CS3:
Introduction to Symbolic Programming

Lecture 7:
The last of recursion (for a while)

Spring 2008

Nate Titterton
nate@berkeley.edu

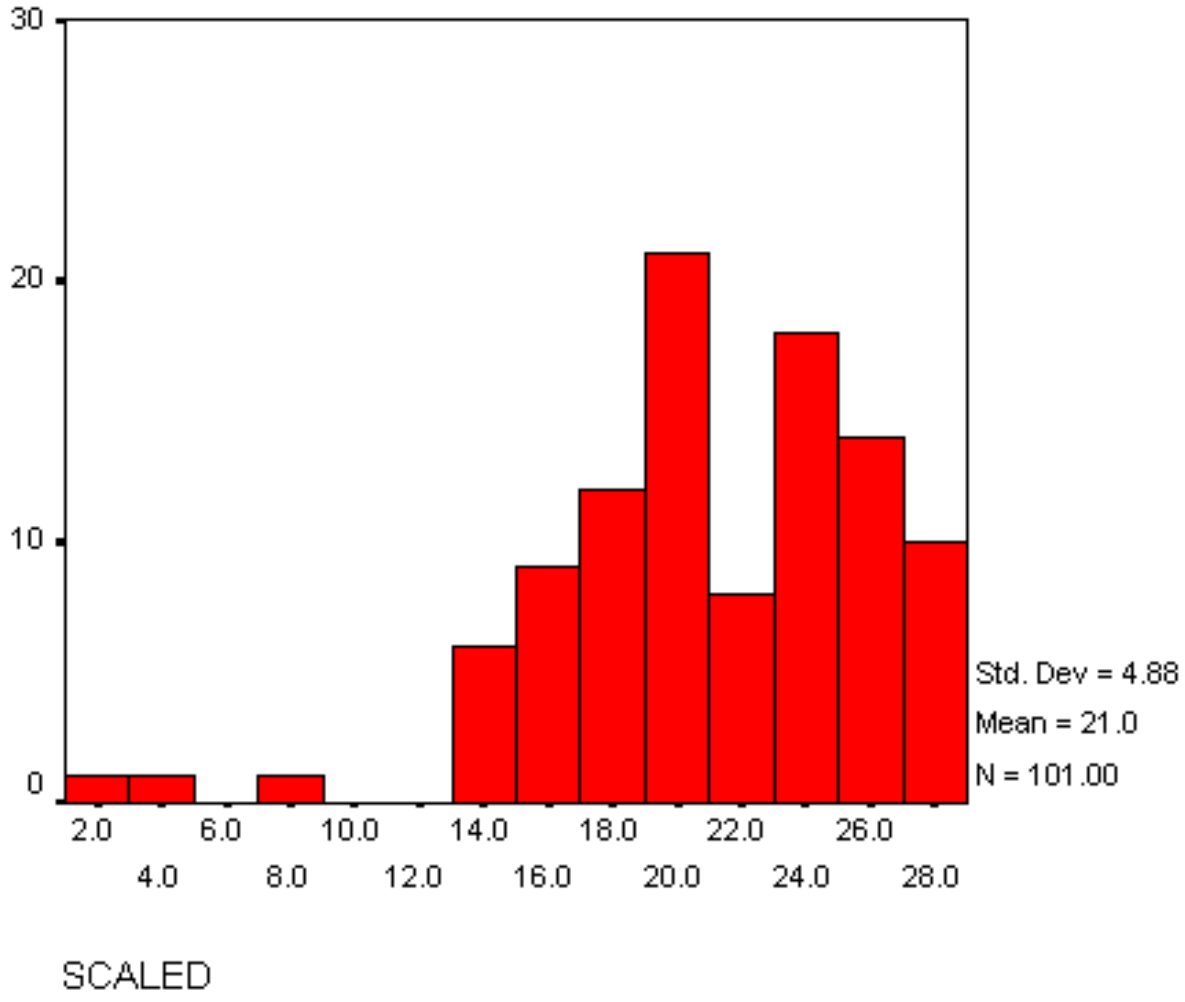
Schedule

6	Feb 25-29	Lecture: *Midterm #1* Lab: Recursion with multiple arguments Homework: The “big” homeworks…
7	Mar 3-7	Lecture: Advanced Recursion Lab: Advanced Recursion Miniproject #2: Number Spelling
8	Mar 10-14	Lecture: Higher Order Functions Lab: Introduction to HOF Reading: Simply Scheme, Ch 9, 10 "DbD" HOF version Note: MP#2 due Tue/Wed
9	Mar 17-21	Lecture: Advanced HOFs Lab: Advanced HOF, tic-tac-toe
10	Mar 24-28	*Spring Break!*

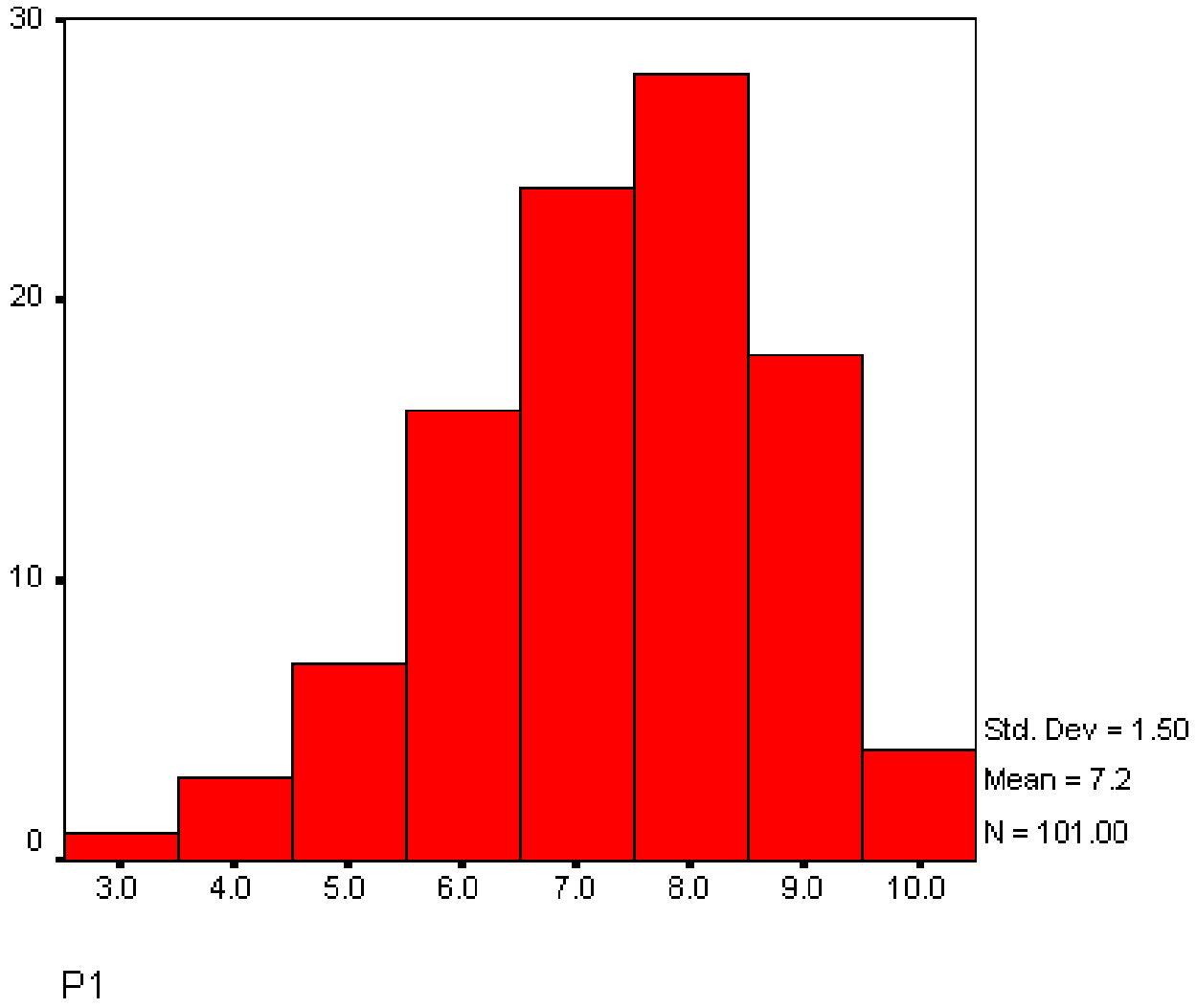
Announcements

- **Nate's office hours:**
  - Wed, 10-12 this week only

Midterm 1

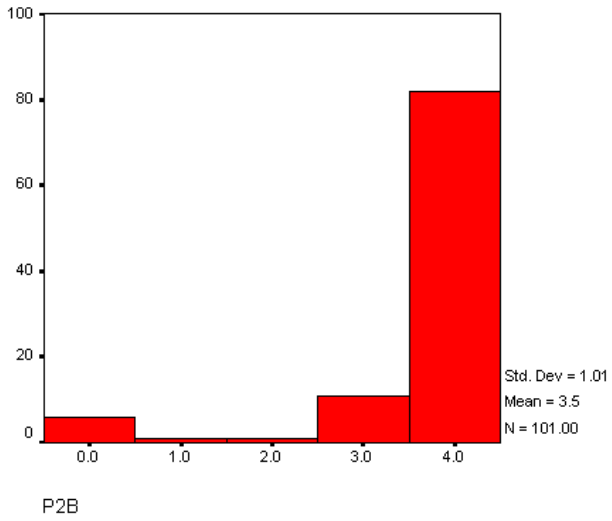

30
20
10
0
2.0
4.0
6.0
8.0
10.0
12.0
14.0
16.0
18.0
20.0
22.0
24.0
26.0
28.0
Std. Dev = 4.88
Mean = 21.0
N = 101.00
SCALED

Question 1: fill in the blanks

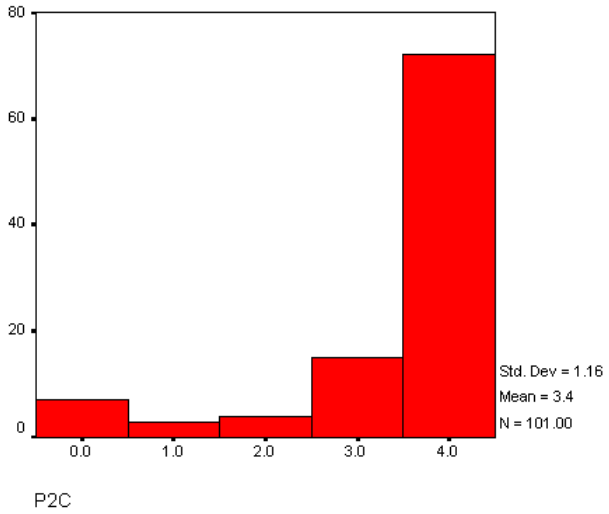


Question 2: When in Rome

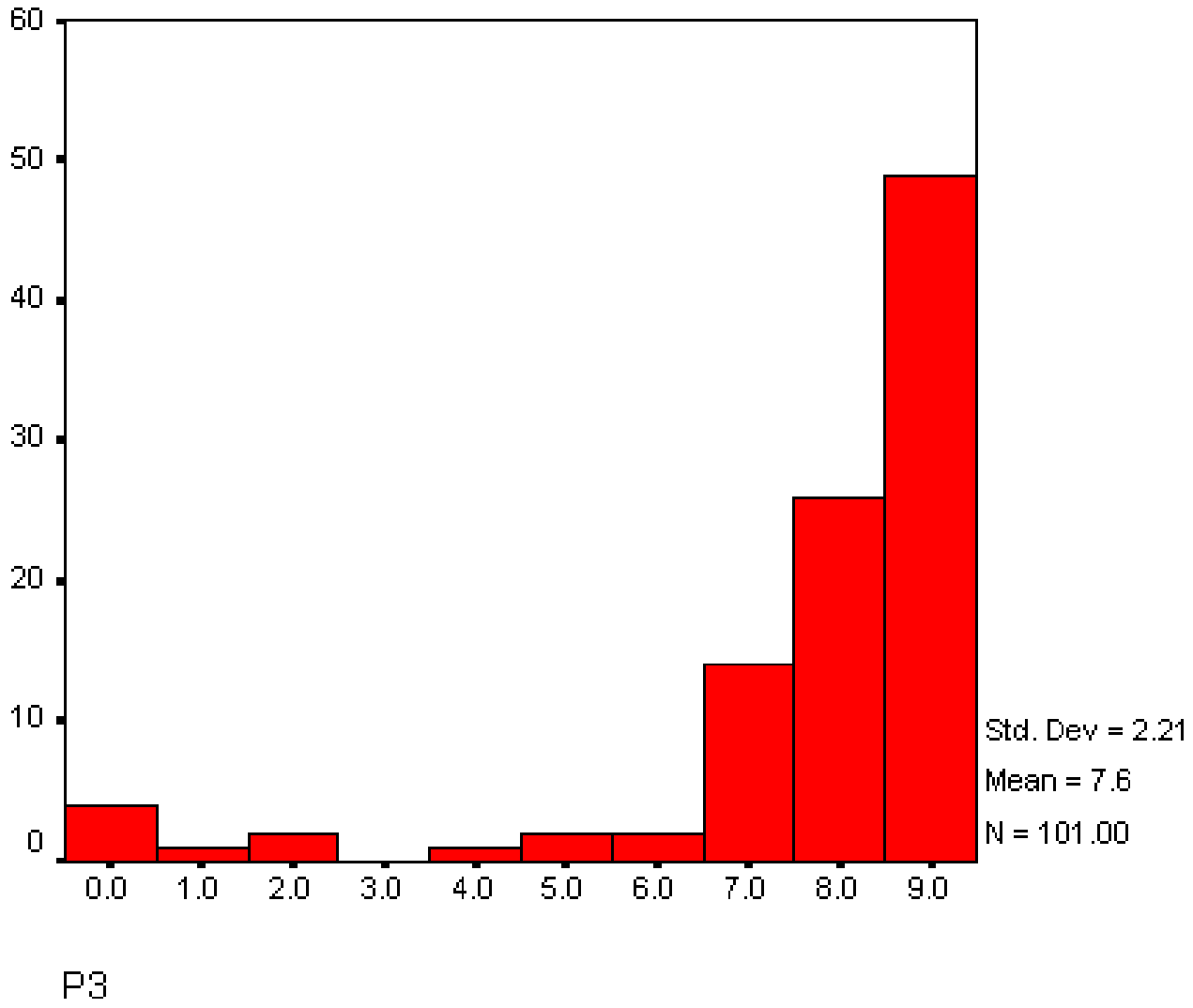
2b: decimal-values



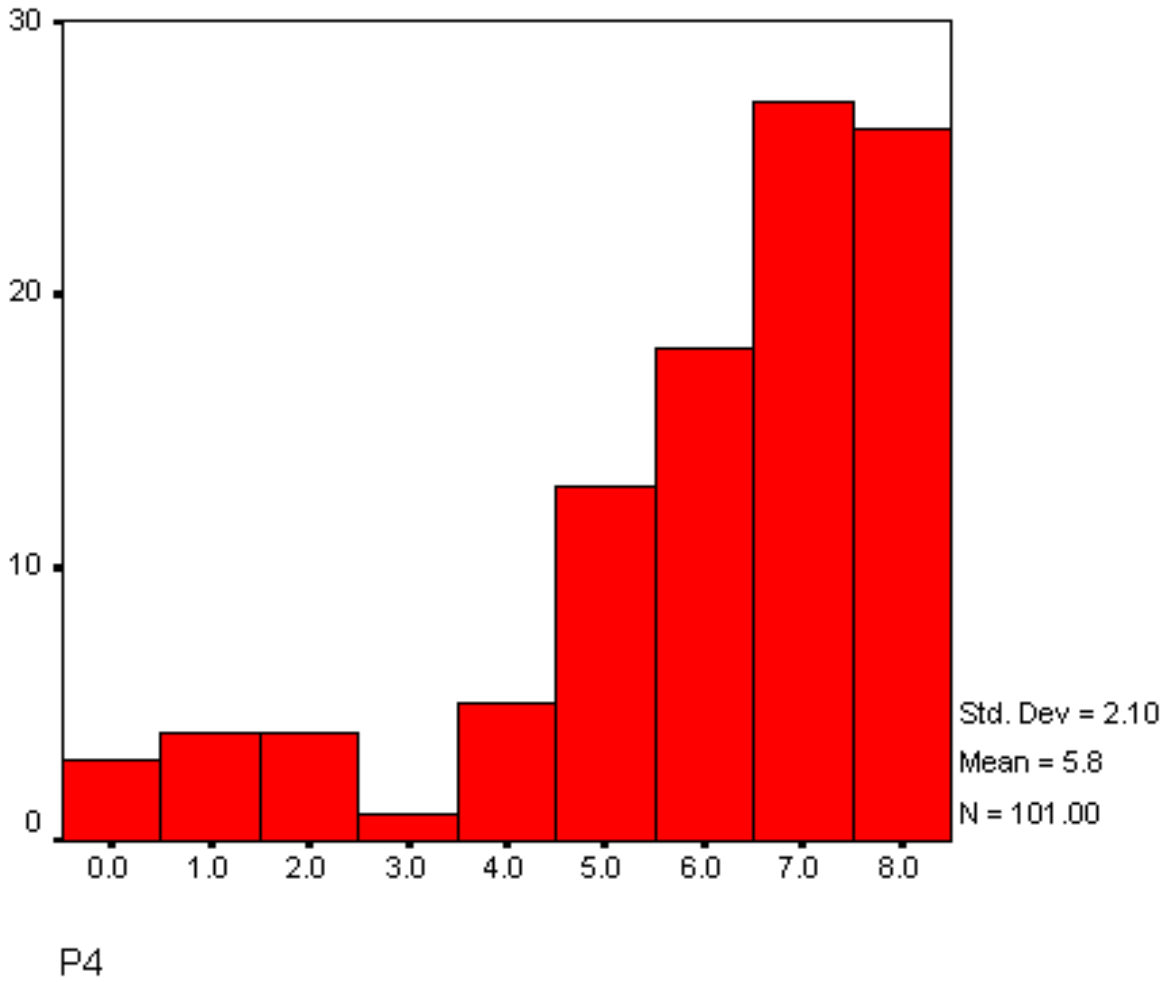
2c: valid-prefix?



Q3: `look-at` and `pluralized`

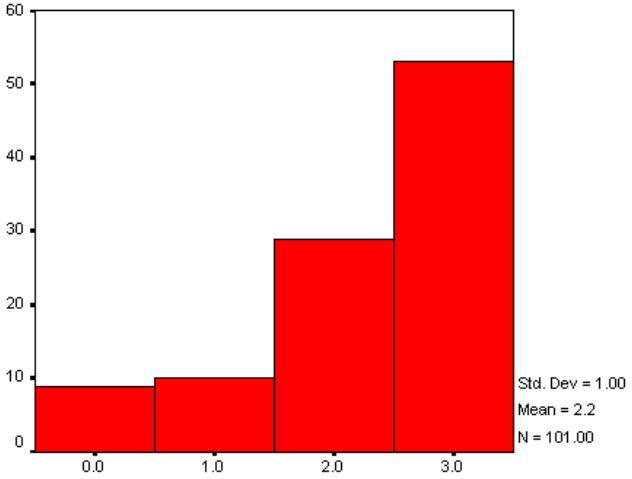


Q4: nines

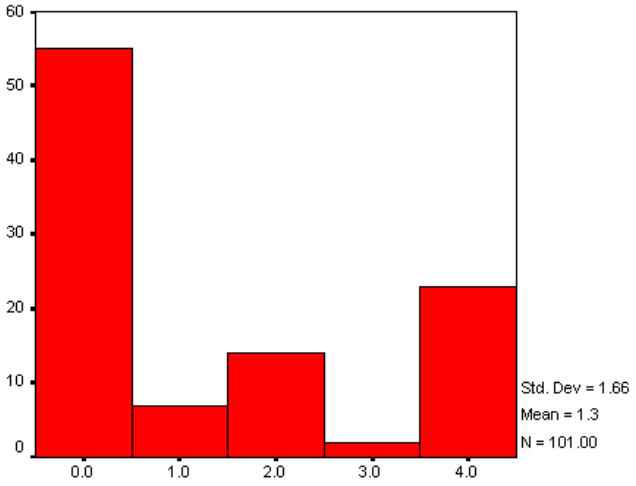


Q5: bring-element-to-front

5c:
buggy betf



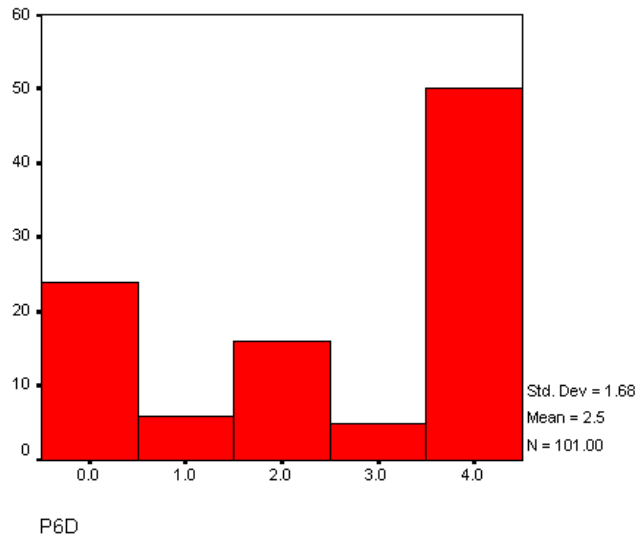
5d:
Recursive betf



P5D

Q6: mystery

6d:
mystery
without and or or



Goodbye recursion?

- Nope. We'll do more with recursion later

- What have we done in the last few weeks?
  - pairwise-recursions
    - roman numerals, `is-sorted?`
  - "Advanced recursions": ones that work on multiple sentences, or do more than one thing at a time
    - `zip`, `merge`, `my-equal?`
    - `mad-libs` takes a sentence to mutate `'(I saw a * horse named * with * legs)` and replacement words `'(fat Henry three)`
  - Recursive patterns (map, filter, etc)
  - Sorting (insertion sort)
  - Accumulating recursion (e.g., using `so-far`)

Patterns in basic recursion (1-3 of 6)

- **Mapping**
  - does something to every part of the input sentence
  - E.g., square-all
- **Counting**
  - Counts the number of elements that satisfy a predicate
  - E.g., count-vowels, count-evens
- **Finding**
  - Return the first element that satisfies predicate (or, return rest of sentence)
  - E.g., member, member-even

Patterns in basic recursion (4-6 of 6)

- **Filtering**
  - Keep or discard elements of input sentence
  - E.g., `keep-evens`

- **Testing**
  - A predicate that checks that *every* or *any* element of input satistfies a test
  - E.g., `all-even?`

- **Combining**
  - Combines the elements in some way…
  - E.g., `sentence-sum`

`roman-sum-helper` (from lab)

Write roman-sum-helper:

```
(define (roman-sum number-sent)
 (if (empty? number-sent)
     0
     (roman-sum-helper (first number-sent)
                       (bf number-sent)
                       (first number-sent)) ) )
```

Roman-sum-helper takes three arguments:

```
(define (roman-sum-helper so-far number-list most-
 recent) ... )
```

`(roman-sum '(100 10 50 1 5))` **will recurse with:**

```
(roman-sum-helper 100 '(10 50 1 5) 100)
(roman-sum-helper 110 '(50 1 5) 10)
(roman-sum-helper 140 '(1 5) 50)
(roman-sum-helper 141 '(5) 1)
(roman-sum-helper 144 '( ) 5)
```

Accumulating or “tail” recursions

- **Accumulating recursions are sometimes called "tail" recursions (by TAs, me, etc).**
  - **But, not all recursions that keep track of a number are "tail" recursions.**

Tail versus “embedded” recursions

- A <u>tail</u> recursion has no combiner, so it can end as soon as a base case is reached
  - Compilers can do this efficiently
- An <u>embedded</u> recursion needs to combine up all the recursive steps to form the answer
  - The poor compiler has to keep track everything

Tail or embedded? (1/3)

```
(define (my-count sent)
   (if (empty? sent)
      0
      (+ 1 (my-count (bf sent)))))
```

Embedded!

```
(my-count '(a b c d)) →
  (+ 1 (my-count '(b c d)))
  (+ 1 (+ 1 (my-count '(c d))))
  (+ 1 (+ 1 (+ 1 (my-count '(d)))))
  (+ 1 (+ 1 (+ 1 (+ 1 (my-count '()))))))
  (+ 1 (+ 1 (+ 1 (+ 1 0))))
  (+ 1 (+ 1 (+ 1 1)))
  (+ 1 (+ 1 2))
  (+ 1 3)
  4
```

Tail or embedded? (2/3)

```
(define (find-evens sent)
  (cond ((empty? sent)  '() )
        ((odd? (first sent))
         (find-evens (bf sent)) )
        (else
         (se (first sent)
             (find-evens (bf sent))) ) ))
```

```
➢ (find-evens '(2 3 4 5 6 7))
  (se 2 (find-evens '(3 4 5 6 7)))
  (se 2 (find-evens '(4 5 6 7)))
  (se 2 (se 4 (find-evens '(5 6 7))))
  (se 2 (se 4 (find-evens '(6 7))))
  (se 2 (se 4 (se 6 (find-evens '(7)))))
  (se 2 (se 4 (se 6 (find-evens '()))))
  (se 2 (se 4 (se 6 '())))
  (2 4 6)
```

Coming up…

- **Work on "buggy" recursions**

- **Two-stage recursions**
  - Where a recursive procedure calls another recursive procedure each step
  - (You have done things like this without knowing about it: e.g., `remove-dupls`)
  - Most often, when doing something to each word in a sentence.
    - You saw this (briefly) in `no-vowels`
    - Also `(l33t '(I like to type))`
      → `(i 1i/<3 +0 +yP3)`

Number Spelling (Miniproject #2)

- **A program to write out names of almost any number**
  - You can work in a partnership (if you want)
  - Read *Simply Scheme*, page 233, which has hints
- **You will be using a new testing library**
- **Another hint (principle): don't force "everything" into the recursion.**
  - Special/border cases may be easier to handle before you send yourself into a recursion

Any other questions?