrarily nested—except `define` and `check-expect`.

So far, definitions have to be made “at the top level”, outside any expression.

The Intermediate language provides the special form `local`, which contains a series of local definitions plus an expression using them.

```
(local [(define x_1 exp_1) ... (define x_n exp_n)] bodyexp)
```

What use is this?
Consider Heron’s formula for the area of a triangle with sides $a$, $b$, $c$:

$$\sqrt{s(s - a)(s - b)(s - c)}$$, where $s = (a + b + c)/2$

It is not hard to create a Racket function to compute this function, but it is difficult to do so in a clear and natural fashion.

We will describe several possibilities, starting with a direct implementation.
Motivation: direct translation

\[
\text{(define (t-area-v0 a b c)}
\text{(sqrt)}
\text{(* (/ (+ a b c) 2) )}
\text{(- (/ (+ a b c) 2) a)}
\text{(- (/ (+ a b c) 2) b)}
\text{(- (/ (+ a b c) 2) c))})
\]

The repeated computation of \( s = (a + b + c)/2 \) is awkward.
We could notice that \( s - a = (-a + b + c)/2 \), and make similar substitutions.

\[
\text{(define } (t-area-v1 a b c) \\
\text{(sqrt} \\
\text{(* (/ (+ a b c) 2) \\
\text{(/ (+ (- a) b c) 2) \\
\text{(/ (+ a (- b) c) 2) \\
\text{(/ (+ a b (- c)) 2)))))}
\]

This is slightly shorter, but its relationship to Heron’s formula is unclear from just reading the code, and the technique does not generalize.
We could instead use a helper function.

\[
\text{(define } (t\text{-area-v2 } a \ b \ c) \\
\text{ (sqrt} \\
\text{(\*) (s } a \ b \ c) \\
\text{ (- (s } a \ b \ c) a) \\
\text{ (- (s } a \ b \ c) b) \\
\text{ (- (s } a \ b \ c) c))})
\]

\[
\text{(define } (s } a \ b \ c) \\
\text{(/) (+ } a \ b \ c) 2))
\]

This generalizes well to formulas that define several intermediate quantities.

But the helper functions need parameters, which again makes the relationship to Heron’s formula hard to see. And there’s still repeated code and repeated computations.
We could instead move the computation using $s$ into a helper function, and provide the value of $s$ as a parameter.

```scheme
(define (t-area-v3 a b c)
  (t-area/s a b c (/ (+ a b c) 2)))

(define (t-area/s a b c s)
  (sqrt (* s (- s a) (- s b) (- s c))))
```

This is more readable, and shorter, but it is still awkward because the value of $s$ is defined in one function and used in another.
Motivation: use `local`

The `local` special form we introduced provides a natural way to bring the definition and use together.

```scheme
(define (t-area-v4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

This is nice and short!

It looks like Heron’s formula.

No repeated code or computations.

Since `local` is another special form (like `cond`) that results in double parentheses, we will use square brackets to improve readability. This is another convention.
Local definitions permit reuse of names.

This is not new to us:

\[
\begin{align*}
& \text{(define } n 10) \\
& \text{(define (myfn } n) (+ 2 n)) \\
& \text{(myfn 6)}
\end{align*}
\]

gives the answer 8, not 12.

The substitution specified in the semantics of function application ensures that the correct value is used while evaluating the last line.
(define n v) binds a value v to a name n.

The name of a formal parameter to a function may reuse, within the body of that function, a name such as n.

Similarly, a define within a local expression may reuse a name which has already been bound to another value or expression.

The substitution rules we define for local as part of the semantic model must handle this. The resulting substitution rule for local is the most complicated one we will see in this course.
The substitution rule works by replacing every name defined in the `local` with a **fresh name** (a.k.a. **fresh identifier**) – a new, unique name that has not been used anywhere else in the program.

Each old name within the `local` is replaced by the corresponding new name.

Because the new name hasn’t been used elsewhere in the program, the local definitions (with the new name) can now be “promoted” to the top level of the program without affecting anything outside of the `local`.

We can now use our existing rules to evaluate the program.

The next slide shows an example (the last version of Heron’s formula). Then we’ll state the rule more rigorously.
> Example: evaluating t-area4

We’ll need a fresh identifier to replace \( s \). We’ll use \( s_1 \), which we just made up.

\[
(t\text{-}area4 \, 3 \, 4 \, 5) \Rightarrow
\]
\[
\begin{align*}
&\text{(local } [\text{(define } s \, (/ \, (+ \, 3 \, 4 \, 5) \, 2))] \\
&\quad (\text{sqrt } (* \, s \, (- \, s \, 3) \, (- \, s \, 4) \, (- \, s \, 5)))) \Rightarrow
\end{align*}
\]
\[
\begin{align*}
&(\text{define } s_1 \, (/ \, (+ \, 3 \, 4 \, 5) \, 2)) \\
&(\text{sqrt } (* \, s_1 \, (- \, s_1 \, 3) \, (- \, s_1 \, 4) \, (- \, s_1 \, 5)))) \Rightarrow
\end{align*}
\]
\[
\begin{align*}
&(\text{define } s_1 \, (/ \, 12 \, 2)) \\
&(\text{sqrt } (* \, s_1 \, (- \, s_1 \, 3) \, (- \, s_1 \, 4) \, (- \, s_1 \, 5)))) \Rightarrow
\end{align*}
\]
\[
\begin{align*}
&(\text{define } s_1 \, 6) \\
&(\text{sqrt } (* \, s_1 \, (- \, s_1 \, 3) \, (- \, s_1 \, 4) \, (- \, s_1 \, 5)))) \Rightarrow \ldots \, 6
\end{align*}
\]

Work through the trace in the commentary.
In general, an expression of the form

\[(\text{local } [(\text{define } x_1 \text{ exp}_1) \ldots (\text{define } x_n \text{ exp}_n)] \text{ bodyexp})\]

is handled as follows:

- \(x_i\) is replaced with a fresh identifier (call it \(x_i_{\text{new}}\)) everywhere in the \textit{local} expression, for \(1 \leq i \leq n\).
- The definitions \((\text{define } x_1_{\text{new}} \text{ exp}_1) \ldots (\text{define } x_n_{\text{new}} \text{ exp}_n)\) are then lifted out (all at once) to the top level of the program, preserving their ordering.
- What remains looks like \((\text{local } [] \text{ bodyexp}')\), where \text{bodyexp}' is the rewritten version of \text{bodyexp}. Replace the \textit{local} expression with \text{bodyexp}'.

All of this (the renaming, the lifting, and removing the \textit{local} with an empty definitions list) is a \textbf{single step}. 

> Substitution rule for \textit{local}
Our previous statement about using our existing rules isn’t quite correct. Consider the code on the right.

Where is 2 substituted for $x$?

$$(\text{define } (\text{foo } x \ y))$$

$$\quad (+ (\text{local } [(\text{define } x \ y)$$

$$\quad \quad (\text{define } z (+ x \ y))]$$

$$\quad \quad (+ x \ z))$$

$$\quad x))$$

$$(\text{foo } 2 \ 3)$$

$$(f \ v_1 \ldots v_n) \Rightarrow \exp' \text{ where } (\text{define } (f \ x_1 \ldots x_n) \exp) \text{ occurs to the left, and } \exp' \text{ is obtained by substituting into the expression } \exp, \text{ with all occurrences of the formal parameter } x_i \text{ replaced by the value } v_i \text{ (for } i \text{ from 1 to } n) \text{ except where } x_i \text{ has been redefined within } \exp \text{ (e.g. within a } \text{local}).$$
Write a function \( \text{check-msg-length to from body min-len max-len} \). \( \text{to, from and body} \) are of type \( \text{Str} \) and represent a message. The length of the message is the combined lengths of \( \text{to}, \text{from}, \text{and body} \).

\( \text{min-len} \) and \( \text{max-len} \) are \( \text{Nat} \) values representing the minimum and maximum message lengths allowed. The function produces 'too-short for messages shorter than \( \text{min-len} \), 'too-long for messages longer than \( \text{max-len} \), and otherwise the length of the message.

\[
\begin{align*}
\text{(check-expect (check-msg-length "Ed" "Santa" "Xmas List" 3 14) 'too-long)}
\text{(check-expect (check-msg-length "Ed" "Santa" "Xmas List" 3 140) 16)}
\text{(check-expect (check-msg-length "Charlie" "Santa" "No presents for Ed!" 140 280) 'too-short)}
\end{align*}
\]

First implement this function without \( \text{local} \). Then reimplement the function using \( \text{local} \) to avoid computing the message length multiple times.
Reasons to use local

- Clarity: Naming subexpressions
- Efficiency: Avoid recomputation
- Encapsulation: Hiding stuff
- Scope: Reusing parameters
A subexpression used twice within a function body always yields the same value.

Using `local` to give the reused subexpression a name improves the readability of the code.

This was a motivating factor in `t-area`. Naming the subexpression made the relationship to Heron’s Formula clear.

```scheme
(define (t-area-v4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c))))))
```
Sometimes we choose to use `local` in order to name subexpressions mnemonically to make the code more readable, even if they are not reused. This may make the code longer.

```
(define-struct coord (x y))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (coord-x p1) (coord-x p2)))
       (sqr (- (coord-y p1) (coord-y p2))))))

(define (distance p1 p2)
  (local [(define delta-x (- (coord-x p1) (coord-x p2)))
           (define delta-y (- (coord-y p1) (coord-y p2))]
       (sqrt (+ (sqr delta-x) (sqr delta-y))))))
```
Recall that in lecture module 09, we saw a version of max-list that used the same recursive application twice. The repeated computation of values caused it to be very slow, even for lists of length 25.

We can use local to avoid recomputation.
(max-list-v2 lon) produces the maximum element of lon

(check-expect (max-list-v2 (list 6 2 3 7 1)) 7)

(max-list-v2: (listof Number) -> Number

Requires: lon is nonempty

(define (max-list-v2 lon)
  (cond [[(empty? (rest lon)) (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else (max-list-v2 (rest lon))])))
;; (max-list-v4 lon) produces the maximum element of lon
(check-expect (max-list-v4 (list 1 3 2 5 4)) 5)

;; max-list-v4: (listof Num) -> Num
;; requires: lon is nonempty
(define (max-list-v4 lon)
  (cond [(empty? (rest lon)) (first lon)]
        [else
         (local [(define max-rest (max-list-v4 (rest lon)))]
          (cond [>(first lon) max-rest] (first lon]
               [else max-rest]))]))
;; search-bt-path-v1: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v1 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) empty]
    [(list? (search-bt-path-v1 k (node-left tree)))
     (cons 'left (search-bt-path-v1 k (node-left tree)))]
    [(list? (search-bt-path-v1 k (node-right tree)))
     (cons 'right (search-bt-path-v1 k (node-right tree)))]
    [else false])))
;; search-bt-path-v2: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v2 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) empty]
    [else (choose-path-v2 (search-bt-path-v2 k (node-left tree))
                             (search-bt-path-v2 k (node-right tree)))]
  )

(define (choose-path-v2 left-path right-path)
  (cond
    [(list? left-path) (cons 'left left-path)]
    [(list? right-path) (cons 'right right-path)]
    [else false]])
;; search-bt-path-v3: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v3 k bt)
  (cond
    [(empty? bt) false]
    [(= k (node-key bt)) empty]
    [else
      (local [(define left-path (search-bt-path-v3 k (node-left bt)))]
        (define right-path (search-bt-path-v3 k (node-right bt)))
        (cond [(list? left-path) (cons 'left left-path)]
              [(list? right-path) (cons 'right right-path)]
              [else false]))]))
This new version of `search-bt-path` avoids making the same recursive function application twice, and does not require a helper function.

But it still suffers from an inefficiency: we always explore the entire binary tree, even if the correct solution is found immediately in the left subtree.

We can avoid the extra search of the right subtree using nested `locals`.
Efficiency: search-bt-path: with nested local

;; search-bt-path-v4: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v4 k bt)
  (cond
    [(empty? bt) false]
    [(= k (node-key bt)) empty]
    [else
     (local [(define left-path (search-bt-path-v4 k (node-left bt)))]
       (cond [(list? left-path) (cons 'left left-path)]
             [else (local [(define right-path (search-bt-path-v4 k (node-right bt)))]
                        (cond [(list? right-path) (cons 'right right-path)]
                              [else false]))]))]))
Encapsulation is the process of grouping things together in a “capsule”.

We have already seen data encapsulation in the use of structures: we grouped several fields together into one “capsule”, the structure.

There is also an aspect of information hiding to encapsulation which we did not see with structures.

The local bindings are not visible (have no effect) outside the local expression. Thus, they can “hide” information from other parts of the program.

In CS 246 we will see how objects combine data encapsulation with another type of encapsulation we now discuss.
Local definitions can bind names to functions as well as values. Evaluating the local expression creates new, unique names for the functions just as for the values.

This is known as **behaviour encapsulation**.

Behaviour encapsulation allows us to move helper functions within the function that uses them, so they:

- are invisible outside the function.
- do not clutter the “namespace” at the top level.
- cannot be used by mistake.

This makes the organization of the program more obvious and is particularly useful when using accumulators.
Example: sum-list

```
(define (sum-list lon)
  (local [(define (sum-list/acc lst sofar)
        (cond [(empty? lst) sofar]
              [else (sum-list/acc (rest lst)
                                (+ (first lst) sofar)])]))
  (sum-list/acc lon 0)))
```

Advantages of making the accumulatively-recursive helper function local:

- It makes clear the helper has no use outside of sum-list.
- It facilitates reasoning about the program.
  
  HtDP (section VI) discusses reasoning with invariants. It will be discussed further in CS 245. It is important in CS 240 and CS 341.
Exercise

Write a function `(normalize lst)` that consumes a `(listof Num)`, and returns the list containing each value in `lst` divided by the sum of the values in `lst`.

Use only local helper functions, and compute the sum only once.

`(normalize (list 4 2 14)) ⇒ (list 0.2 0.1 0.7)`
Example: Insertion sort

```scheme
(define (isort lon)
  (local [(define (insert n slon)
            (cond [(empty? slon) (cons n empty)]
                  [(<= n (first slon)) (cons n slon)]
                  [else (cons (first slon) (insert n (rest slon))))]])
   (cond [(empty? lon) empty]
         [else (insert (first lon) (isort (rest lon)))]))
)```
A function can enclose the cooperating helper functions that it uses inside a local, as long as these are not also needed by other functions. When this happens, the enclosing function and all the helpers act as a cohesive unit.

Here, the local helper functions require contracts and purposes, but not examples or tests. The helper functions can be tested by writing suitable tests for the enclosing function.

Make sure the local helper functions are still tested completely!
(define (isort lon)
  (local [;; (insert n slon) inserts n into slon, preserving the order
          ;; insert: Num (listof Num) -> (listof Num)
          ;; requires: slon is sorted in nondecreasing order
          (define (insert n slon)
            (cond [((empty? slon)) (cons n empty)]
                  [((<= n (first slon))) (cons n slon)]
                  [else (cons (first slon) (insert n (rest slon))))]
          )
  )
  (cond [((empty? lon)) empty]
        [else (insert (first lon) (isort (rest lon))))])))
Local can also handle mutually recursive functions.

```scheme
(local
  [(define (foo z)
    (cond [(= z 0) 0]
          [else (+ 1 (bar (- z 1)))]))]

  (define (bar z)
    (cond [(= 0 z) 0]
          [else (+ 1 (foo (- z 1)))]))]

(foo 5))
```
Making helper functions local can reduce the need to have parameters “go along for the ride”.

;; (countup-to n) produces a list of the numbers from 0 to n
;; Example:
(check-expect (countup-to 5) (list 0 1 2 3 4 5))

;; countup-to: Nat -> (listof Nat)
(define (countup-to n)
  (countup-from-to 0 n))

;; countup-from-to: Nat Nat -> (listof Nat)
(define (countup-from-to from to)
  (cond [(> from to) empty]
        [else (cons from (countup-from-to (add1 from) to))]))
Example: countup-to

```
(define (countup-v2 n)
  (local [(define (countup-from from)
            (cond [(> from n) empty]
                  [else (cons from (countup-from (+ from 1))))])]
    (countup-from 0)))
```

`n` no longer needs to be a parameter to `countup-from`, because it is in scope.

If we evaluate `(countup-v2 10)` using our substitution model, a renamed version of `countup-from` with `n` replaced by 10 is lifted to the top level.

Then, if we evaluate `(countup-v2 20)`, another renamed version of `countup-from` is lifted to the top level.
Using only one helper function, which is local and has only one parameter, write a function \( \text{list-squares} \ n \) that produces a list containing the squares of the first \( n \) natural numbers.

(\text{check-expect} \ (\text{list-squares} \ 4) \ (\text{list} \ 0 \ 1 \ 4 \ 9))
Example: mult-table

We can use the same idea to localize the helper functions for `mult-table` from lecture module 08.

Recall that

\[
\text{(check-expect (mult-table 3 4) }
\]

\[
\text{(list (list 0 0 0 0) }
\]

\[
\text{(list 0 1 2 3) }
\]

\[
\text{(list 0 2 4 6))}
\]

The \(c^{th}\) entry of the \(r^{th}\) row (numbering from 0) is \(r \times c\).
;; mult-table: Nat Nat -> (listof (listof Nat))
(define (mult-table nr nc)
    (rows-to 0 nr nc))

;; (rows-to r nr nc) produces mult. table, rows r...(nr-1)
;; rows-to: Nat Nat Nat -> (listof (listof Nat))
(define (rows-to r nr nc)
    (cond [(>= r nr) empty]
          [else (cons (cols-to 0 r nc) (rows-to (add1 r) nr nc))])))

;; (cols-to c r nc) produces entries c...(nc-1) of rth row of mult. table
;; cols-to: Nat Nat Nat -> (listof Nat)
(define (cols-to c r nc)
    (cond [(>= c nc) empty]
          [else (cons (* r c) (cols-to (add1 c) r nc))])))
(define (mult-table2 nr nc)
  (local [;; (rows-to r) produces mult. table, rows r...(nr-1)
    ;; rows-to: Nat -> (listof (listof Nat))
    (define (rows-to r)
      (cond [(>= r nr) empty]
        [else (cons (cols-to 0 r) (rows-to (add1 r)))])))

  ;; (cols-to c r) produces entries c...(nc-1) of rth row
  ;; cols-to: Nat Nat -> (listof Nat)
  (define (cols-to c r)
    (cond [(>= c nc) empty]
      [else (cons (* r c) (cols-to (add1 c) r))])))
  (rows-to 0)))
The implementation of `mult-table2` encapsulates helper functions `cols-to` and `rows-to` using `local`. Modify `mult-table2` to further encapsulate `cols-to` into `rows-to`. What parameters of `cols-to` are no longer necessary?
Write a function `(table-ccr nr nc)` that produces a table containing `nr` rows and `nc` columns, where the $c^{th}$ entry of the $r^{th}$ row is $c^2 r$.

For example,

```scheme
(check-expect (table-ccr 4 5)
  (list (list 0 0 0 0 0)
        (list 0 1 4 9 16)
        (list 0 2 8 18 32)
        (list 0 3 12 27 48)))
```

Helper functions should be encapulated within a `local`, of course.
The **binding occurrence** of a name is its use in a definition, or formal parameter to a function.

The associated **bound occurrences** are the uses of that name that correspond to that binding.

The **lexical scope** of a binding occurrence is all places where that binding has effect, taking note of holes caused by reuse of names.

**Global scope** is the scope of top-level definitions.
The use of `local` has permitted modest gains in expressivity and readability in our examples.

The language features discussed in the next module expand this power considerably.

Some other languages (C, C++, Java) either disallow nested function definitions or allow them only in very restricted circumstances.

Local variable and constant definitions are much more common.
Goals of this module

- You should understand the syntax, informal semantics, and formal substitution semantics for the \texttt{local} special form.
- You should be able to use \texttt{local} to avoid repetition of common subexpressions, to improve readability of expressions, and to improve efficiency of code.
- You should understand the idea of encapsulation of local helper functions.
- You should be able to match the use of any constant or function name in a program to the binding to which it refers.
In this module we added the following to our toolbox:

```scheme
local
```

These are the functions and special forms currently in our toolbox: