Local definitions and lexical scope

Readings: HtDP, Intermezzo 3 (Section 18).

Language level: Intermediate Student

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 x1 exp1) . . . (define xn expn)] 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.

```
(define (t-area 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.

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

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

This is short, 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.

```
(define (t-area2 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.

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

```
(define (t-area3/s a b c s)
  (sqrt (* s (- s a) (- s b) (- s c))))
(define (t-area3 a b c)
  (t-area3/s a b c (/ (+ a b c) 2)))
```

This is more readable, and shorter, but it is still awkward.

The value of s is defined in one function and used in another.

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

```
(define (t-area4 a b c)
  (local
    ((define s (/ (+ a b c) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Full Racket provides several related language constructs. Each of the constructs has simpler semantics, but none is as general as local.

Since local is another form (like cond) that results in double parentheses, we will use square brackets to improve readability. This is another *convention*.

```
(define (t-area4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Reusing names

Local definitions permit reuse of names.

This 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.

The name of a formal parameter to a function may reuse (within the body of that function) a name which is bound to a value through define.

Similarly, a define within a local expression may rebind 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.

Semantics of local

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

It works by creating equivalent definitions that can be “promoted” to the top level.

An evaluation of local creates a **fresh** (new, unique) name for every name used in a local definition, binds the new name to the value, and substitutes the new name everywhere the old name is used in the expression.

Because the fresh names can't by definition appear anywhere outside the local expression, we can move the local definitions to the top level, evaluate them, and continue.

Before discussing the general case, we will demonstrate what happens in an application of our function t-area4 which uses local.

In the example on the following slide, the local definition of s is rewritten using the fresh identifier s_47, which we just made up.

The Stepper does something similar in rewriting local identifiers, appending numbers to make them unique.

Evaluating t-area4

(t-area4 3 4 5) $\Rightarrow$

(local [(define s (/ (+ 3 4 5) 2))]

  (sqrt (* s (- s 3) (- s 4) (- s 5)))) $\Rightarrow$

(define s_47 (/ (+ 3 4 5) 2))

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) $\Rightarrow$

(define s_47 (/ 12 2))

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) $\Rightarrow$

(define s_47 6)

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) $\Rightarrow \ldots$ 6

In general, an expression of the form

(local [(define x1 exp1) ... (define xn expn)] bodyexp)

is handled as follows.

x1 is replaced with a fresh identifier (call it x1_new) everywhere in the local expression.

The same thing is done with x2 through xn.

The definitions (define x1_new exp1) ... (define xn_new expn) are then lifted out (all at once) to the top level of the program, preserving their ordering.

When all the rewritten definitions have been lifted out, what remains looks like (local [] bodyexp'), where bodyexp' is the rewritten version of bodyexp.

This is just replaced with bodyexp'. All of this (the renaming, the lifting, and removing the local with an empty definitions list) is a **single step**.

This is covered in Intermezzo 3 (Section 18), which you should read carefully. Make sure you understand the examples given there.

Naming common 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.

In the following example, the function eat-apples removes all occurrences of the symbol 'apple from a list of symbols.

The subexpression (eat-apples (rest lst)) occurs twice in the code.

```
(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(cons? lst)
         (cond [(not (symbol=? (first lst) 'apple))
                (cons (first lst) (eat-apples (rest lst)))]
               [else
                (eat-apples (rest lst))])]))
```

```
(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(cons? lst)
         (local [(define ate-rest (eat-apples (rest lst)))]
           (cond [(not (symbol=? (first lst) 'apple))
                  (cons (first lst) ate-rest)]
                 [else ate-rest]))]))
```

In the function eat-apples, the subexpression

```
(eat-apples (rest lst))
```

appears in two different answers of the same cond expression, so only one of them will ever be evaluated.

Recall that in lecture module 07, we saw that a structurally-recursive version of max-list used the same recursive application twice.

We can use local to avoid this.

Old version of max-list

```
;; (max-list lon) produces the maximum element of lon
;; max-list: (listof Num) → Num
;; requires: lon is nonempty
(define (max-list lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list (rest lon))) (first lon)]
        [else  (max-list (rest lon))]))
```

```
;; max-list2: (listof Num) → Num
;; requires: lon is nonempty
(define (max-list2 lon) ; 2nd version
  (cond [(empty? (rest lon))  (first lon)]
        [else
         (local [(define max-rest (max-list2 (rest lon)))]
           (cond [(> (first lon) max-rest)  (first lon)]
                 [else  max-rest]))]))
```

This was also the reason that we used a helper function in the computation of a list of ancestors in an evolution tree.

Old version of ancestors-ancient

```
;; ancestors-ancient: Ancient Modern → (anyof (listof Str) false)
(define (ancestors-ancient from to)
  (cond
    [(cons? (ancestors (ancient-left from) to))
     (cons (ancient-name from) (ancestors (ancient-left from) to))]
    [(cons? (ancestors (ancient-right from) to))
     (cons (ancient-name from) (ancestors (ancient-right from) to))]
    [else false]))
```

We can now use local to similar effect.

```
(define (ancestors-ancient from to)
  (local [(define from-l (ancestors (ancient-left from) to))
          (define from-r (ancestors (ancient-right from) to))]
    (cond [(cons? from-l) (cons (ancient-name from) from-l)]
          [(cons? from-r) (cons (ancient-name from) from-r)]
          [else false])))
```

This new version of ancestors-ancient avoids making the same recursive call twice, and does not require a helper function.

But it still suffers from an inefficiency: we always explore the entire evolution 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.

```
(define (ancestors-ancient from to)
  (local [(define from-l (ancestors (ancient-left from) to))]
    (cond
      [(cons? from-l) (cons (ancient-name from) from-l)]
      [else
        (local [(define from-r (ancestors (ancient-right from) to))]
          (cond
            [(cons? from-r) (cons (ancient-name from) from-r)]
            [else false]))])))
```

Using local for clarity

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.

Recall our function to compute the distance between two points.

```
(define (distance posn1 posn2)
  (sqrt (+ (sqr (- (posn-x posn1) (posn-x posn2)))
           (sqr (- (posn-y posn1) (posn-y posn2))))))
```

```
(define (distance posn1 posn2)
  (local [(define delta-x (- (posn-x posn1) (posn-x posn2)))
          (define delta-y (- (posn-y posn1) (posn-y posn2)))]
    (sqrt (+ (sqr delta-x) (sqr delta-y)))))
```

Encapsulation

Encapsulation is the process of grouping things together in a "capsule".

We have already seen data encapsulation in the use of structures.

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

The local bindings are not visible (have no effect) outside the local expression.

In CS 136 we will see how objects combine data encapsulation with another type of encapsulation we now discuss.

We can bind names to functions as well as values in a local definition.

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.

It keeps the “namespace” at the top level less cluttered, and makes the organization of the program more obvious.

This is particularly useful when using accumulators.

```
(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)))
```

Making the accumulatively-recursive helper function local facilitates reasoning about the program.

HtDP (section VI) discusses justifying such code using an **invariant** which expresses the relationship between the arguments provided to the main function and the arguments to the helper function each time it is applied.

For summing a list, the invariant is that the sum of lon equals sofar plus the sum of lst.

This idea will be discussed further in CS 245. It is important in CS 240 and CS 341.

```
(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!

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

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.

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

```
(define (countup-to n)
  (countup-to-from n 0))

(define (countup-to-from n m)
  (cond [(> m n) empty]
        [else (cons m (countup-to-from n (add1 m)))]))
```

```
(define (countup2-to n)
  (local
    [(define (countup-from m)
       (cond [(> m n) empty]
             [else (cons m (countup-from (add1 m)))]))]
    (countup-from 0)))
```

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

If we evaluate (countup2-to 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 (countup2-to 20), another renamed version of countup-from is lifted to the top level.

We can use the same idea to localize the helper functions for mult-table from lecture module 06.

Recall that

(mult-table 3 4) $\Rightarrow$

```
(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 $r \times c$.

```
;; code from lecture module 06
;; (mult-table nr nc) produces multiplication table
;;     with nr rows and nc columns
;; mult-table: Nat Nat → (listof (listof Nat))
(define (mult-table nr nc)
   (rows-from 0 nr nc))

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

```
;; (row r nc) produces rth row of mult. table of length nc
;; row: Nat Nat → (listof Nat)
(define (row r nc)
  (cols-from 0 r nc))

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

```
(define (mult-table2 nr nc)
  (local
    [(define (row r)
       (local [(define (cols-from c)
                 (cond [(>= c nc) empty]
                       [else (cons (* r c) (cols-from (add1 c)))]))]
         (cols-from 0)))
     (define (rows-from r)
       (cond [(>= r nr) empty]
             [else (cons (row r) (rows-from (add1 r)))]))]
    (rows-from 0)))
```

If we evaluate (mult-table2 3 4) using the substitution model, the outermost local is evaluated once.

But (row r) is evaluated four times, for $r = 0, 1, 2, 3$.

This means that the innermost local is evaluated four times, and four renamed versions of cols-from are lifted to the top level, each with a different value of r substituted.

We will further simplify this code in lecture module 10.

The use of local has permitted only 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 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.