Lecture 8: The last bit of recursion
Miniproject #2
<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Oct 9-13</td>
<td>Advanced recursion</td>
</tr>
<tr>
<td>8</td>
<td>Oct 16-20</td>
<td>Finishing recursion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Miniproject #2: Number names</td>
</tr>
<tr>
<td>9</td>
<td>Oct 23-27</td>
<td>Introduction to Higher Order Procedures</td>
</tr>
<tr>
<td>10</td>
<td>Oct 30 - Nov 3</td>
<td>More HOF</td>
</tr>
<tr>
<td>11</td>
<td>Nov 6-10</td>
<td>Finish HOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Miniproject #3: Election processing</td>
</tr>
<tr>
<td>12</td>
<td>Nov 13-17</td>
<td>Lecture: <em>Midterm #2</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Start on &quot;Lists&quot;</td>
</tr>
</tbody>
</table>
Any "notetaker" volunteers?

• A student in the course needs a note taker, which does pay a stipend. If you are taking notes anyway…

  - Come and see me after lecture if interested
The "screwed up" labs

• This is the order things should have happened:
  - First "advanced recursion" Lab: recursions with multiple arguments
    - my-equal?, zipping, merging
  - Second Lab
    - patterns in recursion, no-vowels, sort (using insert), roman-sum-helper
  - Last Lab
    - mad-libs quiz, l-extra?, fibonacci, thorough-reversal
Number Spelling Miniproject

• 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.
"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 remember everything
(define (length sent)
   (if (empty? sent)
       0
       (+ 1 (length (bf sent))))
)
(length '(a b c d)) \Rightarrow
(+ 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
(define (sent-max sent)
  (if (empty? sent)
      '()
      (sent-max-helper (bf sent) (first sent)))))

(define (sent-max-helper sent max-so-far)
  (if (empty? sent)
      max-so-far
      (sent-max-helper (bf sent)
          (if (> max-so-far (first sent))
              (max-so-far
               (first sent)))))
(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))

(sent = (2 3 4 5 6))

(se 2)

(sent = (3 4 5 6))

(se 4)

(sent = (4 5 6))

(se 6)

(sent = (6))

()
Tree recursion: fibonacci

- The fibonacci sequence:
  1 1 2 3 5 8 13 21 34 55

```
(define (fib n)
  (if (<= n 2)
      1 ;; base case
      (+ (fib (- n 1)) ;; recursive case
          (fib (- n 2)))))
```
### Tree recursion: Pascals triangle

**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.

<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>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
(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 2) ➞ 1

(pascal 1 3)

(+)

(pascal 1 2) (+)

(pascal 1 1) ➞ 1

(pascal 0 2) ➞ 1

(pascal 1 4)

(+)

(pascal 1 3)

(+)

(pascal 1 2) (+)

(pascal 1 1) ➞ 1

(pascal 0 2) ➞ 1

(pascal 0 3)

⇒ 1
pair-all

- Write pair-all, which takes a sentence of prefixes and a sentence of suffixes and returns a sentence pairing all prefixes to all suffixes.

- (pair-all '(a b c) '(1 2 3)) \(\Rightarrow\)
  (a1 b1 c1 a2 b2 c2 a3 b3 c3)

- (pair-all '(spr s k) '(ite at ing ong)) \(\Rightarrow\)
  (sprite sprat spring sprong site sat sing song kite kat king kong)
• Write binary, a procedure to generate the possible binary numbers given $n$ bits.

(binary 1) $\rightarrow$ (0 1)
(binary 2) $\rightarrow$ (00 01 10 11)
(binary 3) $\rightarrow$ (000 001 010 011 100 101 110 111)
Write `roman-sum-helper`:

```scheme
(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:**

```scheme
(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 156 '( ) 5)`