**COMP 210, FALL 2000**
**Lecture 4: Natural Numbers and Recursion**

**Things to do**
- Check the web site for laboratory section assignments
- Homework assignment 0 available on the web site, due next Wednesday

**Review**
Last class, we:

1. Applied the design methodology to build a program Area-of-disk-with-a-hole.
2. Wrote a program to compute (approximately) the graduated income tax on the wages from your work study job (taken on to pay for all that pizza). This exercise introduced the Scheme construct **cond** that selects one of several clauses based on the values assumed by their associated conditions.
3. As a take-home exercise, wrote a program that computed the number of hours of exercise required to work off the extra pizza required by COMP 210 homework.

**Take home Exercise**
The following table specifies how much exercise is required, daily, as a function of the number of slices of pizza that you consume. Write a program WorkOut that computes the number of hours of exercise required to counter the excess fat in pizza (as compared to that staple of undergraduate life, CK food.)

| For a daily intake of | You need to work out for |
|---|---|
| 0 slices | 1/2 hour |
| 1 to 3 slices | 1 hour |
| above 3 slices | 1 hour + 1/2 hour for each slice in excess of 3 |

```
; WorkOut:  num -> num
; Purpose: compute the daily exercise period required after eating S slices of pizza
;              data relating calories to exercise supplied by the instructor
(define  (WorkOut S)
   ( … ) )
```

Next, test data. Analyzing the intervals leads to

```
  0--------1-----------------------3---------------------------------------->
  1/2 hr.   |---------1 hr-----------|------1 hr + (1/2 * (S-3))------------->
```

Suggesting the following as important cases

| | |
|---|---|
| Slices = 0 | (WorkOut 0) => 1/2 |
| Slices = 1 | (WorkOut 1) => 1 |
| Slices = 3 | (WorkOut 3) => 1 |
| Slices = 5 | (WorkOut 5) => 2 |

The program might look like

```
(define (WorkOut  S)
   (cond
      ((=  0  S)   (/ 1 2))
      ((and (< 0 S) (<= S 3))  1)
      ((< 3 S)  (+ 1 (* (- S 3) (/ 1 2))) ) ))
```

Evaluate this program using Dr. Scheme. Use the Stepper to watch the cases get simplified.

**Natural Numbers**

So far, all of our programming examples have involved evaluating simple, flat expressions over numbers—unspecified, general numbers. If this were all that programming involved, we'd be done. For today's class, let's work with a restricted set of numbers that should be familiar to all of us—the "natural numbers."

What are the natural numbers? *(ask class)*

As we move from simple, flat expressions into the more complex domain of *information*, we need to develop a more formal way to specify the domain over which the programs that we write will operate. In COMP 210, we will write a **data definition** that describes the domain.

Writing the data definition will become the first step in our design methodology; I believe that the book refers to this step as **data analysis**.

We can define the natural numbers quite simply. The natural numbers form an infinite set, but one with a specific, rule-based structure. The smallest natural number, or the **base case** (for you fans of mathematical induction), is zero. Zero is a natural number—it is the unique smallest element of the set of natural numbers.. All other natural numbers can be derived from zero by repeated application of **add1**.

**Natural numbers:**

*1.) Zero is a natural number*

2.) *If N is a natural number, then* (add1 N) *is a natural number.*

This data definition is **recursive**—that is, it defines natural numbers in terms of natural numbers. (**n** is a natural number if **n-1** is a natural number, with **zero** as a base case.) Because it has this particular structure, the set of natural numbers is said to be "**recursively enumerable**". (Fancy discrete math term)

Notice that the set of natural numbers is totally ordered—that is, for any pair of distinct natural numbers, *s* and *t*, either (< *s t*) or else (> *s t*). We will use this property to talk about why programs over the natural numbers terminate.

**Programming with the Natural Numbers**

We can use the structure of the natural numbers to organize computations over natural numbers. Consider, for example, the mathematical function **factorial**. For a natural number **n**, (factorial **n**) is defined as

(factorial **n**) is the product of the numbers from 1 to **n**.

Alternatively, (factorial **n**) = 1 * 2 * 3 * … * n-1, * n.

How would we write the program, (factorial **n**) ? In COMP 210, the only allowable answer is that we should follow the design methodology. But what does it tell us about this particular problem? And how does our **data analysis** help in the process?

We have already written the data definition. Now to Contract, Purpose, and Header.

```
;; Factorial: num -> num
;; Purpose: given N, compute N!
(define  (Factorial N)  … )
```

Next, we develop some test cases:

(Factorial 3) => (* 3 (* 2 1)) = 6

(Factorial 5) => (* 5 (* 4 (* 3 (* 2 1))))

(Factorial 1) => 1

(Factorial 0) => ???? what does the definition say ????

The product of 1 to 0. We'll define it to be 1.

That's probably enough test examples (determined from looking at the data definition and the natural language description of the program). How do we fill in the ellipsis in the code body?

```
(define (Factorial N)  … )
```

Not surprisingly, the key lies in the data definition.

**Natural numbers:**

1. *0 is a natural number*
2. *If N is a natural number, then* (add1 N) *is a natural number.*

This suggests an organization with two cases. On real numbers, we talked about performing a data analysis based on the notion of open and closed intervals to work out the cases of a **cond** construct. On the natural numbers, we perform an analogous case analysis. Based on the data definition, we get two cases—one for zero and one for the rest of the natural numbers.

```
(define (Factorial N)
   (cond
      ((= 0 N) … )
      ((> 0 N) … ) ))
```

Finally, we fill in the expressions controlled by the cases in the **cond** expressions.

The first case is easy—for N = 0, the program should return 1.

The second case is more complex—for N > 0, the program should multiply N times the product from (N-1) to 1. We cannot write out the code for each value of N. We can, however, notice that the product from (N-1) to 1 is simply (Factorial (- N 1)). This leads to the following code:

```
(define (Factorial N)
   (cond
      ((= 0 N)     1)
      ((< 0 N)  (* N  (Factorial (- N 1)))) ))
```

This self-referential call—invoking Factorial from inside itself—is called a **recursive** call, or a **recursion**. Before you can write a **recursive** call, you must convince yourself that the recursion will (eventually) halt—or **terminate**. This property, termination, is one of the most important properties that a program can have. (Almost all programs are designed to terminate. One major exception is an operating system. Unfortunately, all of them that have been written to date have terminated one or more times.)

Why does **Factorial** terminate? *(ask class)*

**Sketch of Proof:**
Go back to the data definition. If we invoke **Factorial** with an argument N that is a Natural Number, two cases are possible. Either N is zero, in which case Factorial does **not** call itself, or N > 0, in which case we know that N can be derived from zero by repeated application of **add1**. In this latter case, we know that repeated application of **sub1** will get us back to zero from <u>any</u> natural number. Each time **Factorial** recurses, it subtracts one from N. Thus, it must, eventually, invoke **Factorial** with an argument of zero, halting the recursion.

**Next step in the methodology is to test the code**.

**USE Dr. Scheme, File "Lecture 3". Step through several examples.**

This suggests an addition to the design methodology—we do a case analysis on the data definition, and turn that into a set of cases for a **cond** construct in the program's body.

**Final Note**
We can simplify the Scheme code a little by using two special functions

(zero? N) is equivalent to (= 0 N)

else is a keyword that can only be used in the last position in a **cond** construct. It behaves as if it evaluates to **true**.

```
(define (Factorial N)
   (cond
      ((zero? N)     1)
      (else (* N  (Factorial (- N 1)))) ))
```

You should use predicates like **zero?** whenever possible because they improve the readability of your code.

**Next Class**
Symbols—Computing Beyond Numbers