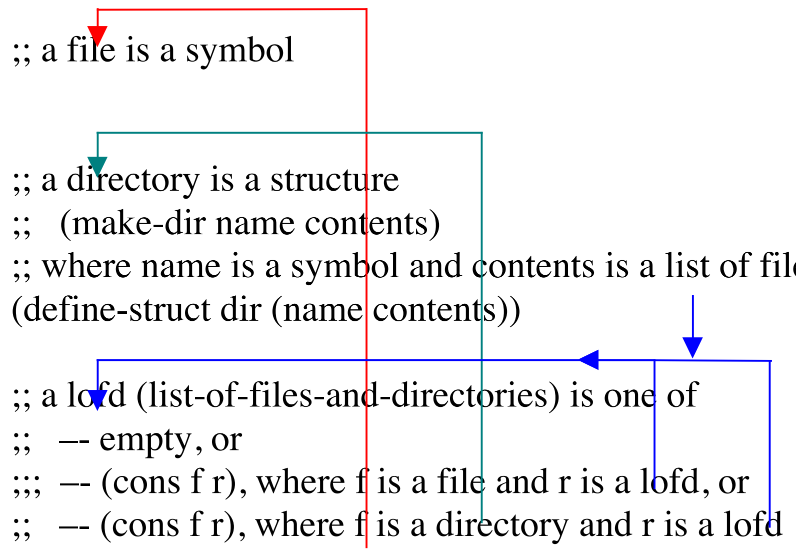
COMP 210, FALL 2000
Lecture 15: More Files, Directories, & Folders

Reminders:
- Homework assignment due Wednesday
- Exams have been graded, will be available tomorrow in DH 3114

Review

1. We looked at another example of a complex information structure implemented as a set of mutually recursive structures. In this case, it was a model for the user interface to a file system.

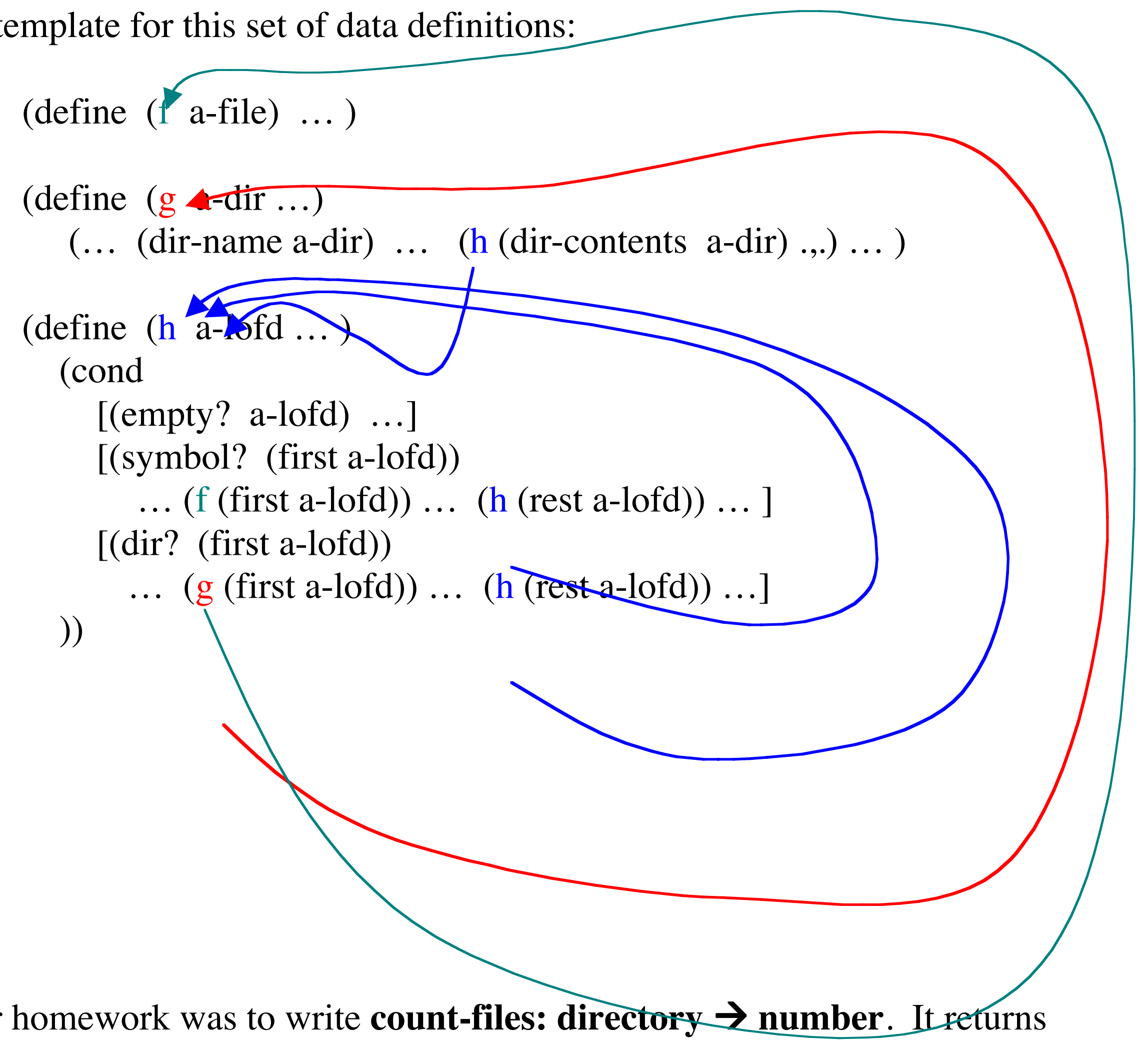


```
;; a file is a symbol


;; a directory is a structure
;;   (make-dir name contents)
;; where name is a symbol and contents is a 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 f r), where f is a directory and r is a lofd
```

The template for this set of data definitions:



```
(define  (f  a-file)  … )

(define  (g  a-dir …)
    (…  (dir-name a-dir)  …   (h (dir-contents  a-dir) .,.) … )

(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)) …]
   ))
```

Your homework was to write **count-files: directory → number**. It returns the number of files (and directories) in the tree rooted at its argument.

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 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 a program with two complex arguments? 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.