CS3:
Introduction to Symbolic Programming

Lecture 8:
Midterm 1, Last bit of recursion,
Higher order functions

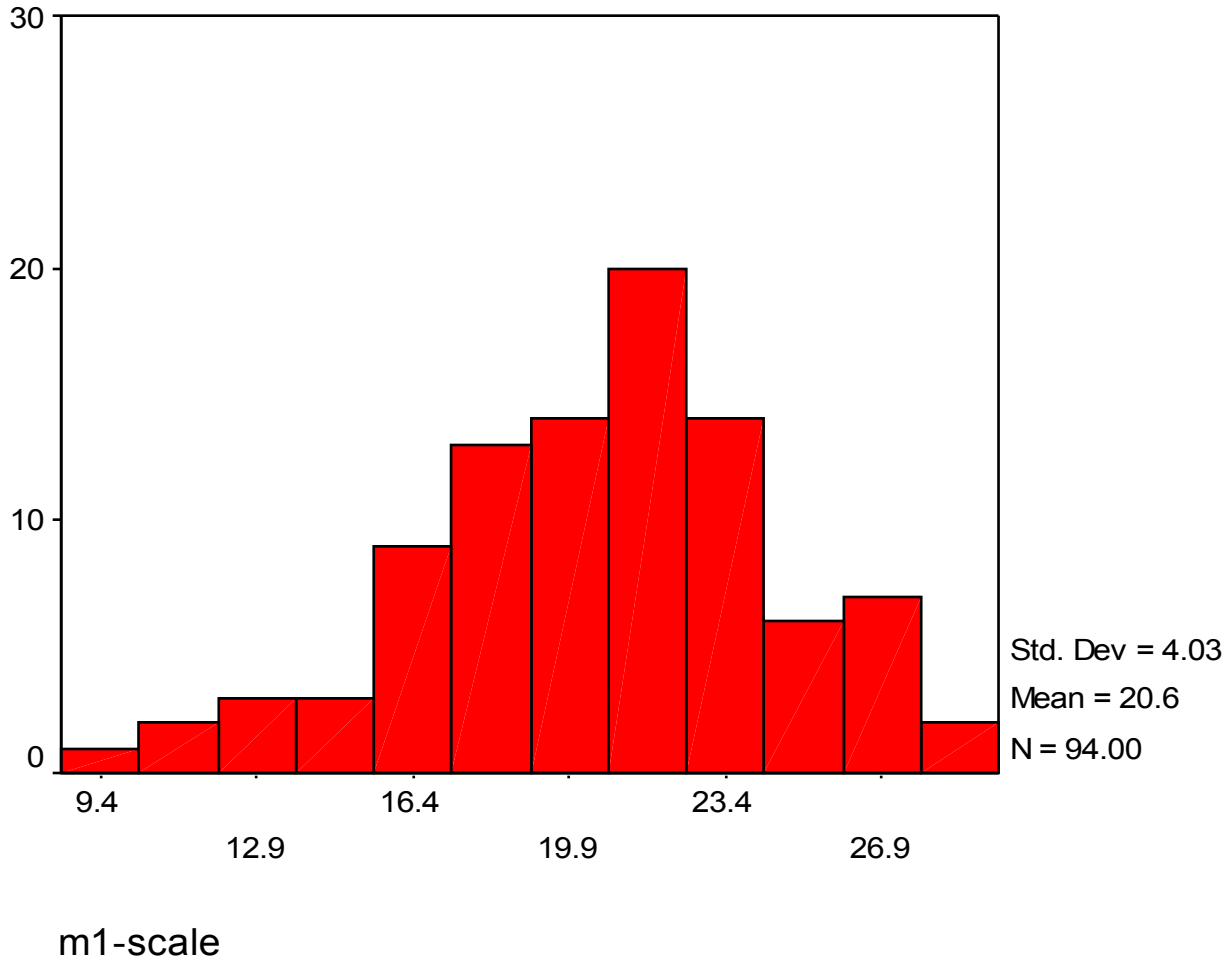
Spring 2007

Nate Titterton
nate@berkeley.edu

Schedule

8	Mar 5 – 9	Lecture: Today Reading: Simply Scheme, ch 7-8 Lab (T/W): Miniproject #2 Second Survey Lab (T/F): Begin higher order functions
9	Mar 12-16	Lecture: Higher order functions Reading: Simply Scheme, Ch 9, 10 "DbD" HOF version Lab: Higher order functions, tic-tac-toe Note: Miniproject #2 is due (Tue/Wed)
10	Mar 19-23	Lab: More Higher order functions… Start on Miniproject #3?
11	Mar 26-30	*Spring Break*

Midterm #1: overall



Midterm #1: questions

1. `can-order?`, write it and test it
   - You did quite well here
2. Booleans (`not-in-order?`, etc.)
3. `general-day-span-r`
   - 4b was the hard one!
4. `scramble`

Announcements

- If you have any questions or comments on your midterm, please see me or your TA.
  - Nate's office hours: 329 Soda, Wed 2-4
- The Tue/Thur 2-5 section is losing Alex, and getting Bobak.

Number Spelling (Miniproject #2)

- **A program to write out names of almost any number**
- **Read *Simply Scheme*, page 233, which has hints**
- **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**

Goodbye recursion?

- **Nope. We'll do more with recursion later**

- **What have we done in the last few weeks?**
  - **Work with roman numerals**
  - **"Advanced recursions": ones that work on multiple sentences, or do more than one thing at a time**
  - **`zip`, `merge`, `my-equal?`, `1-extra?`**
  - **Recursive patterns (map, filter, etc)**
  - **Sorting (insertion sort)**
  - **Accumulating recursion (e.g., using `so-far`)**
  - **Two-stage recursion (inner/outer)**
  - **and more**

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 versus "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.**

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

Tail or embedded? (1/2)

```
(define (length sent)
   (if (empty? sent)
      0
      (+ 1 (length (bf sent)))))
```

Embedded!

```
(length '(a b c d)) →
  (+ 1 (length '(b c d)))
  (+ 1 (+ 1 (length '(c d))))
  (+ 1 (+ 1 (+ 1 (length '(d)))))
  (+ 1 (+ 1 (+ 1 (+ 1 (length '())))))
  (+ 1 (+ 1 (+ 1 (+ 1 0))))
  (+ 1 (+ 1 (+ 1 1)))
  (+ 1 (+ 1 2))
  (+ 1 3)
  4
```

Tail or embedded? (2/2)

```
(define (find-evens sent)
  (cond ((empty? sent)          ;base case
         '()                )
        ((odd? (first sent)) ;rec case 1
         (find-evens (bf sent)) )
        (else                       ;rec case 2: even
         (se (first sent)
             (find-evens (bf sent))) )
        ))
```

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

  (se 2 (se 4 (se 6 '())))
  (2 4 6)
```

Higher Order Functions

What is a procedure?

(or, a *function*).

Treating functions as things

- "define" associates a name with a value
  - The usual form associates a name with a object that is a function

```
(define (square x) (* x x))
(define (pi) 3.1415926535)
```

  - You can define other objects, though:

```
(define *pi* 3.1415926535)
(define *month-names*
   '(january february march april may
     june july august september
     october november december))
```

"Global variables"

- **Functions are "global", in that they can be used anywhere:**

```
(define (pi) 3.1415926535)
(circle-area (radius)
      (* (pi) radius radius))
```

- **A "global" variable, similarly, can be used anywhere:**

```
(define *pi* 3.1415926535)
(circle-area (radius)
      (*  *pi*  radius radius))
```

Are these the same?

Consider two forms of “month-name”:

```
(define (month-name1 date)
    (first date))
```

```
(define month-name2 first)
```

Procedures can be taken as arguments…

```
(define (math-function? func)
  (or (equal? func +)
      (equal? func -)
      (equal? func *)
      (equal? func /)))
```

…and procedures can be returned from procedures

```
(define (choose-func name)
   (cond ((equal? name 'plus) +)
         ((equal? name 'minus) -)
         ((equal? name 'divide) /)
         (else 'sorry)))


(define (make-add-to number)
   (lambda (x) (+ number x)))

(define joe (make-add-to 5))
```

Higher order function (HOFs)

- **A HOF is a function that takes a function as an argument.**

```
(define (do-math f arg1 arg2)
    (if (and (equal? arg2 0)
             (equal? f /))
        '(uh oh - divide by zero)
        (f arg1 arg2)))
```

The three we will focus on

- **There are three main ones that work with words and sentences:**

`every`	do something to each element
`keep`	return only certain elements
`accumulate`	combine the elements

Patterns for simple recursions

- **Most recursive functions that operate on a sentence fall into:**

Mapping:	`square-all`	`<- every`
Counting:	`count-vowels`, `count-evens`	
Finding:	`member`, `first-even`	
Filtering:	`keep-evens`	`<- keep`
Testing:	`all-even?`	
Combining:	`sum-evens`	`<- accumulate`