Tail recursion

A recursive function is tail recursive when all recursive calls are tail calls.
Code transformation 1: recursion -> tail recursion

Key idea: use *accumulators* to track “leftover” computations
Code transformation 2: tail recursion -> iteration

Key idea: parameters become loop-updated variables
Higher-order functions

Taking abstraction to the next level
\[
(+ 32 (* 1 (/ 9 5)))
\]
\[
(+ 32 (* 100 (/ 9 5)))
\]
\[
(+ 32 (* -2 (/ 9 5)))
\]
\[
(lambda (x)
  (+ 32 (* x (/ 9 5))))
\]
(+ 32 (* 1 (/ 9 5)))
(- 32 (* 1 (/ 9 5)))
(* 32 (* 1 (/ 9 5)))

(lambda (x)
  (x 32 (* 1 (/ 9 5))))
(lambda (n) (+ n 1))

(lamba (n) (+ n 200))

(lamba (n) (+ n -324))

(lamba (x)
   (lamba (n) (+ n x)))
Higher-order function

a function that satisfies one or both of:

1) takes a function as an argument
2) returns a function
Case study: three famous list HOFs
Take a list of floats and round each one to two decimal places.

Take a list of strings and strip trailing whitespace.

Take a list of temperatures in Celsius and convert them to Farenheit.

Take a list of HTML elements and extract their attributes.
new_list = []
for x in lst:
    new_item = f(x)
    new_list.append(new_item)
Take a list of floats and remove the ones < 50.

Take a list of strings and remove the ones that start with ‘a’.

Take a list of students and remove the ones in CSC324.

Take a list of HTML elements and remove all but the <a> tags.
new_list = []
for x in lst:
    if f(x):
        new_list.append(x)
Generic list iteration

```
acc = seed
for x in lst:
    acc = f(x, acc)
```
Generic list iteration (done recursively)

\[
\begin{align*}
(\text{define (func } f \text{ acc lst)} & \quad (\text{if (null? lst)} \\
\quad \text{ acc} & \quad (\text{func } f \quad (f \quad (\text{first lst}) \quad \text{acc}) \\
\quad \text{ (rest lst))))
\end{align*}
\]
Summary: lookup the following functions in the Racket and Haskell docs

map
filter
foldl
(and your fave language!)
Two more famous HOFs
(define (compose f g)
    (lambda (x)
        (f (g x))))

\[
\text{f \circ g = } \lambda x \rightarrow f \left( g \left( x \right) \right)
\]
(apply f (list x1 x2 ... xn)) ==

(f x1 x2 ... xn)
\( f \$ x = f \times \)
Implementation question:

How do we implement functions that are returned by other functions?
(define (make-f n)
  (lambda (x)
    ; REALLY long body
    (... n ... x ...)))

(define f-1 (make-f 1))
(define f-2 (make-f 2))
(define f-3 (make-f 3))
(define (make-f n)
  (lambda (x)
    ; REALLY long body
    (... \(n\) ... \(x\) ...)))

(define f-1 (lambda (x) (... \(1\) ... \(x\) ...)))
(define f-2 (lambda (x) (... \(2\) ... \(x\) ...)))
(define f-3 (lambda (x) (... \(3\) ... \(x\) ...)))
environment

a mapping of identifiers to values
closure

A data structure storing two things:

1) a reference to function code
2) an environment containing bindings for all “missing” identifiers in the function body
(define (make-f n)
  (lambda (x)
    ; REALLY long body
    (... n ... x ...)))

(define f-1 (lambda (x) (... 1 ... x ...)))
(define f-2 (lambda (x) (... 2 ... x ...)))
(define f-3 (lambda (x) (... 3 ... x ...)))
(define (make-f n)
  (lambda (x)
    ; REALLY long body
    (... n ... x ...)))

(define f-1 (0x00ff48, {n: 1}))
(define f-2 (0x00ff48, {n: 2}))
(define f-3 (0x00ff48, {n: 3}))