COMP 210, Spring 2001
Lecture 13: Introducing Local

Reminders:
1. Homework due Wednesday.
2. Exam in DH 3055 (McMurtry Auditorium) during class on Friday, February 23, 2001.

Review
1. We looked at three examples of programs with two "complex" arguments. They were **`append`**, **`make-points`**, and **`merge`**. These examples illustrate how our design recipe copes with writing functions where more than one argument has recursive structure. These functions can be partitioned into three distinct groups.
   a) The function does not look inside ("traverse") one of the arguments, so it can use the standard "natural recursion" template for the other argument.
   b) The function traverses both arguments in "lock step". The inputs to such a function must be of the same size for the function's purpose to make sense. Such a function can be written using the standard template corresponding to the first "complex" argument--except that each reference to a selector function for the first argument is paired with a selector function for the second argument.
   c) The function must traverse the recursive structure of both complex arguments but not in lock-step. In this case, the function's purpose does not stipulate that the arguments must be of the same size. To write such a function, we must perform an exhaustive case analysis on the form of both arguments. This analysis is best expressed by a table. We implement this table using a **`cond`** and use recursion only when both arguments have recursive structure. If only argument has recursive structure, we may need a helper function to traverse that particular argument.

Factoring out Common Expressions: A Motivating Example

```
;; max-of-list: list-of-students -> number
;; Purpose: return the largest score attained by a student
in the
;;                argument list
(define (max-of-list) … )
```

Working with the natural recursion template for **`list`** leads us to an interesting quandry–-what should it return for the empty list? What is **`(max-of-list`**

empty) ? We can use a value such as **#i-inf.0** that is smaller than any value that can appear in the list. But this trick is unnecessary if the contract stipulates that the input list is non-empty.

To capture the fact that the list must be non-empty in a contract, we need a data definition for a form of list that excludes the empty list. We can define a *non-empty-list* as follows:

```
;; a nelon (non-empty-list-of-numbers) is either
;; – (cons f  empty), where f is a number, or
;; – (cons f  r), where f is a number and r is nelon
```

This data definition generates the following generic "natural recursion" template

```
(define  (f … a-nelon … )
  (cond
    [(empty? (rest a-nelon)) … (first a-nelon) … ]
    [(cons? (rest a-nelon)) … (first a-nelon) … (f … (rest a-nelon) … ) … ]))
```

With this template we can easily write **max-of-list** and avoid the issue of an empty list.

```
;; max-of-list : nelon -> number
;; Purpose: returns the largest number in the input nelon
(define (max-of-list a-nelon)
   (cond
     [(empty? (rest a-nelon))  (first a-nelon)]
     [(cons?  (rest a-nelon))
      (cond
        [(> (first a-nelon)  (max-of-list (rest a-nelon)))
         (first a-nelon)]
        [else (max-of-list (rest a-nelon))])]))
```

Reflections on max-of-list

First, its name should really be max-of-nelon, not max-of-list. Ignoring that, there is something deeply unsatisfying about this program. It recurs twice, once in evaluating the question (> (first a-nelon) (max-of-list (rest a-nelon))), and the second time if that question evaluates to false. This is problematic for several reasons.

- We wrote the same expression twice. If we need to go back and change it, for example, to instill truth in naming, we need to modify it in several places. We'd like, aesthetically, to have a single point of control. [We've worked several examples in class that fail this criterion. We just haven't pointed them out.]

- If the expression is long and tedious (this one is not), we would rather write it once and read it once. [This is a corollary of the first reason, but in COMP 210, it always seems to get listed separately.]
- Invoking the function twice on the same argument is wasteful. [I know, we keep saying that efficiency is not an objective in COMP 210, but this is getting ridiculous. This program computes the max to figure out whether or not it should compute the max!]

  Consider a list of 6 numbers (list 1 2 3 4 5 6). Invoking max-of-list on it will recur twice on a list of five numbers. Each of those recurs twice on a list of four numbers. Each of those recurs… This leads, quite rapidly, to an exponential blowup in the amount of work required to find a simple maximum. For a list of n numbers, it calls max-of-list 2^n-1 times, or 63 times for our list of 6 elements. (For a list of 7, it takes 127 calls!) If you ask a first grader to solve this problem by hand, they typically go down the list once. Our program should do better than that.

Warning: New Scheme Syntax

Its been a while since we introduced any new syntax in Scheme. [Yes, we've introduced some additional functions, but no new ways of expressing computations.] Today, let's look at the scheme construct **`local`** that is designed to help us out of our quandary with max-of-list.

Local takes two complicated arguments–-a list of definitions and an expression. It creates a new **name space,** or **context,** or **scope** that contains the definitions, then evaluates the expression inside that context. Using **`local`** to rewrite max-of-list, we get

```
;; max-of-list : nelon -> number
;; Purpose: returns the largest number in the input nelon
(define (max-of-list a-nelon)
  (cond
    [(empty? (rest a-nelon)) (first a-nelon)]
    [(cons? (rest a-nelon))
     (local
       [(define maxrest (max-of-list (rest a-nelon)))]
        (cond [(> (first a-nelon) maxrest) (first a-nelon)]
              [else maxrest]))]))
```

In fact, we can do even better.

```
;; max-of-list : nelon -> number
;; Purpose: returns the largest number in the input nelon
(define (max-of-list a-nelon)
  (local
```

```
[(define head (first a-nelon))
 (define tail (rest a-nelon))]
(cond
  [(empty? tail) head]
  [(cons? tail)
   (local [(define max-tail (max-of-list tail))]
    (cond [(> head max-tail) head]
          [else max-tail]))]))
```

Notice that the syntax is

(local [(*def1* … *defn*)] *expression*)

The first argument to local is a list of definitions. The list is enclosed in parentheses. The second argument is an expression.

Local behaves as follows. It creates a new scope–-think of this as a box in the world of Scheme objects. The box has walls that are one-way mirrors. Something inside the box can see through the walls to the outside world, but anything outside the box has no clue as to what is hidden inside the box.

Inside the box, it evaluates the definitions, creating whatever results those definitions imply. After it evaluates the definitions, it then evaluates the expression, ***inside the box***. Thus, the expression sees both the contents of the box and the surrounding context. The expression evaluates to a result–-a value. DrScheme replaces the **local** with that value and discards the box.

Evaluation rules for local

In hand-evaluations, the definitions in a **`local`** are *promoted* to the top-level. Each name (identifier) introduced in a **`define`** within a **`local`** is given a fresh name distinct from all other names in the program (*every* occurrence of the name in the entire **`local`** expression is renamed) and each nested definition is lifted to the top level.

Back to `max-of-list`

In our example, **`max-of-list`** uses a **local** to find the largest value in the **`rest`** of **`a-nelon`**. It saves this result as **`max-tail`** (using the **`define`**). Now, it can reference **`max-tail`** twice–-once in the test and once in the **`else`** clause. The entire program traverses the list once, just as a first grader would.

Notice that it evaluates the local once for each element of the list. Thus, for a list of *n* elements, it will create a nest of *n* boxes. Each box will hold the largest list element found in any of the enclosed boxes. At the outermost box, this produces the largest element from the **`rest`** of the original **`nelon`**, which is compared against the **`first`** element of the **`nelon`**. The result must be the largest element of this list.

This solution examines the list once. It does *n* comparisons. It creates *n* boxes. This is much better than 2^n-1, isn't it.