COMP 210, Spring 2001
Lecture 5: Programming with Lists, Again

Reminders:

- Homework assignment due next Wednesday
- Read Sections 9 through 11

Review

What have we accomplished, here at the end of the week?

1. Learned to write simple programs based on algebra over real numbers
2. Learned to organize data into aggregate structures–a process we call working with compound data in COMP 210
3. Learned to aggregrate date in the form of lists–a tool for handling arbitrary amounts of data.

   When we have a problem with a fixed amount of data, we can treat it as compound data, unless the amount is so large that manipulating the name space becomes a problem. [As in (define mechanics (repair0 repair1 repair2 … repair100)]

Through all of this, you should be reading the book. It develops a systematic methodology for building these programs that I can only approximate in class. We have built up a six-step design recipe for developing these small programs. That recipe should carry you forward for the rest of COMP 210 and for much of your programming experience in the future.

Today, we'll go back over lists, writing programs with lists, and look at other applications of the idea of a list.

Bubba-serves? one more time…

```
;; a list-of-symbol is either
;;    – empty, or
;;    – (make-Plist f r)
;;        where f is a symbol and r is a list-of-symbol
(define Plist (first rest))

;; General template for processing a list-of-symbols
;; (define ( f … a-los …)
;;     (cond
;;         [(empty? a-los) … ]
;;         [(Plist? a-los)   …  (Plist-first a-los)
;;                …  (f … (Plist-rest  a-los) … )]
```

The final case in the **`cond`** has become more complicated. We write down the selector expressions for each of the pieces of a **`Plist`** (and apply them to the parameter **`a-los`**).

Finally, we can fill in the entire program:

```
      ;; bubba-served? :  list-of-symbol -> bool
      ;; Purpose: return true if Bubba is in the list
      (define (bubba-served? a-los)
           (cond
              [(empty? a-los)   false]
              [(Plist? a-los)
                  (cond
                       [(symbol=? (Plist-first a-los) 'Bubba)  true ]
                       [else (bubba-served? (Plist-rest a-los))])]))
```

Notice that the case for a **`Plist`** in the **`cond`** has two cases. These cases arise from the two cases in the problem statement (not in the data definition).

- If the **`mechanic`** is **`'Bubba`**, we're done   (**`true`** or **`false`** = **`true`**)
- If the **`mechanic`** is not **`'Bubba`**, we need to look farther down the list (and we know we can because we are not in the clause of the outer **`cond`** for **`empty`**). To accomplish this, we call **`bubba-served?`** again to reflect the recursion in the data definition.

Test on **`empty`**, on **`(list 'Bess 'Mike 'Susan 'Bubba)`**, on **`(list 'Fred 'Jane 'Felix)`**

Another example

```
;; count-services: list-of-symbol -> number
;; Purpose: count number of times this plane has been services
(define (count-services a-los)
    (cond
        [(empty? a-los)  0]
        [(Plist? a-los)  (add1 (count-services (Plist-rest a-los)))]))
```

This example ignores **`(first a-los)`** because it doesn't care about the contents of **`(first a-los).`** It simply counts any maintenance record, rather than looking for specific mechanics.

`Plist` versus `cons`

In the preceding examples, we defined our own proper list structure **`Plist`**. We also showed how we could define our own empty list structure **`Empty`** and define the variable **`empty`** to be the **`Empty`** object **`(make-Empty)`**, but we elected to use the built-in **`empty`** data object instead. The data definition for **`list-of-symbol`** shows that it is either **`empty`** or a **`Plist`**, where the second component of a **`Plist`** is a **`list-of-symbols`**.

This data definition is so fundamental to computation that a version of it called simply **`list`** is built into the Scheme language.

```
;; a list is either
;;    – empty, or
;;    – (cons f r)
;;  where f is an arbitrary Scheme object and r is a list
```

`cons`, **`first`**, and **`rest`** are built-in Scheme functions. I've always remembered the name **`cons`** as an abbreviation for *list constructor*.

`cons`	≡	**`make-Plist`**	[**cons** checks its 2nd argument to make sure it's a list]
`first`	≡	**`Plist-first`**	
`rest`	≡	**`Plist-rest`**	
`cons?`	≡	**`Plist?`**	

Henceforth, we will use the built-in data structure **`cons`** instead of **`Plist`**, eliminating the need to include a definition of **`Plist`** at the beginning of every program that computes with lists.

Putting Lists to Other Uses

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 Brand is structure
;;     (make-Brand  type speed seats service)
;; where type is a symbol and speed, seats, and service are numbers
(define-struct Brand (type speed seats service))
;; a Plane is a structure
;;    (make-Plane tail-num kind  miles  mechanic)
;; where tail-num is a symbol, kind is a brand, miles is a number,
;; and mechanic is a symbol
(define-struct Plane (tail-num kind miles mechanic))

;; a List-of-planes is either
;;    – empty, or
;;    – (cons f r)
;; where f is a plane and r is a List-of-plane
;; a plane is a
;;   (make-Plane tn b mi me)
;; where tail-num is a symbol, b is a Brand, mi is a number, and
;;   me 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))
(define n1211 (make-Plane ‘N1211 brand2 0 empty))
(define n9510 (make-Plane ‘N9510 brand1 0 empty))
;; …
;; Now, the list of planes
;;  (define lop
;;     (cons n1701
;;         (cons n3217
;;             (cons n1211
;;                 (cons n9510 empty)))))
;;
```

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

```
;;  just-dc10s: list-of-planes -> list-of-planes
;;  Purpose: builds a new list that contains the subset of 'a-lop' that
;;           are 'DC-10s
;; (define (just-dc10s a-lop) … )
```

Design Recipe
Let's review the design recipe for programs that use lists (and other recurseive data definitions). We'll use **`bubba-served?`** as an example, and finish writing the function.

1. Data analysis – determine how many pieces of data describe interesting aspects of a typical object mentioned in the problem statement; add a data definitions for each kind ("class") of object in the problem

   For **`bubba-serve?`** we need a structure that can hold zero or more names

```
;; a list-of-symbols is either
;; – empty or
;; – (cons f r)
;;  where f is a symbol and r is a list-of-symbols
;;
;; Using built-in list constructor, we don't need the define-struct


;; examples
   empty
   (cons 'Bess (cons 'Mike (cons 'Susan (cons 'Bubba empty))))
```

2. Contract, purpose, header

```
;; bubba-served? : list-of-symbols -> boolean
;; Purpose:  determine whether  'Bubba is on the "mechanic" list
;; (define (bubba-served? a-los) …)
```

3. Test Cases

```
;; (bubba-served? empty) = false
;; (bubba-served? (cons 'Bess
;;                  (cons 'Mike
;;                      (cons 'Susan
;;                          (cons 'Bubba empty))))) = true
;; (bubba-served? (cons 'Fred (cons 'Jane (cons 'Felix empty)))) =
;;     false
```

4. Template – for any parameter that is a compound object, write down the selector expressions (access functions?). Template is problem-independent outline for the code body.

```
;; (define (f … a-los …)
;;      (cond
;;          [(empty? a-los) … ]
;;          [(cons? a-los)  …  (first a-los) …
;;                          (f … (rest a-los) … ) … ]
```

   It is easier to use such a template if we adapt it to the exact form of the function that we have to write:

```
;; (define (bubba-served? a-los)
;;      (cond
;;          [(empty? a-los) … ]
;;          [(cons? a-los)  …  (first a-los) …
;;                          (bubba-served? (rest a-los)) … ]
```

5. Write the body (using the template)

```
;; bubba-served? :  list-of-symbol -> bool
;; Purpose: return true if Bubba is in the list
(define (bubba-served? a-los)
    (cond
        [(empty? a-los) false]
        [(cons? a-los)
            (cond
                [(symbol=? (first a-los) 'Bubba) true]
                [else (bubba-served? (rest a-los))])]))
```

As you write the body, consider each clause in the **cond** separately. You don't need to think about the **`cons?`** clause when your are writing the **`empty?`** clause.

6. Test the program (using the examples from step 3)