Lecture 11:
Midterm #2 review
<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Lecture</th>
<th>Lab:</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Mar 20-24</td>
<td>More HOF, Tic-Tac-Toe, Tree Recursion</td>
<td>Reading: SS 10, 15; &quot;Change Making&quot; case study</td>
</tr>
<tr>
<td>11</td>
<td>Mar 27-31</td>
<td><strong>(Spring Break)</strong></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Apr 3-7</td>
<td>Lecture: Review</td>
<td>Lab: Miniproject #3</td>
</tr>
<tr>
<td>13</td>
<td>Apr 10-14</td>
<td>Lecture: MIDTERM #2</td>
<td>Lab: Start on &quot;Lists&quot;</td>
</tr>
<tr>
<td>14</td>
<td>Apr 17-21</td>
<td>Lecture: Lists, and introduce the big project</td>
<td>Lab: Lists; start on the project</td>
</tr>
<tr>
<td>15</td>
<td>Apr 24-28</td>
<td>Lecture: Lists, and ?</td>
<td>Lab: Work on the project</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>rows (R)</th>
<th>columns (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
| ...      | ...| ...| ...| ...| ...| ...| ...

### 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)))))
\[
\begin{align*}
> \quad \text{(pascal 2 5)} \\
\text{(pascal 2 5)} \equiv \\
(+) \quad \text{(pascal 2 4)} \equiv \\
(+) \quad \text{(pascal 2 3)} \equiv \\
(+) \quad (\text{pascal 2 2}) \rightarrow 1 \\
(+) \quad (\text{pascal 1 2}) \equiv \\
(+) \quad (\text{pascal 0 2}) \rightarrow 1 \\
\text{(pascal 1 3)} \equiv \\
(+) \quad (\text{pascal 1 2}) \equiv \\
(+) \quad (\text{pascal 0 2}) \rightarrow 1 \\
\text{(pascal 1 4)} \equiv \\
(+) \quad \text{(pascal 1 3)} \equiv \\
(+) \quad (\text{pascal 1 2}) \equiv \\
(+) \quad (\text{pascal 0 2}) \rightarrow 1 \\
\text{(pascal 0 3)} \rightarrow 1
\end{align*}
\]
Midterm like Problems...
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:

\[
\begin{array}{c}
(\text{possible-amounts } 1) \Rightarrow (1 \ 5 \ 10) \\
(\text{possible-amounts } 2) \Rightarrow (2 \ 6 \ 11 \ 10 \ 15 \ 20) \\
(\text{This includes two pennies, a penny and a nickel, a penny and a dime, two nickels, a nickel and a dime, and two dimes}) \\
(\text{possible-amounts } 3) \Rightarrow (3 \ 7 \ 12 \ 11 \ 16 \ 21 \ 15 \ 20 \ 25 \ 30)
\end{array}
\]
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))
```
(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

- 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
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.