COMP 210, FALL 2000
Lecture 4: Moving Beyond Numbers

May need Friday's lecture for parts of Homework 2

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

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 information domains than 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 'Pizza
'Ryon-102 'Keith-Cooper 'John-Greiner
'Tim-Harvey 'Jamie-Raymond

What can we do with a symbol in Scheme? Does (+ ‘Tim 4) make any sense? No. ‘Tim 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=?   ‘Keith ‘Tim)        = false
(symbol=?   ‘Pizza ‘Pizza)      = true
```

[**Notice** that we can compare for equality, but not for magnitude. (> ‘Keith ‘Tim) elicits an error message about the fact that Tim is not a number as does (< ‘Keith ‘Tim). 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</u>, since we can only compare them for equality.]

We can use symbols in a program. For example, consider the program OfficeHours.

```
;; OfficeHours: symbol -> symbol
;; Purpose: report the office hours for COMP 210 Staff
(define (OfficeHours Name)
    (cond
        [(symbol=? Name ‘Keith) ‘Monday-13:30-to-15:00]
        [(symbol=? Name ‘John)   ‘Tues-&-Wed-13:00-to-14:00 ]
        [(symbol=? Name  ‘Tim)   ‘Monday-13:30-to-15:00]
        [(symbol=?  Name ‘Jamie)  ‘Mon-&-Wed-14:00-to-15:00]
        ))
```

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

```
;; OfficeNumber: symbol -> symbol
;; Purpose:  report the office number of COMP 210 Staff members
(define (OfficeNumber Name)
    (cond
        [(symbol=? Name ‘Keith)  ‘DH-2065]
        [(symbol=? Name ‘John)   ‘DH-3118]
        [(symbol=? Name  ‘Tim)   ‘DH-2064]
        [(symbol=?  Name ‘Jamie) ‘DH-3107]
        ))
```

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 bunch of information that the programmer decides should go together. [This is the first principle of abstraction, as well as locality: put together those things that should go together!] The Scheme incantation for this is

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

This tells Dr. Scheme "I need a new kind of information. I would like to call it SN and each SN has an info-1, an info-2, and so on…" Define-struct creates a new kind of **compound data** and gives it a name (that you choose). When you write a define-struct in the definitions window and execute it, Dr. Scheme creates a set of functions for 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 the define-struct just as we would document a program.

Let's make this more concrete.

Emphasize this as part of the methodology.

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

This creates several functions.

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

takes its arguments, creates a staff with this data, and returns it. We call make-staff a **constructor** for staff members.

define-struct creates these small programs for you!

Along with constructors, define-struct creates **selectors** or **access programs**—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

We can use these selectors to re-write our earlier programs:

```
;; OfficeHours: staff → symbol
;; Purpose: return the office hours of a given 210 staff member
(define (OfficeHours a-staff)
        (staff-office-hours a-staff))

;; OfficeNumber: staff → symbol
;; Purpose: return the office number of agiven 210 staff member
(define (OfficeNumber a-staff)
        (staff-office-number a-staff))
```

These programs do not completely duplicate the earlier programs. The earlier programs had, embedded inside them, all of the data. Thus, they took a staff member's name and returned the appropriate data. Here, we have elected to make the programs take a "staff" and return the data. This sidesteps (rather inelegantly) the issue of where the staff data resides. You will have to trust me on this one; it will become clear in a couple of lectures.

We can use these 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 sm)
    (cond
        [(symbol=?  (staff-position sm) 'teacher)  true]
        [else                  false]
        ))
```

(in-charge (make-staff 'Keith 'DH2065 'Monday 'teacher)) = true
(in-charge (make-staff 'Tim 'DH2064 'Tuesday 'teaching-assistant)) = false

We will use define-struct to create more interesting examples on Friday.

Add

0. Data Analysis

to the methodology.