COMP 210, Spring 2002

Lecture 5: 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. Worked too many programs using **cond** and hand evaluated a number of them.

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 **information 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

```
(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             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
```

This exercise shows us that we want to multiply the speed by hours.

And, fill in the function body:

To fill in the function body, we must use the facts that we know about **brand**s. As a matter of methodology (and a crutch for those of us with bad short-term memories), we write it all down in a ***template***.

```
(define (… a-brand …)
    (… (brand-type a-brand) …
     … (brand-speed a-brand) …
     … (brand-seats  a-brand) …
     … (brand-service a-brand) …)
)
```

The template lacks a program name, because it is problem-independent. It relies solely on the structure of the input data.

The template shows us all of the pieces of information that are available to us for a brand. It explicitly lists the access-programs, or *selectors*, that let a program obtain the value for those pieces of information inside the structure.

Given the template, we can write the program by filling in the details, from our examples, and discarding the rest.

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

Hand evaluation of one or more examples.

We plug templates into the methodology right after examples.

The template 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.

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. Follow the five steps of the methodology.

Summary
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)

See the summary on page 71 of the book.

Next Class—Tying data together