

## COMP 210, FALL 2000

### Lecture 8: Programming with Lists, Again

#### Reminders:

- Homework assignment 1 available today, due Wednesday, February 9, 2000
- 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 write recursive programs over more structured domains, such as the natural numbers
3. Learned to organize data into aggregate structures—a process we call working with compound data in COMP 210
4. Learned to provide meta-organization 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 methodology for developing these small programs. That methodology 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...

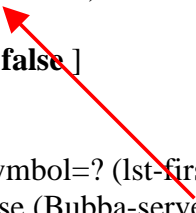
```
;; a list-of-symbol is either
;;   – empty, or
;;   – (make-lst name rest)
;;   where name is a symbol and rest is a list-of-symbol
(define lst (first rest))

;; Template for list-of-symbols
;; (define ( ... a-lst ... )
;;   (cond
;;     [(empty? a-lst) ... ]
;;     [(lst? a-lst) ... (lst-first a-lst)
;;     ... (lst-rest a-lst) ... ]
```

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

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-lst)
  (cond
    [(empty? a-lst) false]
    [(lst? a-lst)
     (cond
       [(symbol=? (lst-first a-lst) 'Bubba) true]
       [else (Bubba-served? (lst-rest a-lst))])
     ])
  ))
```



Notice that the case for a **lst** 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)

### LST versus LIST

In the example, we defined our own list constructor, named **lst**. The data definition for **lst** shows that it is either **empty** or a pair, where the second element of the pair is a **lst**. This construct is so fundamental to Scheme, and so heavily used in Scheme, that a version of it is built into the language.

```
;; a list is either
;;   – empty, or
;;   – (cons first rest)
;; where first is an arbitrary Scheme object and rest 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-lst    (cons checks its 2nd argument to make sure it's a list)
first ≡ lst-first
rest  ≡ lst-rest
```

### Another example

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

This example ignores (first a-1st) because it doesn't care about the contents of (first a-1st). It simply counts any maintenance record, rather than looking for specific mechanics.

### 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 list-of-planes is either
;;   – empty, or
;;   – (cons first rest)
;; where first is a plane and rest is a list-of-plane
;; 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))))))
;;
```

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 Methodology

Let's review the design methodology for programs that use lists (and other recursive data definitions). We'll use **Bubba-serve?** 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 first rest)
;; where first is a symbol and rest is a list-of-symbols
;;
;; Using the built-in list construct, we don't need the define-struct
```

```
;; examples
```

```
empty
```

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

2. Contract, purpose, header

```
;; Bubba-serve? : list-of-symbols -> boolean
```

```
;; Purpose: determine whether 'Bubba is on the "mechanic" list
```

```
;; (define (Bubba-serve? a-los) ...)
```

3. Test Cases

```
;; (Bubba-serve? empty) = false
```

```
;; (Bubba-serve? (cons 'Bess
```

```
                    (cons 'Mike
```

```
                        (cons 'Susan
```

```
                            (cons 'Bubba empty)))) ) = true
```

```
;; (Bubba-serve? (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 ( ... a-los ...)
```

```
;;   (cond
```

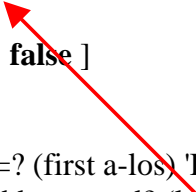
```
;;     [(empty? a-los) ... ]
```

```
;;     [(list?  a-los)   ... (first a-los)
```

```
;;     ... (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]
    [(list? 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 **list?** clause when you are writing the **empty?** clause.

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