COMP 210, FALL 2000

Lecture 16: Working With Two Complicated Arguments

Reminders:

1. Homework due Wednesday
2. Next homework will be available Wednesday, due Wednesday after break

Review

1. We looked at files, directories, and programs that manipulate them. Between class, lab, and the homework, you should have had your fill of family trees.

Arguments to Programs

So far in COMP 210, we have written programs that take, at most, one complicated argument. Real life is not so simple. In lab, you build an **insert** program for a binary search tree where the values were real numbers. In real life, it would be more useful and more interesting to develop a set of programs that perform searches where the information is more complex, as in

```
;; a customer is a
;;    (make-customer name address phone)
;; where name and address are strings and phone is a number
(define-struct customer (name address phone))
```

This information resembles a phone book. Searching a small set of customer records is feasible by hand. Searching the set of telephone customer records for Houston without some sort of automated support (such as an alphabetized phone listing) is impractical.

Of course, the phone company uses techniques, quite similar to those developed in the binary search tree that you saw in lab, to build and maintain its phone lists. Actual phone books are printed from those lists.

To insert a new customer into the Houston phone book, we would need a program that took two complicated arguments—the first a binary search tree over customers and the second a customer. What's hard about that? The key issue, as with much of COMP 210, is getting the right template.

The rest of this lecture works through three different examples, using them to show the variety of templates that can arise from this situation.

[**Warning**: In this lecture, we reach the point where our templates incorporate some program-specific knowledge. Specifically, the templates will need to reflect the contract for the program that we are writing, as well as some knowledge of how that program will use the various pieces of information in its input arguments.]

Examples (Some functions that take two complex arguments)

1. **append**

```
;; append:  list  list  ->  list
;; Purpose:  produces a list with all the elements of the first
;;           argument followed by all the elements of the
;;           second argument
(define (append   list1  list 2)
    (cond
        [(empty? list1)  list2]
        [(cons?    list1)
           (cons  (first list1) (append (rest list1) list2)]
    ))
```

This program never examines its second argument. It never treats it as a list, except to return it, untouched, as the second argument. Thus, we can write this function quite easily by using the standard template for a program that takes a list-valued argument.

2. **make-points**

```
;; a point is
;;   (make-point x y )
;;  where x and y are numbers
(define-struct point  (x y))

;; make-points: list-of-numbers list-of-numbers -> list-of-points
;; Purpose: takes two lists of numbers, interprets them as a list of
;;        x and y coordinates, and produces the corresponding list of
;;        points.
(define (make-points x-list y-list) …)
```

How does the template look? Clearly, make-points must manipulate the contents of both of its arguments. For the program to make sense, however, a simple fact must be true–-both lists must have the same number of elements. This fact simplifies the structure of the template.

```
(define (f x-list  y-list)
   (cond
       [(empty?  x-list) … ]
       [(cons?     x-list) …
          … (first x-list) … (first y-list) …
          … (f  (rest x-list) (rest y-list)) … ]
    ))
```

Given this template, we can develop the program by filling in the blanks and eliding unneeded constructs.

```
;; make-points: list-of-numbers list-of-numbers -> list-of-points
;; Purpose: takes two lists of numbers, interprets them as a list of
;;         x and y coordinates, and produces the corresponding list of
;;         points.
(define (make-points x-list  y-list)
   (cond
       [(empty?  x-list)     empty]
       [(cons?     x-list)
           (cons
               (make-point (first x-list) (first y-list))
               (make-point (rest x-list) (rest y-list)) ) ]
    ))
```

Notice that the template incorporates knowledge of the contract and purpose, making it a function of both the data definition(s) and the program being developed. This is quite a leap away from what we've done previously. This will become even more extreme for problems where we lack the kind of special case knowledge that simplified this template.

3. **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 …

```
(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.