13: Local Definitions

Local definitions

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?

Motivating local definitions

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

```
(define (t-area-v0 a b c)
  (sqrt
   (* (/ (+ a b c) 2)
      (- (/ (+ a b c) 2) a)
      (- (/ (+ a b c) 2) b)
      (- (/ (+ a b c) 2) c))))
```

The repeated computation of $s = (a + b + c)/2$ is awkward.

> Motivation: rewrite expressions

We could notice that $s - a = (-a + b + c)/2$, and make similar substitutions.

```
(define (t-area-v1 a b c)
  (sqrt
    (* (/ (+ a b c) 2)
       (/ (+ (- a) b c) 2)
       (/ (+ a (- b) c) 2)
       (/ (+ 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.

> Motivation: use a helper function (v1)

We could instead use a helper function.

```
(define (t-area-v2 a b c)
  (sqrt
   (* (s a b c)
      (- (s a b c) a)
      (- (s a b c) b)
      (- (s a b c) c))))

(define (s a b c)
  (/ (+ 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.

> Motivation: use a helper function (v2)

We could instead move the computation using *s* into a helper function, and provide the value of *s* as a parameter.

```
(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 (it looks like Heron's formula!), shorter, and avoids recomputation, but it is still awkward because

- the value of *s* is defined in one function and used in another.
- `t-area/s` has no apparent use other than to support `t-area-v3`.

> Motivation: use `local`

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

```
(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*.

Reusing names

Local definitions permit reuse of names. Reusing names is not new to us:

```
(define n 10)
(define (myfn n) (+ 2 n))
(myfn 6)
```

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.

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

The **local** substitution rules in our semantic model must handle this.

```
(define x 5)
(define (fun a)
  (local [(define x 3)]
         (+ a x)))
```

The resulting substitution rule for **local** is the most complicated one we will see in this course.

> Informal substitution rule for `local`

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.

We will state the rule rigourously a little later.

Example 1

```
(define x 5)
(define (fun a)
  (local [(define x 3)] (+ a x)))
(fun 4) ⇒

(local [(define x 3)] (+ 4 x)) ⇒

(define x_1 3)
(+ 4 x_1)  ⇒
(+ 4 3) ⇒
7
```

> Example2: evaluating t-area4

We'll need a fresh identifier to replace `s`. We'll use `s_1`, which we just made up.

```
(t-area4 3 4 5) ⇒
(local [(define s (/ (+ 3 4 5) 2))]
  (sqrt (* s (- s 3) (- s 4) (- s 5)))) ⇒
(define s_1 (/ (+ 3 4 5) 2))
(sqrt (* s_1 (- s_1 3) (- s_1 4) (- s_1 5))) ⇒
(define s_1 (/ 12 2))
(sqrt (* s_1 (- s_1 3) (- s_1 4) (- s_1 5))) ⇒
(define s_1 6)
(sqrt (* s_1 (- s_1 3) (- s_1 4) (- s_1 5))) ⇒ ... 6
```

Example 3:

```
(define (foo x)
   (local [(define a (+ x x))
           (define b (* x x))
           (define c (+ a b))]
     (+ a b c x)))
(foo 5) ⇒

(local [(define a (+ 5 5))
        (define b (* 5 5))
        (define c (+ a b))]
  (+ a b c 5))) ⇒

(define a_1 (+ 5 5))
(define b_1 (* 5 5))
(define c_1 (+ a_1 b_1))
(+ a_1 b_1 c_1 5) ⇒ ... ⇒ 75
```

Example 4: Revising function substitution

Our previous statement about using our existing rules isn't quite correct. Consider the code on the right.

Where is 2 substituted for x?

```
(define (foo x y)
  (+ (local [(define x y)
             (define z (+ x y))]
       (+ x z))
     x))


(foo 2 3)
```

(f v_1 ... v_n) ⇒ exp' where (define (f x_1 ... x_n) exp) occurs to the left, and exp' is obtained by substituting into the expression exp, with all occurrences of the formal parameter x_i replaced by the value v_i (for i from 1 to n) *except where x_i has been redefined within exp (e.g. within a **local**).*

Example 5

```
(define (foo x y)
  (+ (local [(define x y)
             (define z (+ x y))]
       (+ x z))
     (local [(define x (* 2 y))
             (define z (* x y))]
       (+ x z))))
(foo 2 3)  ⇒ ...

(define x_1 3)
(define z_1 6)
(+ (+ x_1 z_1)
   (local
     [(define x (* 2 3))
      (define z (* x 3))]
     (+ x z)))  ⇒ ...
```

```
(define x_1 3)
(define z_1 6)
(define x_2 (* 2 3))
(define z_2 (* x_2 3))
(+ 9 (+ x_2 z_2)) ⇒ ... ⇒ 33
```

Ex. 1

Write a function `(check-msg-length to from body min-len max-len)`. `to`, `from` and `body` are of type `Str` and represent a message. The length of the message is the combined lengths of `to`, `from`, and `body`.

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

```
(check-expect (check-msg-length "Ed" "Santa" "Xmas List" 3 14) 'too-long)
(check-expect (check-msg-length "Ed" "Santa" "Xmas List" 3 140) 16)
(check-expect (check-msg-length "Charlie" "Santa" "No presents for Ed!"
                                140 280) 'too-short)
```

First implement this function without **`local`**. Then reimplement the function using **`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

> Clarity: naming subexpressions

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.

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

> Clarity: meaningful names

Sometimes we choose to use **local** in order to give subexpressions meaningful names 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)))))
```

> Efficiency: avoid recomputation

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.

» Efficiency: `max-list` without local

```
;; (max-list-v2 lon) produces the maximum element of lon
;; Examples:
(check-expect (max-list-v2 (list 6 2 3 7 1)) 7)

;; max-list-v2: (listof Num) → Num
;; 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))]))
```

» Efficiency: max-list with local

```
;; (max-list-v4 lon) produces the maximum element of lon
;; Examples:
(check-expect (max-list-v4 (list 6 2 3 7 1)) 7)

;; 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]))]))
```

» Efficiency: `search-bt-path`: original

```
;; search-bt-path: Nat BT → (anyof false (listof (anyof 'right 'left)))
(define (search-bt-path k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) empty]
    [(list? (search-bt-path k (node-left tree)))
     (cons 'left (search-bt-path k (node-left tree)))]
    [(list? (search-bt-path k (node-right tree)))
     (cons 'right (search-bt-path k (node-right tree)))]
    [else false]))
```

The efficiency problems of this code can be solved with a helper function.

» Efficiency: `search-bt-path`: with **`local`**

```
;; 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]))]))
```

» Efficiency: `search-bt-path`: with **`local`**

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 **`local`**s.

» 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: hiding stuff

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 object-oriented programming combines data encapsulation (structures) with another type of encapsulation we now discuss.

» Behaviour encapsulation

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.
  In CS245 this reasoning will be extended to include **invariants**. They are also important in CS 240 and CS 341.

Ex. 2

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

```
(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)))])))
```

» Encapsulation and the design recipe

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!

» Design recipe example

```
;; Full Design Recipe for isort goes here...
(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)))])))
```

Mutual Recursion

Local can also handle mutually recursive functions.

```
;; my-even?: Nat -> Bool
(define (my-even? n)
  (local [(define (is-even? n)
            (cond [(= n 0) true]
                  [else (is-odd? (sub1 n))]))
          (define (is-odd? n)
            (cond [(= n 0) false]
                  [else (is-even? (sub1 n))]))]
    (is-even? n)))
```

> Scope: reusing parameters

Making helper functions local can reduce the need to have parameters “go along for the ride”.

```
;; (countup-to-v1 n) produces a list of the numbers from 0 to n
;; Example:
(check-expect (countup-to-v1 5) (list 0 1 2 3 4 5))

;; countup-to-v1: Nat → (listof Nat)
(define (countup-to-v1 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-v2`

```
;; (countup-to-v2 n) produces a list of the numbers from 0 to n
;; Example:
(check-expect (countup-to-v2 5) (list 0 1 2 3 4 5))

;; countup-to-v2: Nat → (listof Nat)
(define (countup-to-v2 n)
  (local [(define (countup-from from)
            (cond [(> from n) empty]
                  [else (cons from (countup-from (add1 from)))]))]
    (countup-from 0)))
```

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

If we evaluate `(countup-to-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-to-v2 20)`, another renamed version of `countup-from` is lifted to the top level.

Ex. 3

Using only one helper function, which is local and has only one parameter, write a function `(list-squares n)` that produces a list containing the squares of the first `n` natural numbers.

```
(check-expect (list-squares 4) (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

```
(check-expect (mult-table 3 4)
              (list (list 0 0 0 0)
                    (list 0 1 2 3)
                    (list 0 2 4 6)))
```

The c^{th} entry of the r^{th} row (numbering from 0) is $r \times c$.

» mult-table: original

```
;; mult-table: Nat Nat → (listof (listof Nat))
(define (mult-table nr nc)
  (generate-rows 0 nr nc))

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

;; (make-a-row c r nc) produces entries c...(nc-1) of rth row of mult. table
;; make-a-row: Nat Nat Nat → (listof Nat)
(define (make-a-row c r nc)
  (cond [(>= c nc) empty]
        [else (cons (* r c) (make-a-row (add1 c) r nc))]))
```

» mult-table: with local

```
(define (mult-table2 nr nc)
  (local [;; (generate-rows r) produces mult. table, rows r...(nr-1)
          ;; generate-rows: Nat → (listof (listof Nat))
          (define (generate-rows r)
            (cond [(>= r nr) empty]
                  [else (cons (make-a-row 0 r) (generate-rows (add1 r)))]))

          ;; (make-a-row c r) produces entries c...(nc-1) of rth row
          ;; make-a-row: Nat Nat → (listof Nat)
          (define (make-a-row c r)
            (cond [(>= c nc) empty]
                  [else (cons (* r c) (make-a-row (add1 c) r))]))
          ]
    (generate-rows 0)))
```

We will revisit this code again in M16.

Ex. 4

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?

Ex. 5

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,

```
(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.

Terminology associated with local

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.

Substitution rule

An expression of the form `(local [d_1 ... d_n] bodyexp)` is rewritten as follows:

- `d_i` will be of the form `(define x_i exp_i)` or `(define (x_i p_1 ... p_m) exp_i)`. In either case, `x_i` is replaced with a fresh identifier (call it `x_i_new`) everywhere in the `local` expression, for $1 \leq i \leq n$.
- The definitions `d_1 ... d_n` are then lifted out (all at once) to the top level of the program, preserving their ordering.
- What remains looks like `(local [] bodyexp')`, where `bodyexp'` is the rewritten version of `bodyexp`. Replace the `local` expression with `bodyexp'`.

All of this (the renaming, the lifting, and removing the `local` with an empty definitions list) is a **single step**.

Conclusions

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 `local` special form.
- You should be able to use `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.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

`local`

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

```
* + - ... / < <= = > >= abs add1 and append boolean? ceiling char-alphabetic?
    char-downcase char-lower-case? char-numeric? char-upcase char-upper-case?
    char-whitespace? char<=? char<? char=? char>=? char>? char? check-error check-expect
    check-within cond cons cons? cos define define-struct define/trace e eighth else
    empty? equal? error even? exp expt fifth first floor fourth integer? length list
    list->string list? local log max min modulo negative? not number->string number? odd?
    or pi positive? quotient remainder rest reverse round second seventh sgn sin sixth sqr
    sqrt string->list string-append string-downcase string-length string-lower-case?
    string-numeric? string-upcase string-upper-case? string<=? string<? string=? string>=?
    string>? string? sub1 substring symbol=? symbol? tan third zero?
```