

COMP 210, FALL 2000

Lecture 17: Introducing Local

Reminders:

1. Homework due Wednesday. The one after that will be due Friday after break (rather than Wednesday).

Review

1. We looked at three examples of programs that took two (a pair of?) complicated arguments. They were **append**, **make-points**, and **merge**. It wasn't clear how our previous practice of writing templates worked on these more intricate examples. They divided into three distinct cases.
 - a) The program does not look inside one of the arguments, so it can use the standard template for the data definition.
 - b) The program uses both arguments completely, but they must be of the same length for the problem instance to make sense. This leads to a simplified template that looks like the standard template, except that each reference to a selector function for the first argument is paired with a selector function for the second argument.
 - c) The program uses both arguments completely, with no assumptions about their relative length. In this case, we need to write down a table to compute the questions that we can ask in the clauses of a **cond** to differentiate between the cases.

Each case leads to a template that we can use to solve the problem. However, that template is a function of the data definition, the contract, and the purpose. This is a significant departure from our prior practice.

This also makes it clear why the book places template development after writing down the contract, purpose, and header, rather than after writing down the data definition.

In your homework, when you encounter a program that consumes two complex arguments, I want you to write out the general template as step 1.5 (between data analysis and contract-purpose-header) and then specialize the template, if possible, as step 4. This balances between the book's notion on where the template should go and my own taste for keeping the template problem-independent.

Back to the exam

Consider the task of writing

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

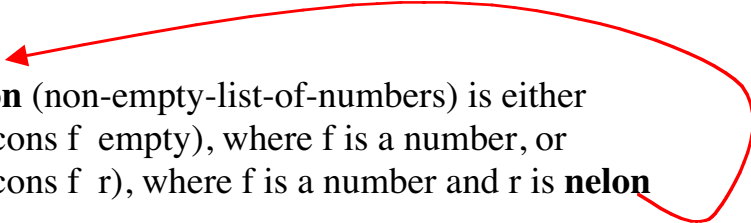
Working with the standard template for **list** leads us to an interesting quandry—what should it return for the empty list? What is (max-of-list empty) ?

On the exam, we had one additional piece of information that made the problem tractable. We had a lower bound on the score that a student could obtain. With a lower bound, we could answer the empty? quandry by returning the lower bound. This simplifies the entire issue. <slide>

To address this quirk of contracts, lists, and arithmetic, the book introduces a slight twist on the notion of a list—it introduces the **non-empty-list**. We can define a non-empty-list

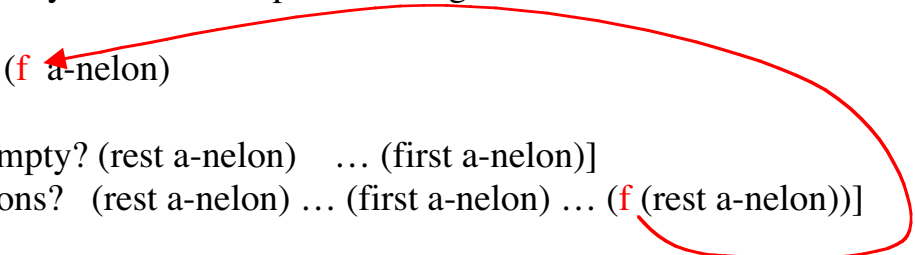
As

```
;; 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
```



Why do it this way? For the template that it generates:

```
(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 sidestep the issue of an empty list. [What we've really done is to restrict the domain of inputs to **max-of-list** so that it excludes the troublesome case—an old and time-honored trick.]

```

;; 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

It's 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 ] )
      )
     ]
  ))
```

Notice that the syntax is

(local ((defines)) (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.

Back to max-of-list

In our example, **max-of-list** uses a **local** to find the largest value in the **rest** of **a-nelson**. It saves this result as **maxrest** (using the **define**). Now, it can reference **maxrest** 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 **nelson**, which is compared against the **first** element of the **nelson**. 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.

Here's another example.

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

This one only makes sense as an example to make the point about local. The straightforward version that multiplies x five times is simpler!

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)`. DrScheme evaluates it to the number 32. If we then type `(cube 2)`, what happens?

DrScheme gives us an error. Why? Because **cube** exists only inside the new name space created by the local. When it is evaluating **exp-5**, it creates that name space, defines **square** and **cube**, and uses them. When it finishes evaluating the local, that name space goes away and those programs no longer can be named. [They exist, in some sense, but are no longer accessible to us.]

More on this next lecture. The homework will hammer away on locals.