COMP 200: Elements of Computer Science
Fall 2004
Lecture 14: September 27, 2004

Recursion over the Counting Numbers

On the Board

Homework 3: Due today
First test: Due Wednesday at 5pm

Back to Programming

As I stated last week, we will devote no more than one class in three to programming – that is, to talking about programs, methodologies, etc. We will, of course, talk about algorithmics the rest of the time. To the extent that algorithmics is related to programming, you may believe that I have deceived you. However, a quick tour of COMP 210's web site will convince you that I am not turning this course into COMP 210.

So far, we have learned to write simple algebraic programs, to work with structures, and to traverse lists. Our list-based programming introduced the notion of structural recursion – a programming style that echoes the structure of the list definition and has the program call itself to process the self-referential links in a list. Our structural recursions halt because the lists are finite in length and the body of the code checks explicitly for the ending case *(**empty?**)* before handling the non-empty (self-reference through ***rest*** access function) case.

Today, we will look at programs that use recursion without an explicit list.

Recursion on the Counting Numbers

For the purpose of today's lecture, we will define the counting numbers as the unbounded set {1, 2, 3, 4, 5, …}. Sometimes, in making arguments about recursion, we will find it convenient to start this set at zero, which is technically not a counting number. (A set with zero elements cannot be counted, since it is empty.) For today, one will suffice as a starting point.

As with all programming in COMP 200, we start out with a data definition. This approach may seem silly for a concept that is as familiar as the counting numbers, but it will prove useful, so please bear with me.

```
; Data analysis for the Counting Numbers
;  a counting number is either
;    - one, or
;    - if N is a counting number, then (add1 N) is a counting number
;  we can use Scheme's builtin implementation of numbers
```

This definition shows that the counting numbers have a recursive structure that is similar to that of a list. The data definition has two cases, one for the basis case – one – and another for the recursive case – (add1 N). [With lists, we had a basis case of empty and a recursive case of (cons first rest).]

The next step in our methodology, of course, is to build a template.

```
; template for the counting numbers

(define (ACountingProg fee)
        (cond
           ((= 1  fee) … )
           ((< 1  fee)   … (f  (sub1  fee)) … )
        ))
```

What is this thing?
Where did we find it?

In the template for list, the data definition shows the non-empty case to be a (***cons first rest***) and the template recurs on ***rest***. ***Rest*** is the inverse of ***cons*** and ***sub1*** is the inverse of ***add1***. If ***cons*** constructs a list, ***rest*** deconstructs it (somewhat different notion from *destruct*). Similarly, if ***add1*** creates a new counting number from one named N, then ***sub1*** takes a counting number greater than one and deconstructs it to discover the counting number from which it was derived.   Whew.

Let's write a program

Compute the sum of the counting numbers from 1 to N.

Contract, purpose, & header

```
; Sum: counting number → counting number
; Purpose: Sum takes as its input a counting number N and returns
;              the sum of the counting numbers from 1 to N
(define (Sum N) …)
```

Test Data

Input	**Result**
0	Undefined (not a counting number)
1	1
2	3
10	55
100	5050

Write the expression and fill in the program body

```
; Sum: counting number → counting number
; Purpose: Sum takes as its input a counting number N and returns
;             the sum of the counting numbers from 1 to N
(define (Sum N)
        (cond
           ((= 1  N)     1 )
           ((< 1  N)   (+ N  (Sum  (sub1  N))) )
        ))
```

Code it up and run it in Dr. Scheme

Why does this program halt?

→ Contract requires that N be a counting number; any other input has undefined behavior (such as 1.5 3.14159, 0, -10)

→ Two cases on counting numbers (to reflect the data definition)

- N = 1 returns one (and halts)
- N > 1
  - Since N is a counting number, we know that it can be derived from 1 by repeated application of ***add1***
  - Thus, repeated calls to ***sub1*** must eventually produce a 1
  - Code always recurs on (***sub1 N***), so it must reach the case that N = 1 and, at that point, halt

→ Key issue is the structural recursion inherent in the definition of the counting numbers.

Can we do it more simply?

The methodology produces a correct solution. That does not say that it always produces the best solution. A simpler program can be built on the algebraic knowledge that the sum of the counting numbers from 1 to N is always given by the formula

$$N (N+1) / 2$$

A program built on that insight will be shorter, quicker, and much harder to explain – unless you understand the summation.

A Second Program On Counting Numbers

What about a program that computes N factorial

```
; Fact : counting number → counting number
; Purpose: given a counting number N, compute N!
(define (Fact N) …)
```

Test data

Input	Result
1	1
2	2
3	6
10	3,628,800

Write the expression for the program body, informed by the test data

This program is the first case that we have seen where the test data does not point out an obvious solution. We need to know the formula for N!

N! is defined as N * (N-1)!

That definition actually looks like something we can use.

```
; Fact : counting number → counting number
; Purpose: given a counting number N, compute N!
(define (Fact N)
          (cond
            ((= 1  N)     1 )
            ((< 1  N)   (* N  (Fact (sub1  N))) )
          ))
```