COMP 210, FALL 2000
Lecture 6: Adding More Structure

Reminders:
- Homework assignment 0 due Wednesday at the start of class
- Sections 6 & 7

Review
Last class, we:
1. Introduced symbols and built a couple of interesting programs, even though almost no interesting operators work on symbols. We discussed the fact that the set of symbols is not ordered, since neither < nor > operate on a symbol. The set of symbols is infinite, but lacks the formal, recursive structure of the natural numbers.
2. Introduced the Scheme mechanism for defining aggregate structures—new kinds of informaton: **define-struct**. Define-structure creates a formal definition for the aggregate, along with a set of auxiliary programs, including a constructor and a set of destructors (or access programs).

Segue
Our example of the class information system is rather limited—after all, how many interesting things can we say about the COMP 210 staff. Today, lets move to another **domain**—records for a small airline.

JetSet Air
JetSet airlines operates three different kinds of planes in its fleet: DC-10s, MD-80s, and ATR-72s. Of course, they need to keep many distinct kinds of records on these planes. The structure of their "database" should be influenced by the kinds of questions they need to ask. These include:

1. How many seats does the DC-10 have?
2. How often must an ATR-72 be serviced?
3. What is the top speed of an MD-80?

These questions all relate to a specific class of aircraft, rather than to an individual plane. This suggests the following structure:

```
;; 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))
```

Example data for JetSet Airlines might be

```
(make-brand   `DC-10      550    282    15000)
(make-brand   `MD-80      505    141    10000)
(make-brand   `ATR-72     300    46     50000)
```

To build up our queries, we could construct **max-dist**, a program which takes a **brand** (the structure) and a number of hours and returns the maximum distance that a plane can fly in that time.

```
;; max-dist: brand  num -> num
;; Purpose: compute the maximum distance that a brand can fly in a given number
;;              of hours
(define (max-dist  a-brand  hours) …)
```

Test data:

```
(max-dist (make-brand `ATR-72 300    46   5000)  0) = 0
(max-dist (make-brand `DC-10    550  282 15000)  2) = 1100
(max-dist (make-brand `MD-80   505  141 10000) 10) = 5050
```

And, fill in the function body:

```
(define (max-dist  a-brand hours)
        (*  (brand-speed  a-brand)  hours) )
```

Hand evaluation of one or more examples.

In-class Example (5 minutes)
Write a program "needs-service?" which consumes a **brand** and a number of miles, and returns **true** if the brand must be serviced after having flown the given number of miles.

1. Data analysis
2. Contract, purpose, and header
3. Examples
4. Write the body
5. Test the code

Of course, the airline 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).

More Complex Structures
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:

```
(make-plane 'N1701 (make-brand `DC-10 550 282 15000)  10000  'Bubba)
(make-plane 'N3217 (make-brand `ATR-72  300  46 5000)   6500   'Jane)
```

Writing out these examples explicitly becomes tedious. It is also highly unrealistic. We can create an object that holds one of these aggregate structures using **define**.

```
(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 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.

We can test **max-dist** against these brands:

```
(max-dist  brand1  2) = 1100
```

Working With Complex Data

Let's write a program **service** that JetSet airlines can use when a mechanic works on a plane. **Service** should take 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) … )
```

To write this program, we need to pull together all of the information that we have about the data structures. We do this by writing a template for the function that incorporates all of the access functions we have for the data.

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

This is the set of all the access programs that we have for the **plane** structure. Presuming that **service** will use some of them, it helps to write them down. This template leaves blank some of the parts that we already know—that seems artificial. For example, we know that we are writing a program service that takes two arguments. Why not fill in those parts of the template? Because this template fits any program written to manipulate the **plane** structure. This is the "problem independent" part of the code that depends entirely on the data structure. The contract, purpose, and header are entirely "problem-specific," and independent of the details of how a **plane** is put together.

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 ))
```

So, in the last two lectures, we've added two steps to our design methodology (or design recipe, as the book prefers to call it).

1. Data analysis – write down data definitions for all the structures
2. Contract, purpose, header
3. Examples
4. Template – write down all the access programs that we might use in the form of a program body
5. Write the body (using the template)
6. Test the program (using the examples from step 3)

Next Class—Tying data together