Administrative Notes

Homework

- Short homework due Wednesday
- On the web site
- Worth five points

Exams

- Graded and returned Friday
- Statistics in slides for last class (*at the end*)

Files and Directories

Simple model of a file system

→ Files represented as symbols

→ Directory is a list of its contents

```
;; 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 an lofd, or
;;   – (cons f r)  where f is a dir  and r is an lofd
```

```
;; a file is a symbol
```

Files and Directories

Template for File System

```
(define (f  a-file …)  …)                 ;; simple template for file

(define (g a-dir …)                       ;; structure template for dir
   (  …  (dir-name a-dir) …
     …  (h  (dir-contents a-dir) …) … )

(define (h a-lofd …)                      ;; list of several types for 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 : dir -> number
;; Purpose: return nesting depth of
;;    most deeply nested directory
(define (count-files  a-dir) … )
```


Files and Directories

Depth-dir

→ Write a program that consumes a dir and produces a number indicating how many levels of nested directories are in the tree

```
;; depth-dir: dir -> number
(define (depth-dir a-dir)
     (add1 (depth-lofd  (dir-contents a-dir)) ) )

;; depth-lofd: lofd -> number
(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)) ) ]
  ))
```

Programs with Multiple Complex Arguments

So far,

- Programs have consumed, at most, one complicated argument
- In **flatten**, you needed a helper that consumed two lists
  - → This lead to **append**

```
;; append: list  list -> list
;; Purpose: consumes two lists and produces a single list
;;           that contains all the elements of the first argument
;;           followed by all the elements of the second argument
(define (append  list1  list2 )
   (cond
      [(empty? list1)           list2 ]
      [(cons?   list1)  (cons (first list1) (append (rest list1) list2 ))]
   )
)
```

Notice how append *uses* list2

Programs with Multiple Complex Arguments

```
;; append: list  list -> list
;; Purpose: consumes two lists and produces a single list
;;           that contains all the elements of the first argument
;;           followed by all the elements of the second argument
(define (append  list1  list2 )
   (cond
      [(empty? list1)           list2 ]
      [(cons?   list1)  (cons (first list1) (append (rest list1) list2 ))]
   )
)
```

- In **append**, the second argument is never treated as a list
  - → Passed along as second argument in recursive call
- **Append** follows the standard list template

Programs with Multiple Complex Arguments

Another example

```
;; 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: consumes two lists of numbers, interprets the first as
;;            a list of x coordinates, the second as a list of y coordinates,
;;            and produces the corresponding list of points
(define (make-points  x-list  y-list  … )
```

What template should we use?

- Make-points manipulates both x-list and y-list
- Make-points only works if (= (length x-list) (length y-list))

Programs with Multiple Complex Arguments

Another example

- This simplifies 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) … ) … ]
   )
)
```

Only need to test one argument

- We can complete the program from the template
  - → Fill in the blanks
  - → Ellide unneeded stuff

Programs with Multiple Complex Arguments

```
(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)) ) ]
   )
)
```

But, …

- Template contained problem specific-knowledge
  - → This violates our (previous) assumption that templates follow (just) the data structure
  - → Here, template depended on set of arguments to the program
- This is a leap from what we've done in the past

Programs with Multiple Complex Arguments

Another example

```
;; merge : list-of-numbers  list-of-numbers -> list-of-numbers
;; Purpose: consumes two lists of numbers, assumed to be in
;;          ascending order by value, and produces a single list of
;;          numbers that contains all the elements of the input lists
;;          (including duplicates) in ascending order by value
(define (merge  a-lon1  a-lon2) …)
```

- Clearly, merge must look inside both lists
- Clearly, the lists can have different length
  - → (merge empty (cons 1 empty)) should be (cons 1 empty)
- How do we write a template for this problem?
  - → Study the possibilities

Programs with Multiple Complex Arguments

Merge

- Consider the possibilities
- 2 inputs, 2 cases in the definition ⇒ 4 cases & 4 examples

(merge empty empty) ⇒ empty
(merge empty (list 1 5)) ⇒ (list 1 5)
(merge (list 1 5) empty) ⇒ (list 1 5)
(merge (list 1 5) (list 3)) ⇒ (list 1 3 5)

- Merge must handle all of these possibilities
  - → **Cond** construct with four clauses
  - → Must work out tests that distinguish them

Programs with Multiple Complex Arguments

Merge

- Questions for list x list

	(empty? a-lon2)	(cons? a-lon2)
(empty? a-lon1)	(and (empty? a-lon1) (empty? a-lon2))	(and (empty? a-lon1) (cons? a-lon2))
(cons? a-lon1)	(and (cons? a-lon1) (empty? a-lon2))	(and (cons? a-lon1) (cons? a-lon2))

The template must include (and handle) all these cases

Programs with Multiple Complex Arguments

Merge – the template

```
(define (f  a-lon1  a-lon2)
  (cond
    [(and (empty? a-lon1) (empty? a-lon2))     …]
    [(and (empty? a-lon1) (cons? a-lon2))
          … (first a-lon2) …  (rest a-lon2) …]
    [(and (cons? a-lon1) (empty? a-lon2))
          … (first a-lon1) …  (rest a-lon1) …]
    [(and (cons? a-lon1) (cons? a-lon2))
          … (first a-lon1) … (first a-lon2) …
          … (rest a-lon1) … (rest a-lon2) …]
  )
)
```

Structure is clear, but where are the recursion relationships?

Programs with Multiple Complex Arguments

Merge – the template

```
(define (f  a-lon1  a-lon2)
  (cond
    [(and (empty? a-lon1) (empty? a-lon2))     …]
    [(and (empty? a-lon1) (cons? a-lon2))
          … (first a-lon2) …  (rest a-lon2) …]
    [(and (cons? a-lon1) (empty? a-lon2))
          … (first a-lon1) …  (rest a-lon1) …]
    [(and (cons? a-lon1) (cons? a-lon2))
          … (first a-lon1) … (first a-lon2) …
          … (rest a-lon1) … (rest a-lon2) …]
  )
)
```

Empty lists implies no recursion

Recur on a-lon2 and empty.

Same case on a-lon1

Recur on both several cases

(f a-lon1 (rest a-lon2))
(f (rest a-lon1) a-lon2))
(f (rest a-lon1) (rest a-lon2))

Programs with Multiple Complex Arguments

Merge – the template

```
(define (f  a-lon1  a-lon2)
  (cond
   [(and (empty? a-lon1) (empty? a-lon2))     …]
   [(and (empty? a-lon1) (cons? a-lon2))
          … (first a-lon2) …  (f  a-lon1 (rest a-lon2)) …]
   [(and (cons? a-lon1) (empty? a-lon2))
          … (first a-lon1) …  (f (rest a-lon1) a-lon2)…]
   [(and (cons? a-lon1) (cons? a-lon2))
          … (first a-lon1) … (first a-lon2) …
          … (f  a-lon1 (rest a-lon2)) …
          … (f (rest a-lon1) a-lon2) …
          … (f (rest a-lon1) (rest a-lon2)) …]
  )
)
```

You fill in the rest to make merge work