Part 1:
Inductive Data Types
& Recursive Operations
(Putting Together & Taking Apart)
# Racket lists

<table>
<thead>
<tr>
<th><strong>syntax</strong></th>
<th><strong>semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>what we write</td>
<td>what it means</td>
</tr>
</tbody>
</table>

### Syntax Examples:

- `'(())`
  - Semantics: Empty list
  - Diagram: ![Empty List Diagram](image1)

- `(cons <value> <list>)`
  - Semantics: Cons cell with value and list
  - Diagram: ![Cons Cell Diagram](image2)

- `'(<value1> ... <valueN>)`
  - or
  - `(list <value1> ... <valueN>)`
    - Semantics: List of values
    - Diagram: ![List Diagram](image3)

- `(cons <value> <non-pair>)`
  - or
  - `'(<value> <non-pair>)`
    - Semantics: Cons cell with value and non-pair
    - Diagram: ![Cons Cell with Non-Pair Diagram](image4)
List construction: your turn
write down answers as either a drawing or a Racket expression

1. `(cons 3 (cons 2 (cons 1 '())))`

2. `'(1 2 3)`

3. 

4. 

5. Bonus: `(cons 1 (cons (cons 2 '()) '())))`

6. Bonus: `'(()) . 1)`
   - What Racket expression would you write to make Racket respond with this?
   - How would you draw it?
List construction: your turn

write down answers as either a drawing or a Racket expression

1. \((\text{cons } 3 \ (\text{cons } 2 \ (\text{cons } 1 \ '(()'))))\)

2. \'(1 2 3)\)

3. \[\text{Diagram of 1 } \rightarrow \text{Diagram of 2}\]

4. \[\text{Diagram of 1 and 2 as a list}\]

5. Bonus: \((\text{cons } 1 \ (\text{cons (cons 2 '(())) '(()))))\)

6. Bonus: \'((() . 1)\)

   - What Racket expression would you write to make Racket respond with this?
   - How would you draw it?

\[(\text{cons } 1 \ (\text{cons } 2 \ '(()))\)
\[(\text{cons } 1 \ 2)\]
\[(\text{cons } '(() \ 1)\]

\[\text{Diagram of 1}
\text{You'll probably never see this}\]
Recursive operations
;;; len
;;;  inputs: a list, L
;;;  outputs: the number of elements in the list
(define (len L)

; tests
(check-expect (len '()) 0)
(check-expect (len '(1 2 3)) 3)
(check-expect (len '((1 2 3))) 1)
;; len
;; inputs: a list, L
;; outputs: the number of elements in the list
(define (len L)
  (if (null? L)
      0
      (+ 1 (len (rest L))))

; tests
(check-expect (len '()) 0)
(check-expect (len '(1 2 3)) 3)
(check-expect (len '((1 2 3))) 1)
;;; len
;;; inputs: a list, L
;;; outputs: the number of elements in the list
(define (len L)
  (if (empty? L)
      0
      (+ 1 (len (rest L)))))

; tests
(check-expect (len '()) 0)
(check-expect (len '(1 2 3)) 3)
(check-expect (len '(((1 2 3))) 1)
### Racket lists

Constructors, accessors, and operations

<table>
<thead>
<tr>
<th><strong>constructors</strong></th>
<th><strong>accessors</strong></th>
<th><strong>operations</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>put together</em></td>
<td><em>take apart</em></td>
<td><em>often recursive</em></td>
</tr>
<tr>
<td><code>()</code></td>
<td>(first &lt;list&gt;)</td>
<td>(length &lt;list&gt;)</td>
</tr>
<tr>
<td><code>(cons &lt;value&gt; &lt;list&gt;)</code></td>
<td>(rest &lt;list&gt;)</td>
<td>(member &lt;value&gt; &lt;list&gt;)</td>
</tr>
<tr>
<td><code>(''&lt;value</code></td>
<td>...</td>
<td>(empty? &lt;list&gt;)</td>
</tr>
<tr>
<td><code>(list &lt;value</code></td>
<td></td>
<td>(list? &lt;value&gt;)</td>
</tr>
<tr>
<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)
Recursion over lists: your turn

See recursion.rkt for solutions
Part 2: Embedded vs Tail Recursion (code optimizations)
Recursion

Math vs machines

\[
n! = \begin{cases} 
1 & : n = 0 \\
(n \cdot (n - 1))! & : \text{otherwise}
\end{cases}
\]

(define (fact N)
  (if (= N 0)
    1
    (* N (fact (- N 1)))))
Embedded recursion

(define (fact N)
  (if (= N 0)
      1
      (* N (fact (- N 1)))))
#lang racket
(require htdp/testing)

;; fact
;; fact: computes n!
;; inputs: n, a number
;; outputs: n!
(define (fact N)
  (if (equal? N 0)
      1
      (* N (fact (- N 1)))))

;; tests
(check-expect (fact 0) 1)
(check-expect (fact 1) 1)
(check-expect (fact 5) 120)

(generate-report)
Tail recursion

Refactor function so that it uses an accumulator

(define (tail-fac N)
  (tail-fac-helper N 1))

(define (tail-fac-helper N accum)
  (if (equal? N 0)
      accum
      (tail-fac-helper (- N 1) (* accum N))))
Use trace to help investigate / debug
Next assignment
Due next Tuesday at 11:59pm

- Available later tonight
- More of what we did today
  Although, you’ll need to research more list operations
- First part is on your own
- Second part may be with a partner (Scrabble!)
- Third part is *optional* Java practice