COMP 210, Spring 2002
Lecture 8: Even More Tricks with Lists

Reminders:

- Exam date will be Wednesday, February 13, 2002

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

Of course, JetSet Airlines doesn't want a system where they must type the name of each plane into DrScheme. If they succeed, they could end up with hundreds or thousands of planes. Thus, they need to organize the set of planes. To do this, we can create a list of all their planes.

```
;; a list-of-planes is either
;;        – empty, or
;;        – (cons  first  rest)
;; where first is a plane and rest is a list-of-plane
;; we will use Scheme’s built-in list constructo

;; a plane is a
;;   (make-plane tailnum kind miles mechanic)
;; where tailnum is a symbol, kind is a brand, miles is a number, and
;; mechanic is a list of symbols

;; example data
;;
;;  (define brand1 (make-brand `DC-10    550  282 15000))
;;  (define brand2 (make-brand `MD-80   505  141 10000))
;;  (define brand3 (make-brand `ATR-72 300    46   5000))
;;  and
;; (define N1701 (make-plane `N1701 brand1 0  empty))
;; (define N3217 (make-plane ‘N3217 brand3 0 empty))
;; …
;; Now, the list of planes
;;  (define LOP
;;     (cons N1701
;;         (cons  N3217
;;             (cons  N1211
;;                 (cons  N9510  empty) ) ) ) )
;;
```

Develop a program that consumes a list-of-planes and produces the number of planes that are DC-10s.

```
;; count-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-kind (plane-kind(first a-lop))) 'DC-10)
          (add1 (count-DC-10s (rest a-lop)))]
         [else  (count-DC-10s (rest a-lop))])
       ]  ))
```

Programs That Return Lists

Write a program that consumes a list-of-planes and produces a list containing all the planes that are DC-10s.

```
;;  just-DC-10s: list-of-planes -> list-of-planes
;;  Purpose: builds a new list that contains the subset of 'a-lop' that
;;        are 'DC-10s
;; (define (just-DC-10s a-lop) … )
```

Need examples and templates

(use standard list template)

```
;; just-DC-10s : 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 2nd argument
 (define (just-DC-10s a-lop)
  (cond
   [(empty? a-lop)    empty]
   [(cons?  a-lop)
      (cond
        [(symbol=? (brand-kind (plane-kind (first a-lop))) 'DC-10)
         (cons (first a-lop) (just-DC-10s (rest a-lop) 'DC-10)) ]
        [else
         (just-DC-10s (rest a-lop) kind)]
      )]
   ))
```

How does this work? It constructs a new list that brings together those elements of the argument list contain DC-10s. On the empty list, it returns an empty list –- the correct answer by definition.

If the list is non-empty, the program tests each element to see if its brand is 'DC-10. If it finds a 'DC-10, it uses **cons** to prepend that element to the list formed by checking (recursively) the rest of the list.

Given a list-of-planes

```
(define N1701 (make-plane 'N1701 brand1 0 empty))
(define N3217 (make-plane 'N3217 brand3 0 empty))
(define N1215 (make-plane 'N1215 brand2 0 empty))
(define N4512 (make-plane 'N4512 brand1 0 empty))
```

```
(cons N1701
      (cons N3217
            (cons N1215
                  (cons N4512 empty) ) ) )
```

when we apply **just-DC10s** to this list it:

```
(just-DC10s (list N1701  N3217  N1215  N4512)
➔ (cons  N1701  (just-DC10s (list N3217  N1215  N4512) ))
   ➔ (cons N1701 (just-DC10s (list  N1215  N4512) ))
      ➔ (cons N1701 (just-DC10s (list N4512) ))
         ➔ (cons N1701 (cons N4512  (just-DC10s empty) ))
            ➔ (cons  N1701  (cons  N4512  empty) )
```

Alternative Formulation–*moving the COND inside the ADD*

An alternative way to write this program, suggested to me by one of the students last year, 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)))]
  ))
```