COMP 210, Spring 2001

Lecture 3B: Adding More Structure (JetSet Air)

Reminders:

- Sections 6 & 7 in the book

Write out the five part design methodology on the top board. Leave space after Examples for Template

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-struct creates a formal definition for the aggregate, along with a set of auxiliary programs, including a constructor and a set of selectors (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 **data 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    5000)
```

To build up our queries, we could construct **`max-dist`**, a program which takes a **`Brand`** (a 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?`** that 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. Follow the five steps of the methodology.

More Complex 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 tail-num brand miles mechanic)
;; where tail-num is a Symbol, brand is a Brand, miles is a Number,
;; and mechanic is a Symbol
(define-struct Plane (tail-num brand miles mechanic))
```

Here, **`tail-num`** is the plane’s identifying registration number, `brand` 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)  3500  ‘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 dc10  (make-Brand `DC-10  550  282 15000))
      (define md80  (make-Brand `MD-80  505  141 10000))
      (define atr72 (make-Brand `ATR-72 300   46  5000))
```

and

```
      (define n1701 (make-Plane `N1701 dc10  10000 ‘Bubba))
      (define n3217 (make-Plane ‘N3217 atr72  3500 ‘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 dc10 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 manipulate the data elements embedded inside a **`Plane`** object. To guide us through this process, we first write athe template for the function that incorporates all of the accessors we have for the input **`a-plane`**.

```
      (define ( … a-plane …)
          ( … (Plane-tail-num a-plane) …
            … (Plane-brand a-plane) ….
            … (Plane-miles a-plane) ….
            … (Plane-mechanic a-plane) … ))
```

This template includes all of the accessors for a `Plane` structure. The 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 a `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 we represent a plane as a structure.

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-tail-num 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-brand a-plane)
      0
      a-mechanic))
```

Updated Design Recipe

In the last two lectures, we've added two steps (noted by an * below) to our design recipe:

1. *Data analysis – write down data definitions for all the data objects involved in the program
2. Contract, purpose, header
3. Examples
4. *Template – write down a schematic program body containing all the accessor operations we might use in the body
5. Write the body (using the template)
6. Test the program (using the examples from step 3)

Next Class—Aggregating data