Recursion

(define (fac N)
  (if (= N 0)
      1
      (* N (fac (- N 1))))

base case
recursive case
Embedded Recursion

(define (fac N)
  (if (= N 0)
      1
      (* N (fac (- N 1)))))

...
Tail Recursion

Refactor to use an accumulator

(define (tail-fac-helper N accum)
  (if (= N 0)
      accum
      (tail-fac-helper (- N 1) (* accum N))))

(define (tail-fac N)
  (tail-fac-helper 1))
Tail Recursion
Refactor to use an accumulator

(define (tail-fac-helper N accum)
  (if (= N 0)
      accum
      (tail-fac-helper (- N 1) (* accum N))))

(define (tail-fac N)
  (tail-fac-helper N 1))
Embedded vs. Tail Recursion

Benefits & Drawbacks?
Inductive Data Types & Recursive Operations
(Putting Together & Pulling Apart)
Racket Lists
Syntax and semantics

<table>
<thead>
<tr>
<th>syntax</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>'()</td>
<td>[ ]</td>
</tr>
<tr>
<td>(cons 1 '())</td>
<td>1 [ ]</td>
</tr>
<tr>
<td>(cons 1 (cons 2 '()))</td>
<td>1 [ ] → 2 [ ]</td>
</tr>
<tr>
<td>(cons &lt;value&gt; &lt;list&gt;)</td>
<td>&lt;e&gt; [ ] → &lt;l&gt; [ ]</td>
</tr>
<tr>
<td>(cons &lt;value&gt; &lt;non-list&gt;)</td>
<td>1 2</td>
</tr>
</tbody>
</table>
## Racket Lists

Constructors, selectors, and operations

<table>
<thead>
<tr>
<th><strong>constructors</strong></th>
<th>put together</th>
<th><strong>selectors</strong></th>
<th>take apart</th>
<th><strong>operations</strong></th>
<th>usually recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>'()</td>
<td></td>
<td>(first &lt;list&gt;)</td>
<td></td>
<td>(length &lt;list&gt;)</td>
<td></td>
</tr>
<tr>
<td>(cons &lt;value&gt; &lt;list&gt;)</td>
<td></td>
<td>(rest &lt;list&gt;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'(&lt;value₁&gt; ... &lt;valueₙ&gt;)</td>
<td></td>
<td>(null? &lt;value&gt;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(list &lt;value₁&gt; ... &lt;valueₙ&gt;)</td>
<td></td>
<td>(list? &lt;value&gt;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... more in the Racket documentation: [http://docs.racket-lang.org/reference/pairs.html](http://docs.racket-lang.org/reference/pairs.html)
Implement these functions
you should need only the constructors and selectors we’ve covered

**sum**: adds up all the numbers in a list \( L \)
you can assume \( L \) contains only integers

**count–ones**: counts the number of times the value 1 appears in a list \( L \)
you can assume \( L \) contains only integers

**plus–one**: adds 1 to every element of a list \( L \)
you can assume \( L \) contains only integers

**deep–count–ones**: same as **count–ones**
extcept \( L \) can contain lists of (lists of …) integers

**BONUS**

**weave**: given two lists \( L_1 \) and \( L_2 \), produces: '(L\(_0\) L\(_2\) L\(_1\) L\(_2\) \ldots \ L\(_1\) n \ L\(_2\) n)
you can assume that \( L_1 \) and \( L_2 \) have the same length

**zip**: given two lists \( L_1 \) and \( L_2 \), produces: '((L\(_1\)\(_0\) L\(_2\)\(_0\)) \ldots (L\(_1\)\(_n\) L\(_2\)\(_n\)))
you can assume that \( L_1 \) and \( L_2 \) have the same length

**stripe**: given a list \( L \) produces: '(L\(_0\) L\(_2\) L\(_4\) \ldots )

---

**Practice pair programming.**
- switch off after every function
- increase your understanding

**Practice test-driven-development (TDD).**
- initial tests available at:
  www.cs.hmc.edu/cs42/code/lab-tests.rkt
- add your own

**No higher-order functions (at first)**

**If things get busy, make a queue on the board.**
- write both partners’ names

**We’ll stop at 9:10am.**
- you probably won’t finish; that’s ok
- if you do finish, let me know; there’s more...