CS3:
Introduction to Symbolic Programming

Lecture 11:
Midterm #2 review

Spring 2006

Nate Titterton
nate@berkeley.edu

Schedule

10	Mar 20-24	More HOF, Tic-Tac-Toe, Tree Recursion Reading: SS 10, 15; "Change Making" case study
11	Mar 27-31	*(Spring Break)*
12	Apr 3-7	Lecture: Review Lab: Miniproject #3
13	Apr 10-14	Lecture: MIDTERM #2 Lab: Start on "Lists"
14	Apr 17-21	Lecture: Lists, and introduce the big project Lab: Lists; start on the project
15	Apr 24-28	Lecture: Lists, and ? Lab: Work on the project

Announcements

- **Midterm 2 is coming…**
  - Next week, 80 minutes (4:10-5:30).
  - Open book, open notes, etc.
  - Check for practice exams and solution on the course portal and in the reader
- **Midterm 2 review session**
  - This Saturday, Apr 8, 1:30-3:30
  - 430 Soda (as last time)
  - send email to Bobak or Andrew for suggestions.
- **Fu and Hiroki (bless their hearts) will be holding an extra lab/office-hours**
  - This Wednesday, April 5, 5:30 to 8pm
  - in the lab room

What does midterm #2 cover?

- Advanced recursion (accumulating, multiple arguments, etc.)
- All of higher order functions
- Those "big" homeworks (bowling, compress, and occurs-in)
- Elections miniproject
- Reading and programs:
  - Change making,
  - Difference between dates #3 (HOF),
  - tic-tac-toe
- SS chapters 14, 15, 7, 8, 9, 10
- Everything before the first Midterm (although, this won't be the focus of a question)

Programming Style and Grading

- During grading, we are going to start becoming “more strict” on style issues
  - Starting with miniproject #3
  - For the big project, style is important

- Why?
  - Program maintenance: 6 months later, will you know what your code does?
  - Code “literacy”: sharing code

What issues of style matter?

- **Keep procedures small !**
- **Good names for procedures and parameters**
- **Adequate comments**
  - **Above and *within* procedures**
- **Put tests cases in a comment block**
- **Indent to aid program comprehension**

- **Proper use of global variables**
- **Avoid nesting conditional statements**
- **Data abstraction**

Tree recursion

Advanced recursion

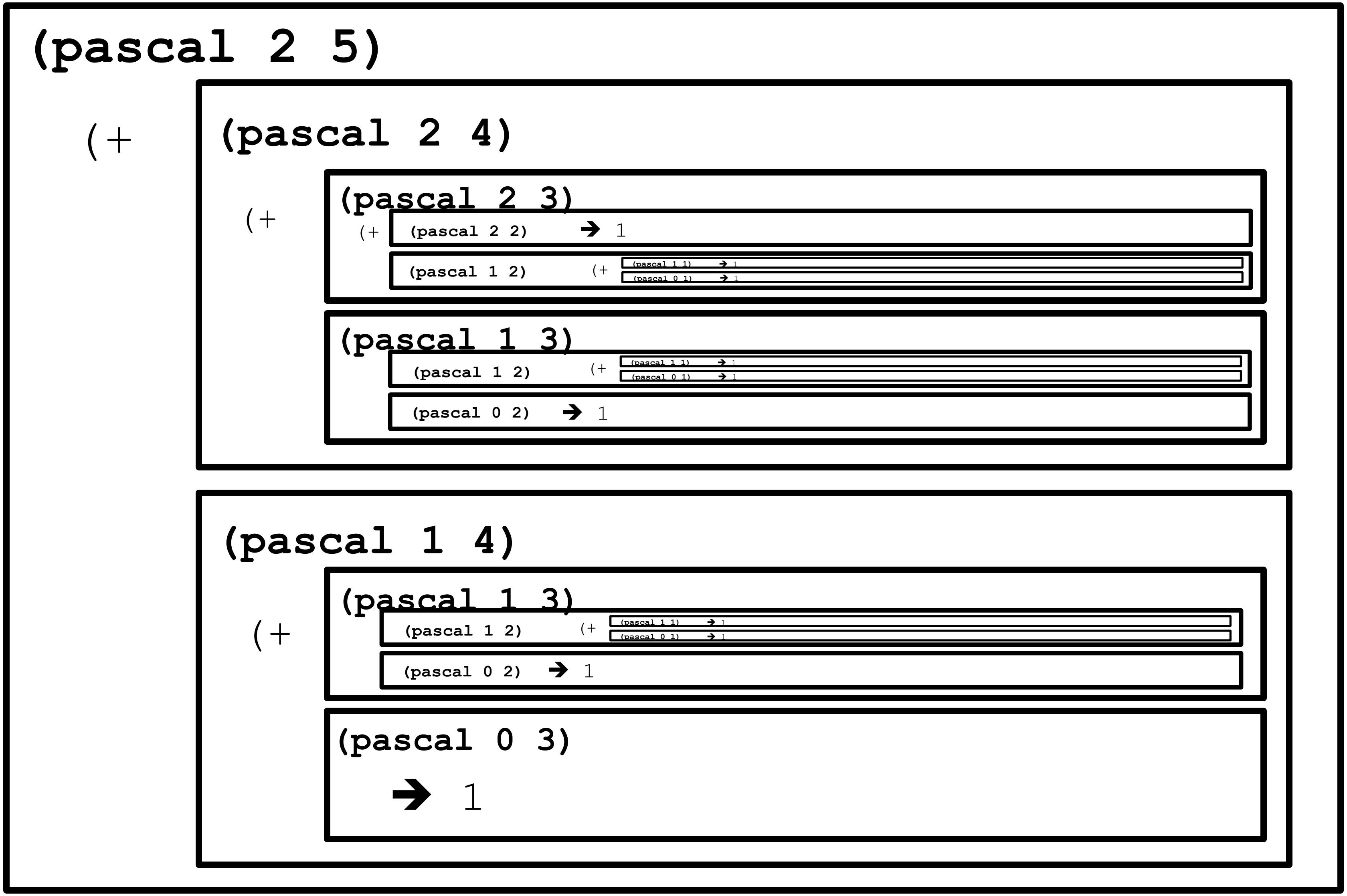
	columns (C)							
		0	1	2	3	4	5	...
rows (R)	0	1						...
	1	1	1					...
	2	1	2	1				...
	3	1	3	3	1			...
	4	1	4	6	4	1		...
	5	1	5	10	10	5	1	...
	...	...	...	...	...	...	...	...

Pascal's Triangle

- How many ways can you choose C things from R choices?
- Coefficients of the (x+y)^R: look in row R
- etc.

```
(define (pascal C R)
  (cond
    ((= C 0) 1)    ;base case
    ((= C R) 1)    ;base case
    (else          ;tree recurse
     (+ (pascal    C    (- R 1))
        (pascal (- C 1) (- R 1))
     )))
```

> (pascal 2 5)



(pascal 2 5)
(+
(pascal 2 4)
(+
(pascal 2 3)
(+
(pascal 2 2) ➔ 1
(pascal 1 2)
(+
(pascal 1 1) ➔ 1
(pascal 0 1) ➔ 1
(pascal 1 3)
(pascal 1 2)
(+
(pascal 1 1) ➔ 1
(pascal 0 1) ➔ 1
(pascal 0 2) ➔ 1
(pascal 1 4)
(+
(pascal 1 3)
(pascal 1 2)
(+
(pascal 1 1) ➔ 1
(pascal 0 1) ➔ 1
(pascal 0 2) ➔ 1
(pascal 0 3)
➔ 1

Midterm like Problems…

Tree-recursion (1 of 2)

- Consider a set of three coins: a penny, worth 1 cent; a nickle, worth 5 cents; and a dime, worth 10 cents. Write a procedure named possible-amounts which takes a number n, and returns a sentence of all the possible amounts that any n coins of these three types can make.

- Fill in the blanks to make the definition of possible-amounts work correctly:

`(possible-amounts 1)` → `(1 5 10)`
`(possible-amounts 2)` → `(2 6 11 10 15 20)` *(This includes two pennies, a penny and a nickel, a penny and a dime, two nickels, a nickel and a dime, and two dimes)*
`(possible-amounts 3)` → `(3 7 12 11 16 21 15 20 25 30)`

Tree-recursion (2 of 2)

```
(define *coin-amounts* ___________________________)

(define (possible-amounts n)
   (pa-helper *coin-amounts* n))

(define (pa-helper coins n)
   (cond ((<= n 1) ____________________)                  ;; base case 1
         ((empty? coins) _________________________)       ;; base case 2
         (else (se (add-coin-to-every                     ;; recur case 1
                        (first coins)
                        (pa-helper coins (- n 1)))
                   (pa-helper                             ;; recur case 2
                        ______________________________________
                        _________________________________ )))))


;; add coin to each element of sent
(define (add-coin-to-every coin sent)
   (every (lambda (num)
              (+ coin num))
          sent))
```

Write `successive-concatenation`

```
(sc '(a b c d e))
➔   (a ab abc abcd abcde)

(sc '(the big red barn))
➔  (the thebig thebigred thebigredbarn)


          (define (sc sent)
              (accumulate
                  (lambda ??
                       )
                  sent))
```

make-decreasing

- `make-decreasing`
  - Takes a sentence of numbers
  - Returns a sentence of numbers, having removed elements of the input that were not larger than all numbers to the right of them.

```
(make-decreasing '(9 6 7 4 6 2 3 1))
  ➔ (9 7 6 3 1)
(make-decreasing '(3)) ➔ (3)
```

Write first as a recursion, then as a HOF

gather

- Consider the recursive procedure gather that takes a sentence of at least two single-character words (i.e., letters such as 'a', 'b', etc.):

```
;; sent-of-ltrs is a sentence of at least 2 words that are
;; single letters
(define (gather sent-of-ltrs)
   (cond ((empty? sent-of-ltrs) '())
         ((empty? (bf sent-of-ltrs))
          (se (first sent-of-ltrs)))
         ((equal? (first (first sent-of-ltrs))
                  (first (bf sent-of-ltrs)))
          (gather (se (word (first sent-of-ltrs)
                            (first (bf sent-of-ltrs)))
                      (bf (bf sent-of-ltrs)))))
         (else
          (se (first sent-of-ltrs)
              (gather (bf sent-of-ltrs))))))
```

- *Part A (3 points).* What will (gather '(a b b b c d d)) return?
- *Part B (6 points).* Write gather-hof, which behaves the same as gather but uses no explicit recursion.