COMP 210, FALL 2000
Lecture 18: Hammering Home "Local"

Reminders:

1. Next homework is due Wednesday after break
2. Next exam will be a take-home, two-hour exam. It will be handed out 10/20/2000, due 10/27/2000. You will be responsible for lectures through 10/20/2000.
3. You will have a walk on 10/23/2000 due to the Computer Science Corporate Affiliates Meeting. I

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 **context,** or **scope**. It evaluates the definitions in that new context (which creates all the objects that they specify), and then it evaluates the expression inside that context. Once the expression has been evaluated, its value becomes the value of the **local**, replacing the entire local construct (and destroying its context)

Notice that the syntax is

(local [(defines)] expression)

The first argument to local is a list of definitions. The list is enclosed in parentheses. Here's another example.

```
(define (exp-5 x)
   (local ((define (square y) (* y y))
             (define (cube    z) (* z (square z))) )
            (* (square x) (cube x))
    ))
```

If we type this into the definitions window, click execute, and go to the interactions window and evaluate (exp-5 2), DrScheme complains bitterly. *We need to move to the Intermediate language level.*

Once we've done that, we can evaluate (+ (exp-5 2) 3)

```
(+  (local ((define (square y) (* y y))
            (define (cube    z) (* z (square z))) )
           (* (square 2) (cube 2 )) )
      3)
```

To evaluate the local, DrScheme copies the definitions of square and cube to the definitions window, executes them, and then evaluates

```
(* (square 2) (cube 2 ))
```

by creating

```
(* (* 2 2) (* 2 (square 2)))
⇨ (* 4  (* 2 (* 2 2) ))
⇨ (* 4  (* 2 4))
⇨ (* 4   8)
⇨ 32
```

Now comes the tricky part. Remember, the local was executing in the context of

```
⇨ (+  (<huge local mess>) 3)
```

DrScheme replaces <huge local mess> with 32, and (as if by magic) destroys the definitions of square and cube. This creates the expression

```
⇨ (+ 32 3)
⇨ 35
```

What happens if we hand evaluate (cube 2)? **DrScheme gives us an error.** Why? Because **cube** exists only inside the new name space created by the local.

Digression: How does DrScheme throw away a definition?

1. You don't really want to know the truth.
2. The simplest model has DrScheme create a unique new name for each object created by a **define** in the local. Then it rewrites the entire **expression** portion of the local to reflect the new names. Since these names are recorded nowhere else, the objects are not accessible anywhere else. It behaves as if DrScheme deliberately forgets the names.

   This is what DrScheme actually does. There are other models that used by other implementations. As long as you believe the rewriting semantics, you will be safe. As long as those other implementations behave in the same way as the rewriting semantics, they are safe.

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.

(Notice that avoiding the use of invariant parameters might fall under either case!) **But this seems counter-intuitive…**

A Little Philosophy

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 implicitly creates the name x–-with an implicit understanding that the name x only exists inside the parentheses that bound the definition of f. These argument names (or 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 "define" inside a local (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 locals.

```
(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 …
… defs …              <renamed for uniqueness>
    expression        <with renaming>
```

until the expression is evaluated, at which point it becomes

```
… top-level definitions
    result
```

Postlude

A common complaint about COMP 210 (often heard among people who have taken the course and are now taking COMP 212, 320, or 314) is "we never use any of the stuff that we learned in COMP 210."

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, as far as I know, in Algol 60 (maybe earlier). It is a feature found in most programming languages, including C, C++, 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 how to use it, but how it works. You now have the tools to answer some reasonably complex questions about lexical scoping–-questions that a simple syntactic introduction to the idea would not make clear.

Because Scheme has such a simple and comprehensible execution model, we can explain complicated features such as local in the same terms that we used to explain algebraic expressions. 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.

[The strongest defenders of COMP 210 are the seniors that I see in COMP 412. They are almost unanimous in their praise for the course, even though almost none of them use Scheme in their coursework for COMP 412.]