COMP 210, Spring 2001
Lecture 3A: Moving Beyond Numbers

Reminders:
1. Anyone still looking for a homework partner?
2. Homework 1 due Wednesday, Homework 2 will be available Wednesday afternoon
3. Read the book. Sections 5-7.

Review
Last class, we:

1. Built another small program in pizza economics
2. Talked about the methodology
   a) Contract, purpose, & header
   b) Work some examples
   c) Develop the body
   d) Test the code

Segue
Our initial attempts at programming in Scheme operated over the domain of numbers. Our goal in COMP 210 is to compute over richer data domains than just numbers. For example, we might want to assign classes to classrooms; this would require computing over some domain that included abstractions for classes (time, department, and size) and for concrete structures like HZ 212 (number of seats, projection facilities). This clearly goes beyond numbers.

A common kind of information is a **word**. In Scheme, we represent words by using **symbols**. A symbol looks like a word, except that it has a single quote mark on the front. A symbol can contain any string of letters, except for a blank. A blank ends the symbol. The notion of a "letter" is interpreted loosely, so that it means letters, numbers, and some kinds of punctuation—dash is legal, semicolon is not—the definition is a little idiosyncratic, but you can always test it directly in Dr. Scheme.

Examples:

```
'Comp210          'Rice             'Scheme
'Ryon-102         'Scott-Schaefer   'Zung-Hguyen
'Cheryl-Hom       'Jamie-Raymond    'Corky-Cartwright
```

What can we do with a symbol in Scheme? Does **`(+ 'Jamie 4)`** make any sense? No. **`'Jamie`** is not a number, so **`+`** should not work on it. The only operation that makes sense on symbols (in Scheme) is comparing them for equality. The Scheme syntax for this kind of comparison is

```
(symbol=? 'Corky 'Jamie) = false
(symbol=? 'Pizza 'Pizza) = true
```

[**Notice** that we can compare for equality, but not for magnitude. **`(> 'Corky 'Jamie)`** generates an aborting error message reporting that **`'Jamie`** is not a number. So does **`(< 'Corky 'Jamie).`** Last class, we used these operators on numbers; numbers are totally ordered. **`(< x y)`** has an answer, for any numbers `x` and `y`. Symbols are <u>not ordered</u>. Hence, we can only compare them for equality.]

We can use symbols in a program. For example, consider the program **`office-hours`**.

```
;; office-hours: symbol -> symbol
;; Purpose: report the office hours for COMP 210 Staff
(define (office-hours name)
    (cond
        [(symbol=? name 'Corky)  'M-13:30-to-15:00]
        [(symbol=? name 'Zung)   'MWF-11:00-to-11:50]
        [(symbol=? name 'Jamie)  'T-14:30-to-16:00]
        [(symbol=? name 'Scott)  'Th-12:30-to-14:30]
        [(symbol=? Name 'Cheryl) 'W-16:00-18:00]))
```

We can use this capability to implement a small database. We might also want to know

```
;; office-number: symbol -> symbol
;; Purpose:  report the office number of COMP 210 Staff members
(define (office-number name)
    (cond
        [(symbol=? name 'Corky)  'DH-3104]
        [(symbol=? name 'Zung)   'DH-3096]
        [(symbol=? name 'Jamie)  'DH-2064]
        [(symbol=? name 'Scott)  'DH-3116]
        [(symbol=? Name 'Cheryl) 'DH-3063]))
```

We might also want to know their phone numbers…. Hey wait a minute, this is getting pretty tedious. This can't be the right way to keep this information—building a separate program for each fact related to the staff member's name.

All of these functions have (and are going to have) a similar structure. [**Remember**: the fundamental thesis of COMP 210 is that the program structure should reflect the structure of the underlying data. We shouldn't be surprised that all the access programs for one set of data look similar.]

Building More Complex Information Structures

Isn't there a better way to do this? Should the information about a staff member be scattered across an array of small programs, or should it be centralized in one place—a place where it can be created, where it can be changed, where any program that needs it can find the information.

Scheme provides a construct for grouping together a collection of data items that the programmer decides belong together. This is the first principle of data design: put together those things that naturally belong together! The Scheme notation for this data definition is

```
(define-struct SN (info-1 info-2 info-3 … info-n))
```

This line of code tells DrScheme "I need a new form of data. I would like to call it **SN** and each **SN** has an **info-1**, an **info-2**, and so on…" The **define-struct** construct creates a new kind of **compound data** and gives it a name chosen by the programmer. When you write a **define-struct** in the DrScheme definitions window and execute it, DrScheme creates a set of functions for constructing and manipulating your new form of compound data. The first such function is

```
(make-SN info-1 info-2 info-3 … info-n)
```

Since executing a `define-struct` has complex actions, we need to document each **define-struct** just as we would document a program.

Emphasize this as part of the methodology.

Let's make this more concrete.

```
;; A Staff is a structure
;; (make-Staff name office-number office-hours position)
;; where name, office-number, office-hours and position are symbols
(define-struct Staff (name office-number office-hours position))
```

This data definition creates several functions.

```
(make-Staff name office-number office-hours position)
```

takes its arguments, creates a **Staff** with these fields, and returns it. We call **make-Staff** a **constructor** for **Staff** members.

define-struct creates these functions for you!

Along with constructors, **define-struct** creates **selectors** (or **accessors)** — one for each data item, or **field**, in a staff member. It names these programs

```
Staff-name              Staff-office-hours
Staff-office-number     Staff-position
```

Note that the functions

```
;; Staff-office-hours: Staff → symbol
;; Purpose: return the office hours of a given 210 staff member

;; Staff-office-number: Staff → symbol
```

```
;; Purpose: return the office number of a given 210 staff member
```

are similar to our earlier functions **`office-hours`** and **`office-number`**. These accessor functions do not completely duplicate the earlier functions. The earlier functions had, embedded inside them, all of the data. Thus, they took a staff member's name and returned the appropriate data. In contrast, the accessor functions take a **`Staff`** structure and return the appropriate data. We have not addressed the issue of where the table of **`Staff`** data resides or how to search it to find the **`Staff`** structure for a person given the person's name. In a subsequent lecture, we will introduce a compound form of data that can aggregate **`Staff`** structures and discuss how to search such an aggregate object for a particular **`Staff`** stucture.

We can use the access functions in other Scheme programs. For example

```
;; in-charge:  Staff -> boolean
;; Purpose:  returns true if a Staff is a teacher, false otherwise
      (define (in-charge a-staff)
          (cond
              [(symbol=? (Staff-position a-staff) 'teacher) true]
              [else false]))

(in-charge (make-Staff 'Corky 'DH3104 'Monday 'teacher))    = true
(in-charge (make-Staff 'Scott 'DH3116 'Tuesday 'assistant)) = false
```

Impact of Data Definition on Our Design Methodology

We must insert the step

Data Analysis and Design

before

Contract, Purpose, and Header