All of the recursion we have done to date has followed a pattern we call **simple recursion**. The templates we have been using have been derived from a data definition and specify the form of the recursive application.

We will now learn to use a new pattern of recursion, **accumulative recursion**, and learn to recognize **generative recursion**.

For the next several lecture modules we will use simple recursion and accumulative recursion. We will avoid generative recursion until the end of the course.
Recall from Module 06:

In **simple recursion**, every argument in a recursive function application (or applications, if there are more than one) is either:

- unchanged, or
- *one step* closer to a base case, using the inverse of the function in the data definition.
(max-list-v2 lon) produces the maximum element of lon

Examples:

(check-expect (max-list-v2 (list 6 2 3 7 1)) 7)

(max-list-v2: (listof Num) -> Num)

Defines: lon is nonempty

(define (max-list-v2 lon)
  (cond [(empty? (rest lon)) (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else (max-list-v2 (rest lon))])))

There may be two recursive applications of max-list.
> max-list is slow

The code for max-list-v2 is correct.

But computing \( \text{max-list-v2 (countup-to 1 25)} \) is very slow.

Why?

The initial application is on a list of length 25.

There are two recursive applications on the rest of this list, which is of length 24.

Each of those makes two recursive applications.
We informally call this **exponential blowup**.
We can take the number of recursive applications as a rough measure of a function’s efficiency. `max-list-v2` can take up to $2^n - 1$ recursive applications.

`length` makes $n$ recursive applications on a list of length $n$.

`length` is clearly more efficient than `max-list-v2`.

We say that `length`’s efficiency is proportional to $n$ and `max-list-v2`’s efficiency is proportional to $2^n$. We express the former as $O(n)$ and the later as $O(2^n)$.
There are “families” of algorithms with similar efficiencies. Examples, from most efficient to least:

<table>
<thead>
<tr>
<th>“Big-O”</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>no recursive calls; <strong>tax-payable</strong></td>
</tr>
<tr>
<td>$O(\log_2 n)$</td>
<td>divide in half, work on one half; <strong>binary-search</strong> on a <em>balanced tree</em></td>
</tr>
<tr>
<td>$O(n)$</td>
<td>one recursive application for each item; <strong>length</strong></td>
</tr>
<tr>
<td>$O(n \log_2 n)$</td>
<td>divide in half, work on both halves; <strong>merge-sort</strong></td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>an $O(n)$ application for each item; <strong>insertion-sort</strong></td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>two recursive applications for each item; <strong>max-list</strong></td>
</tr>
</tbody>
</table>

Much more about “Big-O” notation and efficiency in later courses.
> Recap

Fast $O(n)$

\[\text{(define (max-list-v1 lon)} \]
\[\hspace{1em} \text{(cond [(empty? (rest lon)) (first lon)]}\]
\[\hspace{1em} [\text{else (max (first lon) (max-list-v1 (rest lon))}]\]})

Slow $O(2^n)$

\[\text{(define (max-list-v2 lon)} \]
\[\hspace{1em} \text{(cond [(empty? (rest lon)) (first lon)]}\]
\[\hspace{1em} [\text{>( (first lon) (max-list-v2 (rest lon))) (first lon)]\]
\[\hspace{1em} [\text{else (max-list-v2 (rest lon))}]\])}
Humans don’t seem to use either of the two versions of max-list shown earlier.

Instead, we tend to find the maximum of a list of numbers by scanning it, remembering the largest value seen so far. When we see a value that’s larger than the largest seen so far, we remember the new value – until we see one that is still larger. When we get to the end of the list, the largest value seen so far is the largest value in the list.
Computationally, we can pass down that largest value seen so far as a parameter called an **accumulator**.

This parameter accumulates the result of prior computation, and is used to compute the final answer that is produced in the base case.

This approach results in the code on the next slide.
;; (max-list/acc lon max-so-far) produces the largest
;; of the maximum element of lon and max-so-far

;; max-list/acc: (listof Num) Num -> Num
(define (max-list/acc lon max-so-far)
  (cond [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
         (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]
        ))

(define (max-list-v3 lon)
  (max-list/acc (rest lon) (first lon)))
Now even \((\text{max-list2} \ (\text{countup-to} \ 1 \ 200000))\) is fast.

\[
(\text{max-list2} \ (\text{list} \ 1 \ 2 \ 3 \ 9 \ 5)) \\
\Rightarrow (\text{max-list/acc} \ (\text{list} \ 2 \ 3 \ 9 \ 5) \ 1) \\
\Rightarrow (\text{max-list/acc} \ (\text{list} \ 3 \ 9 \ 5) \ 2) \\
\Rightarrow (\text{max-list/acc} \ (\text{list} \ 9 \ 5) \ 3) \\
\Rightarrow (\text{max-list/acc} \ (\text{list} \ 5) \ 9) \\
\Rightarrow (\text{max-list/acc} \ (\text{list} \ ) \ 9) \\
\Rightarrow 9
\]
This technique is known as **accumulative recursion**.

It is more difficult to develop and reason about such code, which is why simple recursion is preferable if it is appropriate.

HtDP discusses it much later than we are doing (after material we cover in lecture module 10) but in more depth.
Indicators of the accumulative recursion pattern

- All arguments to recursive function applications are:
  - unchanged, or
  - one step closer to a base case in the data definition, or
  - a partial answer (passed in an accumulator).
- The value(s) in the accumulator(s) are used in one or more base cases.
- The accumulatively recursive function usually has a wrapper function that sets the initial value of the accumulator(s).
Another accumulative example: reversing a list

Using simple recursion:

\[
\text{define} \ (\text{my-reverse} \ \text{lst}) \\
\text{  (cond} \\
\text{    [(empty? lst) empty]} \\
\text{    [else (append (my-reverse (rest lst))} \\
\text{      (list (first lst)))]})
\]

Intuitively, \text{append} does too much work in repeatedly moving over the produced list to add one element at the end.

This has the same worst-case behaviour as insertion sort, \(O(n^2)\).
(my-reverse lst) reverses lst using accumulative recursion

Example:
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

;; my-reverse: (listof X) -> (listof X)
(define (my-reverse lst)  ; wrapper function
  (my-rev/acc lst empty))

(define (my-rev/acc lst acc)  ; helper function
  (cond [(empty? lst) acc]
        [else (my-rev/acc (rest lst) (cons (first lst) acc))])))

This is $O(n)$. 
A condensed trace

(my-reverse (list 1 2 3 4 5))
⇒ (my-rev/acc (list 1 2 3 4 5) empty)
⇒ (my-rev/acc (list 2 3 4 5) (cons 1 empty))
⇒ (my-rev/acc (list 3 4 5) (cons 2 (list 1)))
⇒ (my-rev/acc (list 4 5) (cons 3 (list 2 1)))
⇒ (my-rev/acc (list 5) (cons 4 (list 3 2 1)))
⇒ (my-rev/acc (list) (cons 5 (list 4 3 2 1)))
⇒ (list 5 4 3 2 1)
The $n$th Fibonacci number is the sum of the two previous Fibonacci numbers:

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$$

This can be implemented directly using simple recursion, as follows:

```scheme
(define (fib n)
  (cond [(< n 2) n]
        [else (+ (fib (- n 1))
                 (fib (- n 2)))]))
```

This works: $(\text{fib } 6) \Rightarrow 8$, $(\text{fib } 25) \Rightarrow 75025$. But $(\text{fib } 50)$ takes days!

It suffers from exponential blowup.

As it turns out, not $2^n$, but $\phi^n$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the Golden Ratio.
Write a function `(extend-fib n lst)` that consumes a Nat and a `(listof Nat)`. Given lst, a list containing at least 2 Fibonacci values in descending order, it returns a list containing n more Fibonacci values.

`(extend-fib 8 (list 1 0)) ⇒ (list 34 21 13 8 5 3 2 1 1 0)`

Write a function `(fiba n)` that is a wrapper for `extend-fib`, and produces the nth Fibonacci number.

`(fiba 0) ⇒ 0
(fiba 50) ⇒ 12586269025
(fiba 250) ⇒ 7896325826131730509282738943634332893686268675876375`
Given a (listof Num), use accumulative recursion to write mean, which produces the average (mean) of the list.

**Hint**

- mean will be a wrapper function.
- How many accumulators do you need?
In Math 135, you learn that the Euclidean algorithm for Greatest Common Divisor (GCD) can be derived from the following identity for $m > 0$:

$$gcd(n, m) = gcd(m, n \mod m)$$

We also have $gcd(n, 0) = n$.

We can turn this reasoning directly into a Racket function.
;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm

;; euclid-gcd: Nat Nat -> Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))

This function does not use simple or accumulative recursion.
Generative recursion

The arguments in the recursive application were \textit{generated} by doing a computation on \( m \) and \( n \).

The function \texttt{euclid-gcd} uses \textit{generative recursion}.

Once again, functions using generative recursion are easier to get wrong, harder to debug, and harder to reason about.

We will return to generative recursion in a later lecture module. Avoid generative recursion until then.
Simple vs. accumulative vs. generative recursion

In **simple recursion**, all arguments to the recursive function application (or applications, if there are more than one) are either unchanged, or *one step* closer to a base case in the data definition.

In **accumulative recursion**, parameters are as above, plus parameters containing partial answers used in the base case.

In **generative recursion**, parameters are freely calculated at each step. (Watch out for correctness and termination!)
A Collatz sequence is defined as follows: start with any natural number. If the previous term is even, the next term is half the previous term; otherwise, the next term is one more than three times the previous. That is,

\[
s_{k+1} = \begin{cases} 
  s_k/2 & \text{if } s_k \text{ is even} \\
  3s_k + 1 & \text{otherwise.}
\end{cases}
\]

For example, starting at 5, it’s odd, so the next value is \(3 \times 5 + 1 = 16\). This is even, so the next value is \(16/2 = 8\). This is even, so the next value is \(8/2 = 4\). This is even, so the next value is \(4/2 = 2\). This is even, so the next value is \(2/2 = 1\).

1 is odd, so the next value is \(3 \times 1 + 1 = 4\). This is even, so the next value is \(4/2 = 2\). This is even, so the next value is \(2/2 = 1\).

1 is odd, so the next value is....

Trying various values, it seems to always reach 1, then loop 1 \(\rightarrow 4 \rightarrow 2 \rightarrow 1\). That’s funny.
Write a function `(collatz-seq sk)` that returns the Collatz sequence starting at `sk`, until it reaches 1.

`(collatz-seq 13) ⇒ (list 13 40 20 10 5 16 8 4 2 1)`

`(collatz-seq 21) ⇒ (list 21 64 32 16 8 4 2 1)`
Goals of this module

- You should be able to recognize uses of simple recursion, accumulative recursion, and generative recursion.
- You should be able to write functions using simple and accumulative recursion.
- You should know that some functions are much more efficient than others, that efficiency is expressed with “Big-O” notation, and that you’ll learn more about this in future courses.
In this module we added the following to our toolbox:

append

These are the functions and special forms currently in our toolbox: