COMP 210, Spring 2001
Lecture 12: Files, Directories, & Folders

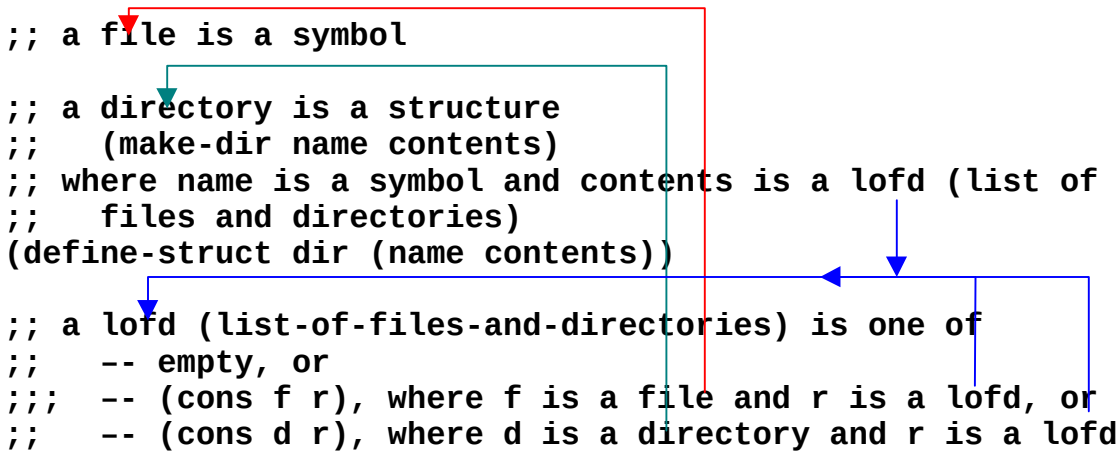
Reminders:

- New homework assignment posted today
- Exam in class on Friday, February 23, 2001. Meet in McMurtry Auditorium.

A more interesting mutually recursive data definition

Consider a "tree-structured" file system for storing files containing data on a computer. Since we are primarily interested in how a "tree-structured" file system is organized, we will represent data files simply as symbols (their names). Later, we can easily extend our definition to add data to files if we want to make our file system more realistic.

Here is how we define a tree-structured file system.



```
;; a file is a symbol

;; a directory is a structure
;;   (make-dir name contents)
;; where name is a symbol and contents is a lofd (list of
;;   files and directories)
(define-struct dir (name contents))

;; a lofd (list-of-files-and-directories) is one of
;;   -- empty, or
;;;  -- (cons f r), where f is a file and r is a lofd, or
;;   -- (cons d r), where d is a directory and r is a lofd
```

The generic template for this set of data definitions has the form:

```
; process a file
(define (f … a-file …) … )

; process a directory
(define (g … a-dir …)
  ( … (dir-name a-dir) … (h … (dir-contents a-dir) …) … )

; process a lofd
(define (h … a-lofd …)
  (cond
    [(empty? a-lofd) …]
    [(symbol? (first a-lofd))
      … (f … (first a-lofd) …) … (h … (rest a-lofd) …) … ]
    [(dir? (first a-lofd))
      … (g … (first a-lofd) …) … (h … (rest a-lofd) …) …]))
```

Let's write a function **count-files: directory → number** that returns the number of files (and directories) in the input directory tree.

Arguments to Programs

So far in COMP 210, we have focused on writing programs that take at most one argument with recursive structure. Real life is not so simple. In the last homework, you needed a function that took two distinct lists as arguments–-two "complex" arguments. This is the first time in COMP 210 that you have used a function with two non-trivial arguments. (You will see more such functions in lab this week.)

What's hard about writing a program with two complex arguments? The key issue is selecting 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.

[**Comment**: Section 17 of the book implicitly acknowledges that templates rely on 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. Of course, we raised this issue in lecture much earlier.]

Examples (Some functions that take two complex arguments)

1. **append**

```
;; append: list list -> list
;; Purpose:  given list1 = (a1 … am) and list2 = (b1 … bn),
;;    (append list1 list2) returns the list (a1 … am b1 … bn)
(define (append list1 list2)
  (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: given two lists of numbers (x1 … xn) (y1 … yn),
;;    returns the list ((make-point x1 y1) … (make-point xn yn))

(define (make-points x-list y-list) …)
```

What template should we use? 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, which must have the same
length, 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-points (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: given two lists of numbers, which must be in
;;    ascending order by value, and produces a single list
;;    that contains all the numbers, including duplicates, 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 design recipe!** 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 gross 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 gross 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.

1. In case 1, both lists are **`empty`** so there is no recursion.

2. In case 2, we need to recur on **alon2**. However, the function **f** needs two list arguments, not one. The book suggests that we call **f** with **alon1** as the other argument. But there is a better approach. Since **alon1** is empty, the computed answer in this case cannot depend on any subsequent processing of **alon1**! As a result, we can use a helper function **f1: list-of-number -> list-of-number** to compute any additional processing that is required on **alon2**. He, we can use **(f1 (rest alon2))** as the form of the answer in this case.
3. In case 3, we need to recur on **alon1**. By symmetry with case 2, we should use **(f2 (rest alon1) alon2)** where **f2: list-of-number -> list-of-number**. It is likely that **f2** will be identical to **f1**.
4. In case 4, we should recur on both **alon1** and **alon2**. We have several choices for distinct ways that we could recur. These include

   - **(f alon1 (rest alon2))**
   - **(f (rest alon1) alon2)**
   - **(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)  … (f1 (rest alon2)) … ]
    [(and (cons? alon1) (empty? alon2))   …
          … (first alon1)  … (f2 (rest alon1)) … ]
    [(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))])))
```

which can be optimized to the form

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

Finger Exercises:

1. Transform the unoptimized form of **`merge`** to the optimized form given above using a sequence of substitutions where each step has a simple justification.
2. Define a function

   `merge-even: list-of-number list-of-number -> list-of-number`
   that takes two sorted lists as arguments and merges the *even* numbers in both lists. This function uses the template given above in more generality than **`merge`** does.

3. Define a function

   `merge-even-odd: list-of-number list-of-number -> list-of-number`
   that takes two sorted lists as arguments and merges the *even* numbers from the first list with the odd numbers from the second list. This function uses the template given above in *full* generality.

The **`merge`** function forms the core of an efficient algorithm for sorting lists.

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.

Another example

```
;; los-equal?: list-of-symbol list-of-symbol → boolean
;; Purpose: returns true if the two lists are, element for
;;   element, identical, and false otherwise
(define (los-equal? los1 los2) … )
```