COMP 210, FALL 2000
Lecture 6: Lists, more lists, & even more lists

Reminders:

- Homework assignment 2 is due Wednesday
- Today we start material that falls in Sections 9 through 11

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 in the book for a summary)
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 (service a-plane a-mechanic)
   ( … (plane-tailnum a-plane) …
       (plane-kind     a-plane) ….
       (plane-miles    a-plane) ….
       (plane-mechanic a-plane) … ))
```

We literally write this down–it is the problem–independent part of the program. We may not need all these selector expressions in a specific program, but we write down the full set. Then, 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.

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)

The most artificial aspect of the programming that we have done to date is the form that the input takes. As many of you have observed (publicly or privately), there is little point in writing a three line program to pick the mileage out of a **make-brand** and test it against a single number. Typing the **make-brand** takes more effort than comparing the two numbers. Today, we are going to talk about the way that Scheme programs tie together related pieces of information. We will be able (next class) to use this mechanism to construct complex and persistent sets of input data.

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 can replace the symbol for **mechanic** with a list of mechanics names. [Later, we can expand these into more complex records for each service action.]

An example list might be <Eddie, Mike, Patty, 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 relationship do 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 (f r))
```

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

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

;; examples of los
empty

(define OneList
   (make-lst 'Eddie
      (make-lst 'Mike
         (make-lst 'Patty
            (make-lst 'Bubba empty) ) ) ) )

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

How would we get Eddie out of OneList?	(lst-first OneList)
What about Mike?	(lst-first (lst-rest OneList))
What about Patty?	(lst-first (lst-rest (lst-rest OneList)))

Let's write a short program using 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-los) … )

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

;; Template
;; (define (…   a-los … )
;;    (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 (Bubba-served? … a-los …)
;;     (cond
;;        [(empty? a-los) … ]
;;        [(los? a-los)     …  (los-f a-los)
;;                          … (Bubba-served? (los-r  a-los)) … ]
```

The recursive call reflects the recursion in the data definition. 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 ]
      [(los? a-los)
                (cond
                    [(symbol=? (los-f  a-los) 'Bubba)  true ]
                    [ else (Bubba-served? (los-r  a-los))] )
         ]))
```

Next class, we'll try this out on the test data and review how we got here.