COMP 210, FALL 2000
Lecture 15: Files, Directories, Folders, and all that …

Reminders:
- 1/2 Homework assignment due today
- Exam solution is on the web
- Next homework will be available this afternoon, due March 1st in class

Review
1. We spent about half the class talking about the exam, the exam results, and how to interpret them.
2. We went back over family trees a final time, wrote **count-members** and tried to write **at-least-two-kids**. The answer for **at-least-two-kids** is in the posted notes for Lecture 14.

Brief Reflection and Philosophical Interlude
In defining family trees, we looked first at a simple, child-centric model. We learned to answer some questions with it. We expanded it to include information not originally part of our model. We discovered that some questions could not be answered from the child-centric model (such as **number-of-descendants**). We formulated another kind of family tree–the parent-centric model. We used it to answer some questions that could not be answered from child-centric trees. (**count-members** is **number-of-descendants** + 1.)

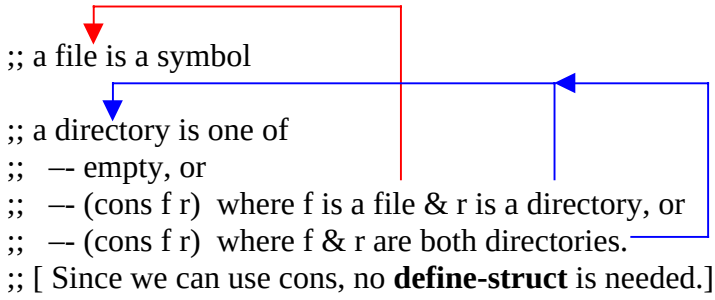
This is the reality of developing computer programs. We propose a model, work with it long enough to discover its shortcomings, and refine it to create a new model. Sometimes, we will propose simple extensions of the existing model; at other times, we will completely discard our earlier models.

From the perspective of programming patterns, we saw a new pattern in the final model for parent-centric family trees–-mutually recursive data definitions. The recursion in the data definition is, of course, reflected in the template and in the code. You should expect the homework for this week to include mutually recursive programs.

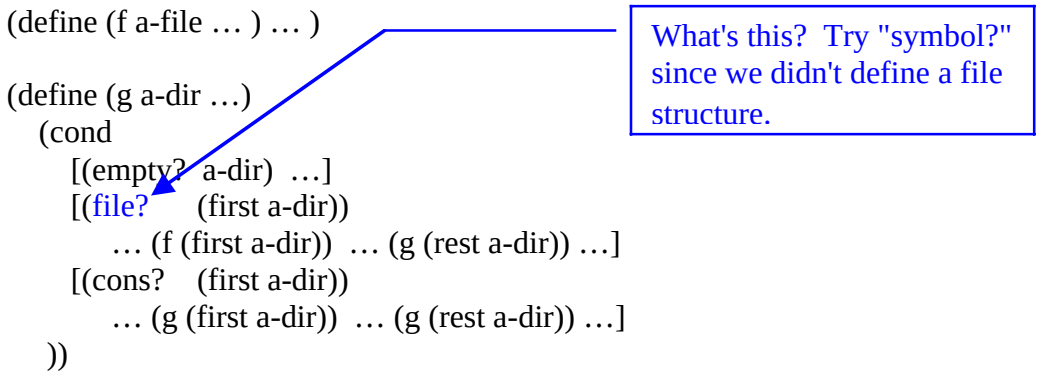
Another Recursive Structure
Let's consider another recursive kind of information that computers model on a daily basis. What about the user interface to a file system? (For the purposes of COMP 210, a file system is just the collection of *files* that reside on the *non-volatile storage* of a *digital computer system*. The notion of files should be familiar to most of you. Note, however, that we haven't seen files in our dealings with DrScheme, except as a place to store some Scheme code between sessions with the Doctor.

File systems are created, manipulated, and destroyed by computer programs–-in most cases, the operating system of the computer. What sort of objects do you find in a file system? A typical file system has files (of course) and directories that hold files. A simple model for the file system might represent files by their names (**symbols** in Scheme), and directories as lists of their contents. Can a directory contain a directory? (For you Mac users, think folders.) How deeply can you nest directories (folders)?

[Several famous operating systems placed arbitrary limits on the depth of nesting in the file system, using the justification that "no one would need more than *k* levels of directories." In reality, these limits usually came from using poor, non-recursive models to represent the name space of the file system.]

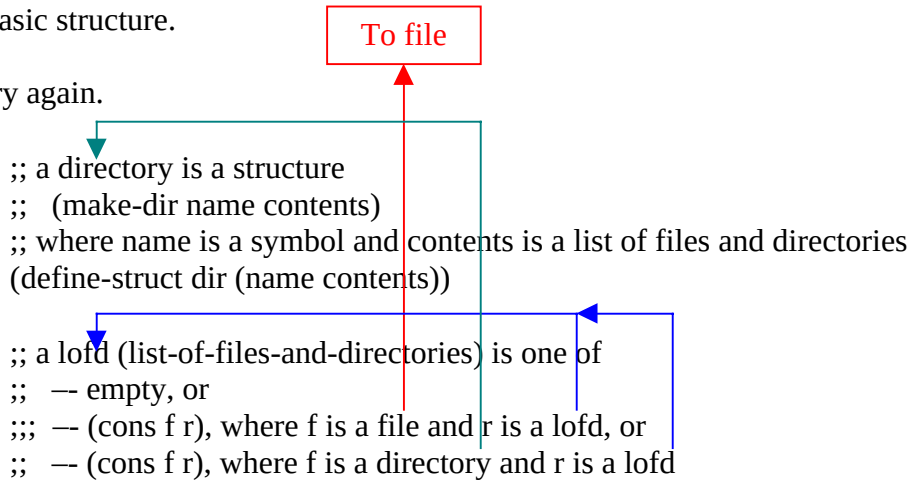


Of course, the template for programs that manipulate files and directories follows from these data definitions.



We could use this to write programs. First, however, we should ask if it is adequate. For example, what is a directory's name? Oooops. The model is not sophisticated enough to allow us to name directories or folders, so it is clearly unusable. (Even though it has the right basic structure.

Let's try again.



We're not going to change the definition of files, since it doesn't materially impact the structure of files and directories. In the homework assignment, you will do this.

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)) …]
   ))
```

Write the program **depth-dir** which consumes a directory and returns a number indicating how many levels of nested directories are in the directory tree.

```
;; depth-dir: directory -> number
;; Purpose: return the number of levels of directories in the tree
(define (depth-dir  a-dir)
   (add1  (depth-lofd (dir-contents a-dir))))

;; depth-lofd: lofd -> number
;; Purpose: return the number of levels of directories in an lofd
(define (depth-lofd  a-lofd)
   (cond
      [(empty?  a-lofd)  0]
      [(symbol?  (first a-lofd)) (depth-lofd (rest a-lofd)) … ]
      [(dir?  (first a-lofd))
          (max
            (depth-dir  (first a-lofd))
           (depth-lofd (rest a-lofd)))]
    ))
```

Write the program **count-files** which consumes a directory and returns the number of files in that directory tree.