COMP 210, FALL 2000
Lecture 7: Lists, more lists, & even more lists

Reminders:

- Homework assignment 0 due at the start of class, today (**NOW**)
- Homework assignment 1 available today, due Wednesday, February 9, 2000
- Rice Computer Science Club meets today, 5:30pm, DH 3092 (“football pitch”)
- Read Sections 9 through 11
- First exam will be Wednesday, 16 February 2000. It will cover material presented in class, lab lectures, and homework through class on 9 February 2000. I am fixing the dates for the second and third exams.

Review
Last class, we built a more complex example with **define-struct**. We talked about keeping records for an airline. We defined structures for a **brand** and for a **plane**.

```
;; 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 tailnum kind  miles  mechanic)
;; where tailnum is a symbol, kind is a brand, miles is a number,
;; and mechanic is a symbol
(define-struct plane (tailnum kind miles mechanic))
```

We wrote a program **max-dist** that consumed a brand and a number of hours and produced the maximum distance that a plane of that brand can travel in the given number of hours. We wrote a program **service**: that consumes a plane and a symbol and produces a new plane with its miles to service reset to zero and the symbol listed as the most recent mechanic.

Finally, we saw how to use **define** to create some persistent data in the Scheme workspace, so that we can test programs without typing in all of those **make-plane** and **make-brand** invocations.

Design Methodology (See page 69)
Working with structured data requires us to add some steps to our design methodology.

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
2. Contract, purpose, header
3. Examples
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.
5. Write the body (using the template)
6. Test the program (using the examples from step 3)

As we work with information that has more complex internal structure, the process of writing the program body becomes more involved. In our early examples, we could fill in the program body intuitively–relying on our knowledge of subjects such as high-school algebra and physics. To cope with the added complexity that comes from the structure of the information, we write down a code template that includes all the access functions that we have for the data in the problem.

For our example of a plane, the template looks like

```
(define (… a-plane …)
   ( … (plane-tailnum a-plane) …
       (plane-kind     a-plane) ….
       (plane-miles    a-plane) ….
       (plane-mechanic a-plane) … ))
```

There is no name, because we want to reuse the template for other problems over the same information structures

We literally write this down–it is the problem–independent part of the program. The template reflects that structure of **plane**, not the structure of the problem. When we write a specific program, we may not need all these selector expressions. We write down the full set, anyway, to remind us of the possibilities.

If the program uses several distinct structures, we will create several distinct templates. We won't combine them into a single template, for two reasons. First, we don't want any one function to become too complex. Second, as we develop more complex programming patterns, we will reach a point where using a single function becomes so complex that we should avoid it at almost any cost.

As we write the code body, we copy pieces of the code template into the appropriate positions. The role of the template is to remind us of the possibilities, not to force us into using all of them.

Tying Together Related Pieces of Information (into lists)

Going back to JetSet Airlines, we know that the FAA actually requires JetSet Airlines to keep distinct records for every time a mechanic works on a plane. To keep these records, we need to replace the simple symbol that we've used for **mechanic** with a structure that holds more than one mechanic. If we know, from domain knowledge, that the number of repairs is small (with a fixed upper bound), we could just add a structure.

```
(define repairs (repair0  repair1  repair2 … repairn))
```

The structure of this problem doesn't admit that kind of solution. We need to store an unbounded (or unknown) number of **mechanics**. In real life, we have an organizing concept that we use–a **list**. We write *to do* lists, *grocery* lists, *class* lists, *wish* lists, … The list is conceptually simple, but it allows us to describe collections of arbitrary size.

An example list might be <Bess, Mike, Susan, Bubba>

To turn this into a Scheme data structure, we need a little more formality. What's the shortest list you can envision? What about the degenerate case of an empty list? In Scheme, we write **empty** to represent the empty list. What about more complex lists, like the one we just wrote? What about <Fred, Jane, Felix>? Is that a list?

What's the relationship that these have in common? A list, it seems, consists of a name at the top (the first part), and everything that follows it (the rest). As long as we let the definition of a list include **empty**, we can write down a struct that captures this notion:

```
(define-struct lst (first rest))
```

We can use this **struct** to make some examples:

```
;; 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))

;; Examples of lst

empty

(define OneList
   (make-lst Bess
      (make-lst 'Mike
         (make-lst 'Susan
            (make-lst 'Bubba empty) ) ) ) )

(define AnotherList
    (make-lst  'Fred
        (make-lst  'Jane
            (make-lst  'Felix  empty) ) )
```

How would we get Bess out of OneList? (lst-first OneList)

What about Mike? (lst-first (lst-rest OneList))

What about Susan? (lst-first (lst-rest (lst-rest OneList)))

Let's write a short program that uses lists. Write a program, **Bubba-served?** that conumes a **list-of-symbol**s that represents the mechanics who have serviced a plane, and returns **true** if the list contains 'Bubba.

```
;; Bubba-served?:  list-of-symbol -> boolean
;; Purpose: return true if Bubba's name occurs in the list
;; (define (Bubba-served? a-lst) … )

;; Test data
(Bubba-served?  empty ) = false
(Bubba-served?  OneList) = true
(Bubba-served?  AnotherList) = false

;; Template
;; (define (…    a-lst … )
;;    (cond
;;        [ … ]
;;        [ … ] ))
```

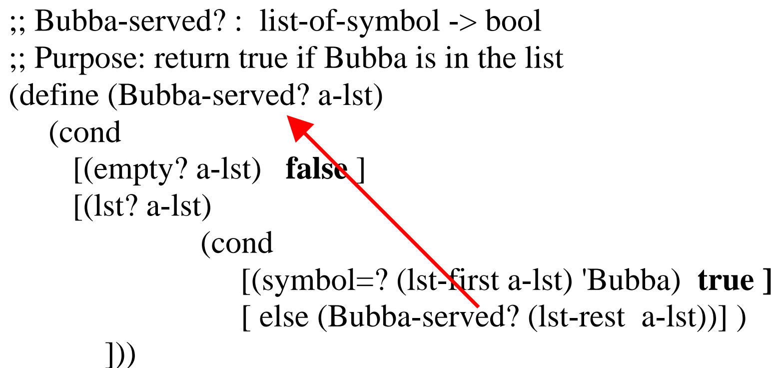
Two cases in a **cond** because the data definition has two clauses.

What questions do we ask in the clauses of the **cond**? To detect **empty**, Scheme provides an operator **empty?** – we use that in the first clause. When Dr. Scheme executes a **define-struct**, it (also) creates a function to test for an instance of the defined structure. For (**define-struct** lst (first rest)), it creates the function **lst?** – we can use that one in the second clause.

```
;; Template
;; Bubba-served? :  list-of-symbol -> bool
;; Purpose: return true if Bubba is in the list
;; (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 use a recursive call on **Bubba-served?** that reflects the recursion in the data definition.

Test on **empty**, OneList, and AnotherList.

The lab lecture (today and tomorrow) will review **define-struct**, build on today's introduction to **lists** and reinforce this expanded notion of **templates**. On Friday, we'll do a some more work with lists to reinforce the ideas, and go over the expanded design methodology.