Local definitions and lexical scope

Readings: HtDP, Intermezzo 3 (Section 18).

Language level: Intermediate Student

Topics:

• Motivating local definitions [1–11]
• Semantics of local [12–16]
• Reasons to use local [17–42]
• Terminology [43–44]
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.

\[
\text{(local [(define x}_1 \text{ exp}_1) \ldots (define x}_n \text{ exp}_n)] \text{ body}_\text{exp})
\]

What use is this feature?
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.
Heron’s formula version 1

(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))))

Key Point The repeated computation of \( s = (a + b + c)/2 \) is awkward.
Heron’s formula version 2

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

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

This approach is shorter, but its relationship to Heron’s formula is unclear from just reading the code, and the technique does not generalize.
Heron’s formula version 3

*Key Idea:* use a helper function instead.

```
(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))
```
Pros and cons of using a helper function

*Pros:* This generalizes well to formulas that define several intermediate quantities.

*Cons:* 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.
Heron’s formula version 4

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.

\[
\text{(define (t-area3/s a b c s)}
\begin{align*}
&\quad (\sqrt{\ast s (- s a) (- s b) (- s c)}))
\end{align*}
\text{(define (t-area3 a b c)}
\begin{align*}
&\quad (t-area3/s a b c (/ (+ a b c) 2)))
\end{align*}
\]

Pros: This version is more readable and shorter.

Cons: But this version is still awkward: The value of \( s \) is defined in one function and used in another.
Heron’s formula using **local**

**Key Benefit:** 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)))))
```

**Convention:** Since **local** is another 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.

This is not new to us:

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

gives the answer 8, not 12.

*Recall:* The substitution specified in the semantics of function application ensures that the correct value is used while evaluating the last line.
Reusing names

*Key Point:* Both function parameters and `local` have similar semantics.

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*.

*How it works:* 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.
Semantics of local

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.
Example: evaluating t-area4

(t-area4 3 4 5) ⇒

(local [(define s (/ (+ 3 4 5) 2))]
  (sqrt (* s (- s 3) (- s 4) (- s 5)))) ⇒

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

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) ⇒

(define s_47 (/ 12 2))

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) ⇒

(define s_47 6)

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) ⇒ ... 6
Semantics of **local**

In general, an expression of the form

\[
\text{local } [(\text{define } x_1 \text{ exp1}) \ldots (\text{define } x_n \text{ expn})] \text{ bodyexp}
\]

is handled as follows.

\(x_1\) is *replaced with a fresh identifier* (call it \(x_1\_\text{new}\)) everywhere in the \text{local} expression.

The same thing is done with \(x_2\) through \(x_n\).

The definitions \((\text{define } x_1\_\text{new} \text{ exp1}) \ldots (\text{define } x_n\_\text{new} \text{ expn})\) are then *lifted out (all at once) to the top level* of the program, preserving their ordering.
Semantics of `local`

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.
Reasons to use local

1. Clarity: Naming subexpressions [18–19]
2. Efficiency: Avoid recomputation [20–27]
3. Encapsulation: Hiding stuff [28–34]
4. Scope: Reusing parameters [35–42]
Upcoming Slides

The slides for much of the rest of the module take previously discussed functions and rewrite them using local.

- **max-list** (slides 21) is from module 09 slide 4
- **search-bt-path** (slide 23) is from module 11 slide 15
- **isort** (slide 32) is from module 08 slides 4, 8
- **countup-to** (slides 35–36) is from module 07 slides 21–22
- **mult-table** (slides 43–47) is from module 06 slides 39–40
1. Clarity: naming subexpressions

A subexpression used twice within a function body always yields the same value.

*Key Reason:* Using \texttt{local} to \textit{give the reused subexpression a name improves the readability} of the code.

This was a motivating factor in \texttt{t-area}. Naming the subexpression made the relationship to Herron’s Formula clear.

\begin{verbatim}
(define (t-area4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
\end{verbatim}
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 (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))))))
```
2. Efficiency: avoid recomputation

Recall that in lecture module 09, we saw a version of max-list used the same recursive application twice. The repeated computation of values caused it to be very slow, even for lists of length 25.

*Key Reason:* We can use local to avoid recomputation.
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))])))
Improved version of max-list

;; 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]))]))
Old version of **search-bt-path**

;; search-bt-path-v1: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v1 k tree) )  ; v1 original version
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [(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]])
(define (search-bt-path-v2 k tree) ; Use a helper function
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [else (choose-path (search-bt-path-v2 k (node-left tree))
                        (search-bt-path-v2 k (node-right tree))))])

(define (choose-path path1 path2)
  (cond [(list? path1) (cons 'left path1)]
        [(list? path2) (cons 'right path2)]
        [else false])))
Using **local**:  

```scheme
;; search-bt-path-v3: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v3 k tree) ; use local
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()] 
    [else (local [(define left (search-bt-path-v3 k (node-left tree)))
                  (define right (search-bt-path-v3 k (node-right tree)))]
                  (cond [(list? left) (cons 'left left)]
                         [(list? right) (cons 'right right)]
                         [else false]))]))
```
Using **local**: 

Version 3 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**.
;; search-bt-path-v4: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v4 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) ’()]
    [else (local [(define left (search-bt-path-v4 k (node-left tree)))]
                  (cond [(list? left) (cons ’left left)]
                        [else (local [(define right
                                       (search-bt-path-v4 k (node-right tree)))]
                                     (cond [(list? right) (cons ’right right)]
                                           [else false]))]))]))
3. 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.

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 246 we will see how objects combine data encapsulation with another type of encapsulation we now discuss.
**Behaviour encapsulation**

*Key Point:* 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 type of encapsulation is known as *behaviour encapsulation.*
Behaviour encapsulation: benefits

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)))
```

Making the accumulatively-recursive helper function local 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.
Example: Insertion sort \texttt{isort}

\begin{verbatim}
(define (isort lon)
  (local [(define (insert n slon)
             (cond [(empty? slon) (cons n empty)]
                   [(\(<\)\(\leq\)\) 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)))]))
\end{verbatim}

Here the \texttt{insert} helper function is included in the body of \texttt{isort}.
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!
(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)))])))
4. Scope: reusing parameters

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

\[
\text{(define (countup-to } n)\text{)}
\]
\[
\text{ (countup-to-from } n \ 0)\text{)}
\]

\[
\text{(define (countup-to-from } n \ m)\text{)}
\]
\[
\text{ (cond [}(> \ m \ n) \ \text{empty}]}\text{)}
\]
\[
\text{ [else (cons } m \ \text{(countup-to-from } n \ (\text{add1 } m)\text{))})]\text{]}
\]

Here \(n\) must be a parameter in \text{countup-to-from} to know when to stop.
(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 \textit{n} no longer needs to be a parameter to \texttt{countup-from}, because it is in scope.
The semantics of **local**

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.

Multiple copies of very similar functions will exist.
Example mult-table

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

Recall that

\[(\text{mult-table } 3 \ 4) \Rightarrow\]
\[(\text{list (list 0 0 0 0)}\]
\[(\text{list 0 1 2 3)}\]
\[(\text{list 0 2 4 6))\]

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

The next two slides present the code you’ve seen in Module 8.
mult-table explained

In the original version (slides 39–40)

- mult-table applies rows-from to produce the rows. rows-from applies rows to produce a single row. rows applies cols-from to produce each individual column entry in a row.

In the version using local (slide 41)

- rows-from ands row are functions local to mult-table2.
- row has its own local function cols-from.
;; code from lecture module 08
;; (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 ... (cons ( ∗ r c) (cols-from (add1 c) 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))]))
Example: **mult-table using local**

```scheme
(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)))
```

CS 135 Fall 2019

12: Local definitions and lexical scope
More on **mult-table2**

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 module 13.
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*.
Terminology associated with local

Looking back at slide 36...

- The binding occurrence of \texttt{countup2-to} and \texttt{n} occur on line 1.
- The binding occurrence of \texttt{countup-from} and \texttt{m} occur on line 3.
- The associated bound occurrence of \texttt{n} is on line 4 and \texttt{countup-from} is on lines 5 and 6.
- The lexical scope of the parameter \texttt{n} is the entire body of the \texttt{countup-to}.
- The lexical scope of the parameter \texttt{m} is within \texttt{local}. 
Use of `local`

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.
Module 12 Summary

The Special Function `local`

1. The special form `local` contains a series of local definitions plus an expression using them. [2]

2. `local` can be used to avoid repetition of common subexpressions. [3-9]

3. `local` can reuse an existing name. [10-11]

4. Expressions in `local` are replaced with a fresh identifier and lifted to the top level in one step. [15]

5. Using `local` to name subexpressions can help with readability. [18]
Module 12 Summary

Encapsulation

6. **Encapsulation** is the idea of keeping related items together. [28]

7. Encapsulation can also help prevent unintended users from using the location definitions. [28]

8. Both values (**data encapsulation**) and functions (**behaviour encapsulation**) can be encapsulated. [28-29]

9. An **invariant** is a relationship that does not change during the course of the computation. [31]
Module 12 Summary

Binding and Scope

10. The binding occurrence of a name is its use in a definition or formal parameter to a function. [43]
11. The associated bound occurrences are the uses of that name that correspond to that binding. [43]
12. The lexical scope of a binding occurrence is all places where that binding has effect. [43]
13. Global scope is the scope of top-level definitions. [43]