COMP 210, FALL 2000
Lecture 5: On Beyond Numbers

Reminders:
- Anyone still looking for a homework partner?
- Homework assignment 0 available on the web site, due next Wednesday
- Read Sections 6 and 7 over the weekend

Review
Last class, we:

1. Introduced natural numbers, wrote down a data definition for natural numbers, and built (factorial n)
2. Used the structure of the data definition to form the structure of a program that operated over data that fit the definition. This adds a couple of new steps to our design methodology—what we will come to call data analysis.
3. Used the structure of the data definition to reason about termination of factorial. This included a subtle point that I'm sure I didn't make. The data definition used **add1** to construct a new natural number from an old one; the code skeleton used **sub1** to discover the number from which N is derived. This will be a common occurrence in our data definitions—two functions: one to construct, and a symmetric & opposite operation to tear it apart (or **destruct** it).

Segue
Our initial attempts at programming in Scheme operated over the real numbers. Last class, we turned to the natural numbers and saw that they have an inherent structure that we can use to guide the construction of programs. This allowed us to create recursive programs to perform computations, such as **factorial** and **fibonacci**, and to reason about why those programs terminate—from a number theoretic basis. (Not just some hand-waving argument!)

Still, using natural numbers is pretty close to using real numbers. We want to explore the whole realm of information, with its richness, its complexity, and its diversity. So, today we are going to look at data that none of us could confuse with numbers, and programs that none of us could confuse for arithmetic.

Symbols
In many situations, we want to compute over richer domains of information than plain, old, well-understood numbers. Another 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 ‘Todd-Waterman

What can we do with a symbol in Scheme? Does (+ ‘Todd 4) make any sense? No. ‘Todd 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 noted that the natural numbers are totally ordered. 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)  ‘Monday-&-Friday-13:00-to-14:30 ]
        [(symbol=? Name  ‘Tim)  ‘Monday-13:30-to-15:00]
        [(symbol=?  Name ‘Todd)  ‘Tuesday-16:00-to-18:00]
        ))
```

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 ‘Todd) ‘DH-2069]
        ))
```

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.

```
;; 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.

Along with constructors, define-struct creates **destructors** 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 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 'Todd 'DH2069 'Tuesday 'teaching-assistant)) = false

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

[Minor corrections from first posting; this version posted Monday, January 31, 2000]