COMP 210, FALL 2000
Lecture 8: Even More Tricks with Lists

Reminders:

- Will set a date for the exam by early next week

Review

1. Did more work with lists, introduced Scheme's built-in list construct, based on **cons**, **first**, **rest**, and **cons?**
2. Talked some about data definitions and when we need them. At this point in COMP 210, we want you to write a data definition for each list construct (even though you use **cons** *et al*.) because the data definition specifies what kind of object is going into the list (list-of-symbol versus list-of-natnum versus list-of-plane).
3. Talked some about templates. The template relates directly to a data-definition. We write a separate template for each kind of information that needs a data definition.

Back to JetSet Airlines

At the end of class, I asked you to solve the following problem. Write a program that consumes a list-of-planes and produces a list containing all the planes that are DC-10s. This is my version.

```
;; just-DC-10s: list-of-plane -> number
;; Purpose:  consumes a list-of-plane and returns the number that
;;           are DC-10s
(define (count-DC-10s a-lop)
  (cond
    [(empty? a-lop)  0]
    [(cons?  a-lop)
      (cond
        [(symbol=? (brand-type (plane-kind(first a-lop))) 'DC-10)
         (add1 (count-DC-10s (rest a-lop)))]
        [else  (count-DC-10s (rest a-lop))])
      ]  ))
```

An alternative way to write this program, suggested to me by one of the students in the spring semester, is

```
;; just-DC-10s: list-of-plane -> number
;; Purpose:  consumes a list-of-plane and returns the number that
;;          are DC-10s
(define (count-DC-10s a-lop)
  (cond
    [(empty? a-lop)  0]
    [(cons?  a-lop)
      (add
        (cond
            [(symbol=? (brand-type (plane-kind(first a-lop))) 'DC-10)
1]
            [else 0] )
        (count-DC-10s (rest a-lop))] ) ]
  ))
```

Is this acceptable? This brings us back to the heart of COMP 210. COMP 210 is a course about a bottom-up, data-driven, design methodology for programming in the small. This alternative formulation will produce the correct results, but it is not the code that the methodology would generate. It is clever, but it is not the code that the methodology would generate. Thus, in COMP 210, it is **not** the code that you should write.

[Yes, it contains one fewer textual copy of "(count-DC-10s (rest a-lop))." That does not make it run faster. That does not make it inherently better. In our judgement, the former version is easier to understand, easier to go back and read ten years later, and (probably) easier to modify. Equally important, from the COMP 210 perspective, it is the code that the design methodology will generate, and COMP 210 is a course about the design methodology.]

Still, the idea isn't bad. The implementation is. If we followed the methodology, we would consider using a separate program for each object that was complex enough to need a data definition. In this problem, that gives us three programs to consider–-one that handles a **list-of-plane**, one that handles a **plane**, and one that handles a **brand**. The need for a program based on the **list-of-plane** is obvious; we must traverse the entire list. What about the other two?

With plane, the only thing that the program would do is apply the selector function **plane-kind** to its argument, so it seems ridiculous (on the surface) to write that program.

```
;; GetPlanesKind: plane→brand
;; Purpose: pull the brand (or "kind") out of a plane
(define (GetPlanesKind a-plane)
   (plane-kind a-plane))
```

This (clearly) is overkill. What about the computation based on **brand**? The program looked inside the brand, tested a value, and did different things based on the result of the test. This is complex enough behavior to encapsulate (or isolate [or abstract]) into a separate program. This suggests the program structure shown in the third example on the slides.

```
;; A cleaner formulation that uses a helper function because
;; we are going to access two distinct kinds of data (planes
;;  and brands)
;;
;; First, the helper function

;; Is-DC10: brand → number
;; Purpose: consumes a brand and returns 1 if the brand's
;;                type is DC10 and returns 0 otherwise
(define (Is-DC10 a-brand)
  (cond
    [(symbol=? (brand-type a-brand) `DC10)   1]
    [else                                    0]
   ))

;; and, the desired program

;; just-DC-10s: list-of-plane -> number
;; Purpose:  consumes a list-of-plane and returns the
;;                   number that are DC-10s
(define (count-DC-10s a-lop)
 (cond
   [(empty? a-lop)  0]
   [(cons?  a-lop)
     (add  (Is-DC10 (plane-brand (first a-lop)))
           (count-DC-10s (rest a-lop)))]
 ))
```

A More Complex Variation

Write a program **all-the-brand** that consumes a list-of-plane and a symbol and produces a list-of-plane containing all the planes whose brand matches the symbol. Build on your knowledge from **just-dc10s**. You can use the same data-definitions and example data.

```
;; all-the-brand : list-of-plane symbol -> list-of-plane
;; Purpose: consumes a list-of-plane and produces a list-of-plane
;;          that contains all the planes whose type matches the
;;          second argument
(define (all-the-brand a-lop kind) … )

;; Templates
;;
;; for brand
;; (define ( ... a-brand ... )
;;    ( ... (brand-type    a-brand) ...
;;      ... (brand-speed   a-brand) ...
;;      ... (brand-seats   a-brand) ...
;;      ... (brand-service a-brand) ... ))
;;
;; for plane
;; (define ( ... a-plane ... )
;;    ( ... (plane-tailnum   a-plane) ...
;;      ... (plane-kind   a-plane) ...
;;      ... (plane-miles     a-plane) ...
;;      ... (plane-mechanics a-plane) ... ))
;;
;; for list-of-symbols
;; (define ( ... a-los ... )
;;    (cond
;;      [(empty? a-los) ... ]
;;      [(cons?  a-los) ... (first a-los) ... (rest a-los) ... ]
;;    ))
;;
;; for list-of-planes
;; (define ( ... a-lop ...)
;;    (cond
;;      [(empty? a-lop) ... ]
;;      [(cons?  a-lop) ... (first a-lop) ... (rest a-lop) ... ]
```

```
;;    ))


;; all-the-brand : list-of-plane symbol -> list-of-plane
;; Purpose: consumes a list-of-plane and produces a list-of-plane
;;          that contains all the planes whose type matches the
;;          second argument
(define (all-the-brand a-lop kind)
  (cond
    [(empty? a-lop)    empty]
    [(cons?  a-lop)
      (cond
        [(symbol=? (brand-type (plane-kind (first a-lop))) kind)
         (cons (first a-lop) (all-the-brand (rest a-lop) kind)) ]
        [else
         (all-the-brand (rest a-lop) kind)]
      )]
    ))
```