COMP 210, Spring 2002
Lecture 16: Finishing up Programs With Two Complicated Arguments

Reminders:
1. New homework out today

Programs that consume two complex arguments
Last class, we started talking about this subject …

merge

```
;; merge:  list-of-numbers list-of-numbers -> list-of-numbers
;; Purpose: takes as input two lists of numbers, which are assumed
;;         to be in ascending order by value and produces a single list
;;        that contains all the numbers, including duplicates, sorted in
;;        ascending order
(define (merge alon1  alon2) …
```

Clearly, merge must look inside both lists. It can make no assumptions about the length of either list. (merge empty (cons 1 empty))) should produce (cons 1 empty). What do we do to produce a template? **Rely on the methodology!** Let's write down examples.
2 inputs, 2 cases in data definition => at least 4 examples

```
(merge empty empty) => empty
(merge empty (cons1 (cons 5 empty)) ) => (cons 1 (cons 5 empty))
(merge (cons 1 (cons 5 empty)) empty) => (cons 1 (cons 5 empty))
(merge (cons 1 (cons 5 empty)) (cons 3 empty))
                    => cons 1 (cons 3 (cons 5 empty)))
```

Our program must be able to handle all these diverse cases correctly. Thus, we need to work out a set of questions that the program can use in the **cond** statement to distinguish them. We can fill in a table to derive the conditions …

Questions for list x list -> list		
	(empty? alon2)	(cons? alon2)
(empty? alon1)	(and (empty? alon1) (empty? alon2))	(and (empty? alon1) (cons? alon2))
(cons? alon1)	(and (cons? alon1) (empty? alon2))	(and (cons? alon1) (cons? alon2))

The table makes the structure of the template clear.

```
(define  (f  alon1  alon2)
  (cond
       [(and (empty? alon1)  (empty? alon2))   …    ]
       [(and (empty? alon1)  (cons?    alon2))  …
              … (first alon2)   … (rest alon2) …      ]
       [(and (cons?    alon1) (empty? alon2))   …
              … (first alon1)   …  (rest alon1) …     ]
       [(and (cons?    alon1)  (cons?   alon2))    …
              … (first alon1)   …  (first alon2) …
              …  (rest alon1) …   (rest alon2) ]
          ))
```

While the structure is clear, the template is missing *all* of the recursion relationships? This case is a little more complex than the ones we've seen in the past.

- In case 1, both lists are empty so there is no recursion.
- In case 2, we need to recur on alon2. However, the function needs two list arguments, not one. What is the other argument? We are tempted to pass in **empty**, but we should pass in alon1, instead. It makes better logical sense, even though we know it is equivalent to empty. (We just tested it.) So, we can use (f alon1 (rest alon2)).
- In case 3, we need to recur on alon1. By symmetry with case 2, we should use (f (rest alon1) alon2).
- In case 4, we should recur on both alon1 and alon2. We have several choices for distinct ways that we could recur. These include

  1. (f alon1 (rest alon2))
  2. (f (rest alon1) alon2)

3. (f (rest alon1) (rest alon2))

We will see cases where each of these is the right thing to do. Since we are building a template, we should write down all of these forms. When we tailor the template to a specific program, we can delete (or cross out) the ones that we do not need.

This leads to our final template for this program.

```
(define  (f  alon1  alon2)
  (cond
     [(and (empty? alon1)  (empty? alon2))   …    ]
     [(and (empty? alon1)  (cons?    alon2))   …
           … (first alon2)   … (f alon1 (rest alon2)) …  ]
     [(and (cons?    alon1) (empty? alon2))   …
           … (first alon1)   …  (f (rest alon1)  alon2) … ]
     [(and (cons?    alon1)  (cons?   alon2))    …
           … (first alon1)   …  (first alon2) …
                  (f  alon1  (rest  alon2))
                  (f  (rest alon1)  alon2))
                  (f  (rest alon1)  (rest alon2))
        ))
```

and then the code for merge almost writes itself …

[You were to write this at home!]

```
(define  (merge  alon1  alon2)
  (cond
     [(and (empty? alon1)  (empty? alon2))   empty       ]
     [(and (empty? alon1)  (cons?    alon2))   alon2       ]
     [(and (cons?    alon1) (empty? alon2))    alon1       ]
     [(and (cons?    alon1)  (cons?   alon2))
       (cond
         [(<  (first alon1)  (first alon2))
          (cons (first alon1)  (merge (rest alon1) alon2))]
        [else
          (cons (first alon2) (merge  alon1  (rest alon2))]

   ))
```

This function, **merge**, forms the core of a general algorithm for sorting.

The Lesson
We saw three kinds of programs that process to complicated inputs:

1. one complex input need not be examined (or traversed)
2. both inputs must be examined in their entirety, but they must always have the same length
3. both inputs must be examined, and we known nothing of their lengths.

Each case leads to a distinctly different template.