COMP 210, Spring 2001
Lecture 8: Recursion and the Natural Numbers

Reminders:

- Homework due Wednesday in class
- Exam will be 2/23/2001, in class, closed-notes, closed-book

Review

1. Talked about lists with mixed data.

Clarifications

1. Write helper functions that solve the subproblems that arise in decomposing a problem. Helper functions are mandated whenever more than one instance of the subproblem arises in the decomposition. Examples: the mph-fps conversion in **`houston-driver-distance`** the squaring operation in **`point3-add.`**
2. Eliminate all "magic numbers" from your programs by defining variables (with meaningful names) equal to the magic numbers. Examples: feet-per-mile and seconds-per-hour in **`houston-driver-distance.`**
3. Despite claims in the book to the contrary, the form of a program template is not solely dependent on the data definition. In particular, the template must identify the primary argument of the program, which cannot be determined without inspecting the program contract, header, and purpose. Example:

```
; append: list-of-Plane list-of-Plane -> list-of-Plane
; Purpose: given l1 = (a1 … am) and l2 = (b1 … bn), returns
;          the list (a1 … am b1 … bn)
; (define (append lop1 lop2) …)
; Examples
; …
; Data-only Template
; (define (f … lop … )
;   (cond [(empty? lop) … ]
;         [(cons? lop) … (first lop) … (f … (rest lop) … )]))
;
; Correct Template
; (define (append lop1 lop2 )
;   (cond [(empty? lop1) … ]
;         [(cons? lop1) …(first lop1)
;                       … (append (rest lop1) … )]))

(define (append lop1 lop2 )
  (cond [(empty? lop1) lop2]
        [(cons? lop1) (cons (first lop1)
                            (append (rest lop1) lop2))]))
```

4. In addition, programs involving numbers like **houston-driver-distance** often rely on templates derived from careful data analysis which goes beyond determining the data definitions (numbers are already built-in to Scheme).

An Issue of Taste

Programming is part science (the COMP 210 part) and part art (the part built on taste, experience, and all those other "soft" terms). Consider our list-of-nums-and-syms.

We wrote it as

```
;; a list-of-nums-and-syms is one of
;; - empty, or
;; - (cons s lons)
;;  where s is a symbol and lons is a list-of-nums-and-syms, or
;; - (cons n lons)
;; where n is a number and lons is a list-of-nums-and-syms
```

We could also have written it as

```
;; a NumSym is either
;;    - a number, or
;;    - a symbol

;; a list-of-NumSyms is either
;;    - empty, or
;;    - (cons f r)
;; where f is a NumSym and r is a list-of-NumSyms
```

Another example:

```
;; a list-of-toppings is one of
;; - empty, or
;; - (cons 'cheese a-lot), where a-lot is a list-of-toppings, or
;; - (cons 'pepperoni a-lot), where a-lot is a list-of-toppings,
;; or
;; - (cons 'spinach a-lot), where a-lot is a list-of-toppings.
```

Versus

```
;; a topping is one of
;;    - 'cheese, or
;;    - 'pepperoni, or
;;    - 'spinach

;; a list-of-toppings is one of
;;    - empty, or
;;    - (cons f r)
;;  where f is a topping and r is a list-of-toppings
```

Which of these data definitions is preferred? This is a matter of taste, experience–-in short, what Knuth called "The Art of Computer Programming." As you write more programs, larger programs, programs that are used by other people, and, finally, programs that are modified by other people, you will develop insight into this issue. [In fact, programmers with good taste can disagree over such fundamental issues.]

Natural Numbers

Recently, we've been working with lists and writing programs that recursively traverse various kinds of lists. Is this the only use for recursion? (Obviously, the answer is **no!)** If this were all that programming involved, we'd be done. For today's class, we will 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)*

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**, just as the data definition for a list is recursive. This structure should remind you of an induction proof from some high school math class. (**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)** ? We need a template for natural numbers. (Of course, some programs over natural numbers will be simple, flat expressions where the template is serious overkill. However, interesting programs over the natural numbers will have a template drawn from the data definition.

```
;; 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.

;; a **natural number** is either
;; – **0**, *or*
;; **-** **(add1 n)**, where **n** is *a natural number*

This suggests a template with two cases. Back when we wrote a program or two with real numbers (such as the **workout** program in Pizza Economics), we use the mathematical notions of open and closed intervals to model the cases that occurred in the problem statement. On the natural numbers, we have a more restricted structure, so the data definition actually produces a template.

```
(define (f a-natnum)
  (cond
    [(= 0 a-natnum) … ]
    [(> 0 a-natnum)  (f (something? a-natnum))… ] ))
```

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 (sub1 N)))]))
```

Changed name from template for historical reasons

Notice that we filled in *something* with **sub1**. Was that insight? Was it disciplinary knowledge (based on my special understanding of natural numbers)? Or was it analogous to the way we built templates for lists and compound objects?

With lists, the data definition uses the list constructor, **cons**, and the template uses the list selectors **first** and **rest**. With compound objects, the data defintion uses the constructor, such as **make-Plane**, and the template uses the selectors, such as **Plane-tail-num** and so on. With natural numbers, the data definition uses **add1**. What undoes an **add1**? A **sub1** does.

This is a new use for a recurrence–-to perform a computation rather than to traverse some concrete structure. [To the extent that you can call a Scheme list "concrete."] It brings up, in a particularly clear way, an issue that we have sloughed off to this point. *How do we know that the recursion will halt – or **terminate**?* Before you can write any **recursive** call, you should convince yourself that the recursion will terminate. This is one of the most important properties that a program can have. (Almost all programs are designed to terminate. Major exceptions include a clock program and 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 any natural number. Each time **factorial** recurs, it subtracts one from **n**.

Thus, it must, eventually, invoke **`factorial`** with an argument of zero, halting the recursion.

This argument can be formalized as a proof by mathematical induction.

Next step in the methodology is to test the code.

Final Note

We can simplify the Scheme code a little by using the special function

`(zero? n)` is equivalent to **`(= 0 n)`**

```
(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.