COMP 210, Spring 2001
Lecture 14: Hammering Home "Local"

Reminders:
1. Homework due Friday Wednesday
2. Exam in DH 1055 (McMurtry Auditorium) in class on Friday.

Review

1. We introduced a new Scheme construct, **`local`**, that creates a new name space. We worked an example, and talked about how **`local`** works.

Local
Local takes two complicated arguments

`(local [ <defines> ] <expression> )`

where <***defines***> is a set of one or more definitions, and <***expression***> is a Scheme expression that will be evaluated. Local creates a new *name space***,** or *scope*. It evaluates the definitions within the new scope and then it evaluates the expression in the new scope. Once the expression has been evaluated, its value becomes the value of the **`local.`**

Here's another example of the use of local:

```
; exp5: number -> number
; Purpose: given x, computes x^5
; (define (exp-5 x) …)
; Examples
; (exp-5 0) = 0
; (exp-5 2) = 32
(define (exp-5 x)
    (local [(define (square y) (* y y))]
  (* x (square (square x)))))
```

If we enter this program in DrScheme at the Beginner language level and evaluate **`(exp-5 2)`**, what happens? DrScheme complains bitterly. *We must move to the Intermediate language level* .

If we change the language level to Intermediate, execute the program, and evaluate **`(exp-5 2)`**, DrScheme prints the result **`32.`**

If we try to evaluate **`(square 2)`**, what happens?

DrScheme gives us an error. Why? Because **`square`** exists only inside the new scope created by the **`local`**. When it is evaluating the body of **`exp-5`**, it creates that name space, defines **`square`** and uses it. When it finishes evaluating the **`local`**, that scope goes away and the definitions introduced in that scope no longer exist.

In our formal evaluation rules, we capture the concept of local scope by renaming all of the identifiers introduced in a **`local`** when the **`local`** is evaluated.

Let's trace the execution of **`(exp-5 2)`**:

```
    (define (exp-5 x) …)
    (exp-5 2)
⇒   (define (exp-5 x) …)
    (local [(define (square y) (* y y))]
      (* 2 (square (square 2))))
⇒   (define (exp-5 x) …)
    (define (square' y) (* y y))
    (* 2 (square' (square' 2)))
⇒   (define (exp-5 x) …)
    (define (square' y) (* y y))
    (* 2 (square' (* 2 2)))
⇒   (define (exp-5 x) …)
    (define (square' y) (* y y))
    (* 2 (square' 4))
⇒   (define (exp-5 x) …)
    (define (square' y) (* y y))
    (* 2 (* 4 4))
⇒   (define (exp-5 x) …)
    (define (square' y) (* y y))
    (* 2 16)
⇒   (define (exp-5 x) …)
    (define (square' y) (* y y))
    32
```

Once all of the references to renamed identifiers are evaluated, the local definitions are inaccessible because no program text in the remainder of the evaluation can mention those names. At this point, the local definitions can

be dropped from the prelude of **`define`** statements preceding the expression being evaluated.

Digression: how does DrScheme actually manage the pool of program definitions?

1. You don't really want to know the whole truth (all the gory details).
2. In effect, DrScheme keeps all of the definitions in a list ordered from newest to oldest. This list is called the environment. When DrScheme evaluates a **`local`** embedded in some larger expression, it appends the list of new definitions to the front of the environment. Once the evaluation of the **`local`** is complete, DrScheme resumes evaluation the larger expression using the former environment
3. The rewriting semantics used in our hand evaluations is the governing definition of how Scheme works. The environment method used by DrScheme is justified by the fact that it produces the same results as the rewriting semantics.

When should you use a `local`?

The real justifications for using a local are:

1. To avoid computing some complicated value more than once.
2. To make complicated expressions more readable by introducing helper functions that break the expression up into more tractable parts.

As another example of ways that you can use **`local`,** notice that we can use it to hide an unchanging (or *invariant*) parameter. For example, in the homework due today, you undoubtedly had occasion to develop a function similar to

```
;; is_in?: symbol list-of-symbol → boolean
;; Purpose: return true if the argument symbol is in the list,
;;          otherwise, return false
(define (is_in? asym alos)
  (cond [(empty? alos) false]
        [(cons? alos)
         (or (symbol=? (first alos) asym)
             (is_in? asym (rest alos)))]))
```

This function passes asym at each recursive call, even though it never changes. We can avoid this (for aesthetic reasons, for efficiency reasons, or simply to avoid typing asym that many times in the hand evaluations) by using local.

```
;; is-in?: list-of-symbol symbol → boolean
```

```
;; Purpose: return true if the argument symbol is in the list,
;;          otherwise, return false
(define (is-in? asym alos)
  (local [(define (is-in-help los)
            (cond [(empty? los) false]
                  [else (or (symbol=? (first los) asym)
                            (is-in-help (rest los)))]))]
    (is-in-help alos)))
```

Here, the code avoids passing **asym** around to all the recursive invocations of **is-in-help**. By defining **is-in-help** in a context where **asym** is already defined and visible, we can use it without passing it around. This only works because the function never changes **asym**; it just uses it for the comparison in the **symbol=?** clause. This use of local to avoid passing an invariant parameter might fall under either case of our rule!

Digression: local names

We could have written **is-in?** as follows:

```
;; is-in?: list-of-symbol symbol → boolean
;; Purpose: return true if the argument symbol is in the list,
;;          otherwise, return false
(define (is-in? asym alos)
  (local [(define (is-in-help alos)
            (cond [(empty? alos) false]
                  [else (or (symbol=? (first alos) asym)
                            (is-in-help (rest alos)))]))]
    (is-in-help alos)))
```

Inside the **local**, the inner definition of **alos "shadows"** the outer definition. When substituting a value for a variable, you must not replace shadowed occurrences of the variable!

Example:

```
   (define is-in? …)
   (is-in? 'Corky empty)
⇒ (define is-in? …)
   (local [(define (is-in-help alos)
             (cond [(empty? alos) false]
                   [else (or (symbol=? (first alos) 'Corky)
                             (is-in-help (rest alos)))]))]
     (is-in-help empty)))
⇒  (define is-in? …)
    (define (is-in-help' alos)
      (cond [(empty? alos) false]
            [else (or (symbol=? (first alos) 'Corky)
                      (is-in-help' (rest alos)))]))]
```

```
    (is-in-help' empty)))
⇒   (define (is-in? …) …)
    (define (is-in-help'…) …)

    (cond [(empty? empty) false]
          [else (or (symbol=? (first empty) 'Corky)
                    (is-in-help' (rest empty)))])

⇒   (define (is-in? …) …)
    (define (is-in-help'…) …)
    (cond [false false]
          [else (or (symbol=? (first empty) 'Corky)
                    (is-in-help' (rest empty)))])
⇒   (define (is-in? …) …)
    (define (is-in-help'…) …)
    false
```

A Little Philosophy

Scheme **`local`** gives us a glimpse under the covers of how the Scheme implementation really works. Every time you define a name, you really specify where that name may be used. For example, when you type a name at DrScheme in the interactions window, it responds with an **`error`** unless there has been a corresponding **`define`** for that name in the definitions window.

```
> x
reference to undefined identifier: x
```

But if we type

```
(define (f x) (+ x 3))
```

in the definitions window, DrScheme is perfectly happy with our use of **`x`**. Why? Because the header for the function **`f`** creates a local definition forthe name **`x`**. This definition of **`x`** only exists inside the parentheses that bound the definition of **`f`**. These parameter names that we have been using all semester are actually names with a limited scope–-a limited region in the code where they can be used. This idea isn't new to us; in fact, it should be familiar to us.

`local` creates such a scope. The parentheses that enclose the local construct are the bounds of the scope. One difference between a local "scope" and a function "scope" is that we can use the **`define`** inside a **`local`** to specify the

meaning of the name (and not inside a function). Of course, that means that we can use anything that can legally appear inside a **`define`** within a **`local`**–-including a **`local`**. This creates the possibility for nesting **`local`**s.

```
(define (fee a)
  (local [(define x 2)] (local [(define y 3)] (* a x y))))
```

and so on… The evaluation rules make it clear what happens. What about

```
(define (fie a)
  (local [(define x 2)]
    (local [(define y 3)]
      (local [(define x 17)]
        (* a x y)))))
```

What's the value of **`(fie 1)`** ? => 51

The definition of **`x`** in the innermost local obscures the definition of **`x`** in the outermost local. What if **`x`** is also a function defined outside ***all of the locals*** –- a place that we will call the ***top level***? The innermost **`x`** hides all definitions that are "farther out" in the nested set of locals.

To summarize the behavior (or meaning, or semantics) of **`local`**

```
top-level definitions
(local (defs)
     expression)
```

becomes

```
top-level definitions
[renamed] defs
[renamed] expression
```

Then the right hand sides of the new **`defs`** are evaluated. Finally the **`expression`** is evaluated, at which point it becomes

```
top-level definitions
[renamed] evaluated-defs
[renamed] results
```

Postlude

Some graduates of COMP 210 complain that they never use the concepts from COMP 210 in later courses (suggesting they don't understand object-oriented design as taught in Comp 212).

Today's lecture is a striking example of the fallacy of that statement. Scheme's **`local`** construct is a pure and distilled form of a principle known as *lexical scoping.* The idea was introduced by logicians (notably Alonzo Church) in the λ-calculus, a formal framework for studying the concept of computation. Later it was incorporated in real programming languages in Algol 60 (circa 1960). It is a feature found in most programming languages, including C/C++ (in degenerate form), Pascal, Modula, ML, Ada, Java, and (in its own quirky way) in Smalltalk.

Understanding **`local`** is critical to your ability to program in those languages. The interesting thing about the COMP 210 approach is that we've explained to you how `local` works–-not just-how to use it, but how it works. The *semantics* of **`local`**. You now have the tools to answer complex questions about lexical scoping–-questions that a shallow syntactic introduction to the idea would not make clear.

Because Scheme has such a simple and comprehensible semantics based on algebraic simplication, we can explain the semantics of complicated features such as local in the familiar language of algebra. The knowledge that you gained from this model will come back to help you in all of your programming endeavors, even if you never write another Scheme program after COMP 210.