CS3:
Introduction to Symbolic Programming

Lecture 10:
Tree recursion
Midterm 2

Spring 2008

Nate Titterton
nate@berkeley.edu

Schedule

11	**Mar 31 – Apr 4**	**Lecture: Tree Recursion, Midterm review** **Lab: Tree Recursion** **Mini-Project #3 (Due Friday at midnight)** **Reading: "Counting Change" case study**
12	**Apr 7-11**	***Midterm #2*** **Lab: Introduction to lists**
13	**Apr 14-18**	**Lecture: Lists, lists, lists** **Lab: Generalized lists** **Sequential Programming**
14	**April 21-25**	**Introduction to the project**
15	**April 28 – May 2**	**The project**

Midterm #2

- **Next Week (April 7th)**
  - 90 minutes (4:10-5:40).
  - Room **Valley Life Sciences 2050** (same as last time)
  - Open book, open notes, etc.
  - Check for practice exams and solution on the course portal and in the reader.

- **Midterm 2 review session**
  - Saturday, 2-4 pm
  - 306 Soda (as last time)

What does midterm #2 cover?

- **Advanced recursion (accumulating, multiple arguments, etc.).**
- **Tree-recursion (from this week)**
- **All of higher order functions**
- **lambda, let, global variables, etc…**
- **Those "big" homeworks (bowling, compress, and occurs-in)**
- **Elections and number-name miniprojects**
- **Reading and programs:**
  - **Change making, Roman numerals**
  - **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)**

The last of Advanced HOF

You *need* lambda when…

…you need a procedure to make reference to more values than you can pass it.

For instance, when a procedure for use in an `every` needs two parameters

```
(prepend-every 'sir- '(sam mary loin))
    ➔ (sir-sam sir-mary sir-loin)
```

Write `prepend-every`

Write `appearances`

`make-bookends` (a *small* problem)

- **Write `make-bookends`, which is used this way:**

```
((make-bookends 'o) 'hi) ➔ ohio

((make-bookends 'to) 'ron) ➔ toronto

(define tom-proc (make-bookends 'tom))
(tom-proc "") ➔ tomtom
```

Accumulate: returning sentences

- **`accumulate` can return a sentence…**

```
(accumulate ?? '(a b c d))
        ➔ (ab bc cd)
```

  - **the *first* time accumulate is run, it reads the last two words of the input sentence**

  - **in *later* calls, it uses the return value of its procedure (which is a sentence) as one of its arguments**

every containing every

- You can mimic 2-stage recursion, applying a function to each letter of each word.

- You can get combinatoric effects:

```
(define (pair-all sent)
  (every (lambda (one)
           (every (lambda (two)
                    (word one two))
                  sent))
         sent))

(pair-all '(a b c)) ➔ ???
```

every containing every containing…

```
(make-kindergarten-words '(s t) '(a o))
     ➔ (sas sat sos sot tas tat tos tot)
(make-kindergarten-words '(l n k t s) '(a e i o u))
     ➔ 225 words!

(define (make-kindergarten-words consonants vowels)
  (every (lambda (c)
           (every (lambda (v)
```

```
                    vowels))
         consonants))
```

Tree Recursion

What will happen?

- **What will `countem` return for n=1, 2, …?**

```
(define (countem n)
   (if (= n 0)
       '()
       (se (countem (- n 1))
           n
           (countem (- n 1)))))
```

Tree recursion

A recursive technique in which more than one recursive call is made within a recursive case.

Pascal's triangle

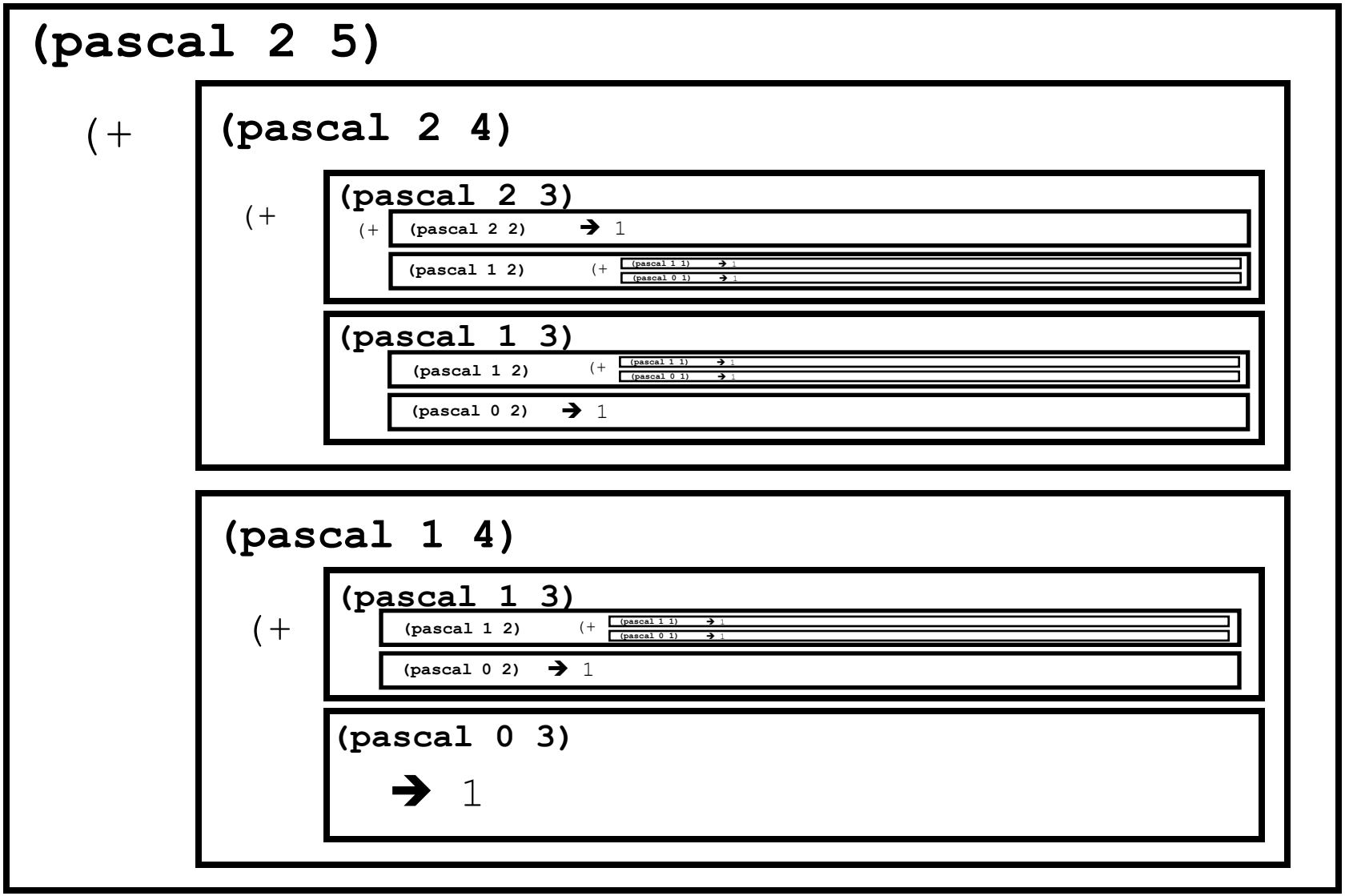
	columns (C)							
rows (R)		0	1	2	3	4	5	...
	0	1						...
	1	1	1					...
	2	1	2	1				...
	3	1	3	3	1			...
	4	1	4	6	4	1		...
	5	1	5	10	10	5	1	...
	...	...	...	...	...	...	...	...

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

Chips and Drinks

"I have some bags of chips and some drinks. How many different ways can I finish all of these snacks if I eat one at a time?

`(snack 1 2)` → `3`

- This includes (chip, drink, drink), (drink, chip, drink), and (drink, drink, chip).

`(snack 2 2)` → `6`

- (c c d d), (c d c d), (c d d c)
  (d c c d), (d c d c), (d d c c)

A variable number of recursive calls…

- **Consider "Joe numbers":**
  - The n^{th} joe-number is the sum of all the joe-numbers under it (i.e., joe^{n-1} to joe^{1}).
  - Joe^{1} is simply 1.

- **Write a procedure to calculate Joe^{n}.**

  - A procedure `down-from` that, given `n`, returns a sentence of numbers from `n` to `1` should be useful. And easy to write!
    - `(down-from 6)` ➔ `(6 5 4 3 2 1)`

Problems

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))
```

binary

- **Write `binary`, a procedure to generate the possible binary numbers given `n` bits.**

```
(binary 1)→(0 1)
(binary 2)→(00 01 10 11)
(binary 3)→(000 001 010 011 100 101 110 111)
```