

COMP 210, Spring 2002

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))
```

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.

Relating Data from Different Kinds of Structures

In addition to facts about models of plane, the airline also needs to keep information about individual planes. Again, the structure of this information should be based on the kinds of questions that programs will need to ask. Clearly, the airline needs to track mileage and service on a plane-by-plane basis (and if that is not clear, there is a little matter of federal regulation).

Let's define a plane

```
;; 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))
```

Here, tailnum is the plane's identifying registration number, kind is a brand, miles is the number of miles flown since the plane was serviced, and mechanic is the name of the person who serviced the plan.

Example Data:

```
(define N1701 (make-plane `N1701 brand1 10000 'Bubba))
(define N3217 (make-plane 'N3217 brand3 6500 'Jane))
```

These definitions create objects in the Scheme workspace that we can use as test data in our programs.

Working With Complex Data

Let's write a program **service** that JetSet airlines can use when a mechanic works on a plane. **Service** should consume a plane and a mechanic's name, and return a *new* plane that reflects the service.

```
:: service: plane symbol -> plane
;; Purpose: update a plane's record to reflect service
(define (service a-plane a-mechanic) ... )
```

Example:

```
(service N1701 `Fred) → (make-plane `N1701 brand1 10000 `Fred)
```

The template:

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

To write the body of **service**, we use the relevant parts of the template and throw the rest away (or erase it). Combining the header and the template, we get something like:

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

Going the next step, we fill in

```
(define (service a-plane a-mechanic)
  (make-plane
   (plane-tailnum a-plane)
   (plane-kind a-plane)
   0
   a-mechanic ))
```

This is a zero (for miles)

A More Complex Program

Last class, you developed a program **needs-service?** that consumes a **brand** and a **number** and returns a **Boolean**. Can we write a program **bring-it-in?** that consumes a **plane** and produces a **Boolean**, where the result is **true** if and only if the plane needs service? Clearly it must work with both **plane**

and **brand**. If we reuse **needs-service?**, then we can formulate **bring-it-in?** as follows:

```
; bring-it-in?: plane → boolean
; purpose: determine if a specific plane needs to be serviced
(define (bring-it-in? a-plane)
  (needs-service? (plane-brand a-plane) (plane-miles a-plane)))
```

This example shows us something about the design methodology. We developed it as two programs—one that deals with **planes** and one that deals with **brands**. We used the template for **plane** when we developed **bring-it-in?** and the template for **brand** when we developed **needs-service?**

This will, in fact, become a design principle. When a problem involves working with several distinct structures—several kinds of information—we should think of it as several distinct, smaller problems. To the extent possible (and in COMP 210, it should *always* be possible), we should **encapsulate** our knowledge of structures in this way.

As a principle in program design, this goes by many names: information hiding, abstraction, or encapsulation. This notion will appear again and again in your study of programming, your study of software systems, and your study of discrete mathematics.

Notice that this is different than the example from Lab. In the lab, the structure was the union of two ways of representing a point in space (one kind of information, two representations): a Cartesian coordinate and a polar coordinate. Thus, the template had a **cond** that allowed the program to handle the two structures in appropriate manners. In **bring-it-in?**, we have two related structures that store fundamentally different kinds of information. **Brand** represents generic information about a type of aircraft, while **plane** represents information about a specific plane. Each **plane** is an *instance* of a **brand**, but they are not variants of each other.

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.

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-los 'Eddie
    (make-los '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:

This lecture fell apart at this point. I mis-wrote the contract for **Bubba-served?** by having it take a **plane** rather than **a-los**. This destroyed the example. Fortunately, we were only five minutes from the end of class, so I gave up and dismissed class. We will revisit this example in Lecture 7 and get it right (I hope).

Write a program, **Bubba-served?** that consumes a **list-of-symbols** 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
;;     [ ... ]
;;     [ ... ]))

```

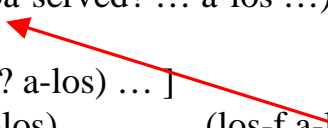
Two cases in a **cond** because the data definition has two clauses.

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)) ... ]

```

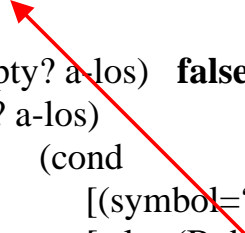


The recursive call reflects the recursion in the data definition. Finally, we can fill in the entire program:

```

;; 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.