

COMP 210, FALL 2000

Lecture 17: Introducing Local

Reminders:

1. Next homework will be available this afternoon, due Wednesday after break

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.*

A Simple Program

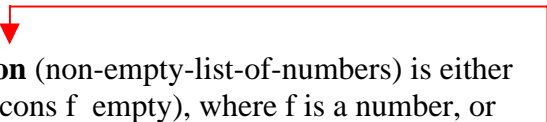
Consider the task of writing

```
;; max-of-list : list-of-numbers -> number
;; Purpose: return the largest number in the input 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) ?

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. If you ask a first or second 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.

If we type `max-of-list` into the definitions window, click execute, and go to the interactions window and evaluate **(max-of-list (cons 1 empty))**, DrScheme complains bitterly. *We need to move to the Intermediate language level.*

Once we've done that, we can evaluate **(max-of-list (cons 1 empty))**. DrScheme evaluates it to the number 1. If we then type **maxrest** what happens? **DrScheme gives us an error.** Why? Because **maxrest** exists only inside the new name space created by the `local`. When it is evaluating **max-of-list**, it creates that name space, defines **maxrest** and

uses it. When it finishes evaluating the local, that name space goes away and the value **maxrest** can no longer be named.

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