If you have not already, make sure you

- Read *How to Design Programs* Sections 11, 10, 12.
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,

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

**Exercise**

Write a function `(collatz-next sk)` that consumes a `Nat` representing an item in a Collatz sequence, and returns the next item in the sequence.

`(collatz-next 3) => 10`  
`(collatz-next 12) => 6`
Collatz Sequences

You might notice:

(collatz-next 1) => 4
(collatz-next 4) => 2
(collatz-next 2) => 1

If the sequence ever reaches 1, it continues 1, 4, 2, 1, 4, 2, ... forever.

Also, numbers seem to eventually reach 1:
17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

It is conjectured that every starting number will eventually reach 1. Assume that this is true.

Write a function (collatz-seq sk) that returns the Collatz sequence starting at sk, until it reaches 1.

(collatz-seq 5) => (list 5 16 8 4 2 1)
(collatz-seq 1) => (list 1)
Recursion

A definition that refers to itself is said to be **recursive**.

Example: The Peano axioms define natural numbers as follows:

1. 0 is a natural number.

2. For every natural number \(n\), \(S(n)\) is a natural number.

I can represent 1 as \(S(0)\), 2 as \(S(S(0))\), 3 as \(S(S(S(0)))\), and so on. \(S(n)\) is called the **successor function**; it consumes a natural number, and returns the next.

In Racket we may use `add1` as a successor function. (Then `sub1` gives the predecessor.)

A **Data Definition** is a comment that describes a data type. We can define a `Nat` as follows:

```scheme
;; A Nat is either:
;; 0 or
;; (add1 r) where r is a Nat.
```
A recursive data definition includes two parts:

- a **base case**
- one or more **recursive cases**, defined using the term itself.

Example:

```
;; A Nat is either:
;; 0 or
;; (add1 r) where r is a Nat.
```

Recursive functions may follow this structure:

- a **base case** specifies the result for a special value.
- **recursive cases** specify the result in terms of the function itself, closer to the base case.

Example:

```
n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise.} \end{cases}
```

```
;; (factorial n) return n!
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (sub1 n)))]))
```
One of the ideas of the HtDP textbook is that the form of a program may mirror the form of the data.

A **template** is a general framework which we will complete with specifics. It is a starting point for our implementation.
A template for counting down

Recursive functions may follow this structure:

- a **base case** specifies the result for a special value.
- **recursive cases** specify the result in terms of the function itself, closer to the base case.

From the recursive data definition:

```scheme
;; A Nat is either:
;;  0 or
;; (add1 r) where r is a Nat.
```

We can extract a **template** to address the two cases:

```scheme
;; (nat-template n) a template on n down to 0.
;; nat-template: Nat -> Any
(define (nat-template n)
  (cond [(= n 0) ...]
        [else (... n ... (nat-template (sub1 n)) ...)])
)
```
A template for counting down

;; (nat-template n) a template on n down to 0.
;; nat-template: Nat -> Any
(define (nat-template n)
  (cond [ (= n 0) ...]
    [else (... n ... (nat-template (sub1 n)) ...)])

To use the template follow this approach:

1. Always fill in the base case(s) first.
   For this template, what is the answer when (= n 0) ?
2. Then fill in the recursive case(s).
A template for counting down

;; (nat-template n) a template on n down to 0.
;; nat-template: Nat -> Any
(define (nat-template n)
  (cond [(= n 0) ...
         [else (... n ... (nat-template (sub1 n)) ...)]))

Exercise
Write a recursive function (sum-to n) that consumes a Nat and returns the sum of all Nat between 0 and n.
(sum-to 4) => (+ 4 3 2 1 0) => 10

Exercise
Complete countdown using recursion. (Hint: use cons.)
;; (countdown n) return a list of the natural numbers from n down to 0.
;; countdown: Nat -> (listof Nat)
;; Examples:
(check-expect (countdown 3) (cons 3 (cons 2 (cons 1 (cons 0 '())))))
(check-expect (countdown 5) (list 5 4 3 2 1 0))
A template for counting down: stopping away from zero

We may make a data definition for numbers greater than any value, for example, 7:

;; A Nat7 is either:
;; 7 or
;; (add1 r) where r is a Nat7.

From this we could made a template for a recursive function on Nat7:

;; (nat7-template n) a template n down to 7.
;; nat-template: Nat7 -> Any
(define (nat7-template n)
  (cond [(= n 7) ...
         [else (... n ... (nat-template (sub1 n)) ...)])})
A template for counting down: stopping away from zero

Doing this for a fixed base value (0 or 7) is rather limiting. We can generalize, providing the base value as a parameter. The data definition becomes:

An integer greater than or equal to \( b \) is either
- \( b \) or
- 1 more than an integer greater than or equal to \( b \).

This gives the template:

```scheme
(define (int-b-template n b)
  (cond [(= n b) ...]
        [else (... n ... (int-b-template (sub1 n) b) ...)]))
```

Note here the parameter \( b \) is passed to the recursive call, unchanged. Only \( n \) changes.
(define (int-b-template n b)
  (cond [(= n b) ...]
        [else (... n ... (int-b-template (sub1 n) b) ...)]))
Our functions can do work on the variables as we go. They can:

- Keep only some values (like `filter`)
- Change values (like `map`)
- Combine value (like `foldr`)
- and more....

```scheme
;; (even-squares-between hi lo) return the list of the squares
;; of the even numbers between lo and hi.
;; even-squares-between: Nat Nat -> Nat

(define (even-squares-between hi lo)
  (cond [((<= hi lo) '())
         [(even? hi) (cons (sqr hi) (even-squares-between (sub1 hi) lo))]
         [else (even-squares-between (sub1 hi) lo)])

This does the same thing:
```
```scheme
(define (even-squares-btw hi lo)
  (map sqr (filter even? (range hi lo -1))))
```
Similarly, we can make a template for counting up.
Start with a new data definition:

An **integer less than or equal to** $t$ is either

- $t$ or
- 1 less than an **integer less than or equal to** $t$.

The recursive call must get closer to the base. So increase the parameter with $(\text{add1 } n)$:

```plaintext
;; (nat-template n t) a template on n up to t.
;; nat-template: Nat -> Any
;; Requires: n <= t

(define (nat-upto-template n t)
  (cond [(= n t) ...]
        [else (... n ... (nat-upto-template (add1 n) t) ...)])
)```
A template for counting up

;;; (nat-template n t) a template on n up to t.
;;; nat-template: Nat -> Any
;;; Requires: n <= t

(define (nat-upto-template n t)
  (cond [(= n t) ...]
        [else (... n ... (nat-upto-template (add1 n) t) ...)]))

Exercise

Use recursion to complete the function list-cubes.

;;; (list-cubes b t) return the list of cubes from b*b*b to t*t*t.
;;; list-cubes: Nat Nat -> (listof Nat)
;;; Examples:
(check-expect (list-cubes 2 5) (list 8 27 64 125))
Step counting

We can count up (or down) by numbers other than 1. Simply replace \((\textit{add1} \ n)\) with \((+ \ n \ k)\) to count up by \(k\), or replace \((\textit{sub1} \ n)\) with \((- \ n \ k)\) to count down by \(k\).

Exercise

Write a function \((\textit{countdown-by} \ \text{top} \ \text{step})\) that returns a list of \textit{Nat} so the first is \(\text{top}\), the next is \(\text{step}\) less, and so on, until the next one would be zero or less.

\[
\begin{align*}
(\text{countdown-by} \ 15 \ 3) & \Rightarrow (\text{list} \ 15 \ 12 \ 9 \ 6 \ 3) \\
(\text{countdown-by} \ 14 \ 3) & \Rightarrow (\text{list} \ 14 \ 11 \ 8 \ 5 \ 2)
\end{align*}
\]

Exercise

Write a recursive function \((\textit{step-sqr-sum-between} \ \text{lo} \ \text{hi} \ \text{step})\), that returns the sum of squares of the numbers starting at \(\text{lo}\) and ending before \(\text{hi}\), spaced by \(\text{step}\).

That is, duplicate the following function:

\[
\begin{align*}
&\text{(define} \ (\text{step-sqr-sum-between} \ \text{lo} \ \text{hi} \ \text{step}) \\
&\qquad (\text{foldr} + 0 (\text{map} \ \text{sqr} \ (\text{range} \ \text{lo} \ \text{hi} \ \text{step}))))
\end{align*}
\]
Lists are defined recursively

It is very easy to add an item at the front of a list:

\[(\text{cons} \ 42 \ (\text{list} \ 6 \ 7)) \Rightarrow (\text{list} \ 42 \ 6 \ 7)\]

It is slightly more tricky to add at the back of the list:

\[
\begin{align*}
& (\text{foldr} \ \text{cons} \ (\text{list} \ 42) \ (\text{list} \ 6 \ 7)) \Rightarrow (\text{list} \ 6 \ 7 \ 42) \\
& (\text{append} \ (\text{list} \ 6 \ 7) \ (\text{list} \ 42)) \Rightarrow (\text{list} \ 6 \ 7 \ 42)
\end{align*}
\]

But there is no built-in, super-easy way to do it. Why?

Answer: in Racket lists are actually defined recursively.

A \(\text{(listof Int)}\) is either

\[
\begin{align*}
& '(()), \text{ or} \\
& \text{(cons} \ v \ L) \text{ where } v \text{ is an } \text{Int} \text{ and } L \text{ is a } \text{(listof Int)}.
\end{align*}
\]

Recall \'(\) is a special symbol: the empty list.

\[
\begin{align*}
& (\text{list} \ 3) \Leftrightarrow (\text{cons} \ 3 \ '(\)) \\
& (\text{list} \ 6 \ 7) \Leftrightarrow (\text{cons} \ 6 \ (\text{cons} \ 7 \ '(\)) )
\end{align*}
\]
A template for functions that process lists

The data definition for any list will resemble that of a (listof Int):

A (listof Int) is either
- '()', or
- (cons v L) where v is an Int and L is a (listof Int).

Recall that recursive functions may follow this structure:
- a base case specifies the result for a special value.
- recursive cases specify the result in terms of the function itself, closer to the base case.

The base case is the empty list, '(). We can get closer to it using rest. So the template is:

;;; (listof-int-template L) a template on L.
;;; listof-int-template: (listof Int) -> Any
(define (listof-int-template L)
  (cond [(equal? L '()) ...]
        [else (... (first L) ... (listof-int-template (rest L)) ...)])
A shortcut

It is so common to need to check for the empty list, there is a special predicate `empty?`. `empty?` consumes one value, and returns `#true` if its argument is `'(())`, and `#false` otherwise.

`(empty? L)` and `(equal? L '(()))` are exactly equivalent.

The template may then be written:

```
;; (listof-int-template L) a template on L.
;; listof-int-template: (listof Int) -> Any
(define (listof-int-template L)
  (cond [(empty? L) ...
         [else (... (first L) ... (listof-int-template (rest L)) ...)]]))
```
Generic Template for lists

The template previously applied specifically to `(listof Int)`.
A generic template can be used for any type `x`. You do not need to write a template for each specific type.

```scheme
;; (listof-x-template L) a template on L.
;; listof-x-template: (listof X) -> Any
(define (listof-x-template L)
  (cond [(empty? L) ...]
        [else (... (first L) ... (listof-x-template (rest L)) ...)]))
```

You will use this template many times — for every function you write that recurses on lists!
Write a recursive function `sum` that consumes a `(listof Int)` and returns the sum of all the values in the list.

```
(sum (list 6 7 42)) => 55
```
That is, use recursion to duplicate the following function:

```
(define (sum L) (foldr + 0 L))
```

---

Write a recursive function `keep-evens` that consumes a `(listof Int)` and returns the list of even values.

That is, use recursion to duplicate the following function:

```
(define (keep-evens L) (filter even? L))
```
What is the largest value in an empty list? Is it zero? $\infty$? $-\infty$?

The question does not make sense. Some computations only makes sense on a nonempty list. For such a function, add a Requires section to the design recipe:

```
;; (list-max L) return the greatest value in L.
;; list-max: (listof Int) -> Int
;; Requires: L is not empty.
;; Example:
(check-expect (list-max (list 3 7 4)) 7)
(check-expect (list-max (list -3 -7 -4)) -3)
```

If we require that our input be a nonempty list, we can’t use the empty list as a base case — we should never receive it as input!

Instead: `(empty? (rest L))` will return #true when L has just one value left in it. This is a perfect base case for our `list-max` function.

**Exercise**: Write a recursive function `list-max` that consumes a nonempty `(listof Int)` and returns the largest value in the list.
You now have tools powerful enough to solve the problem we started with. A Collatz sequence is defined as follows:

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

**Exercise**

Write a function `(collatz-seq sk)` that returns the Collatz sequence starting at `sk`, until it reaches 1.

- `(collatz-seq 5) => (list 5 16 8 4 2 1)`
- `(collatz-seq 1) => (list 1)`
Be comfortable with the following terms: recursion, base case, recursive case, data definition.

Understand recursive data definitions for Nat and (listof Any).

Understand how to build a recursive template based on a recursive data definition, and be able to use the template to write recursive functions that consume the data type.

Before we begin the next module, please

- Read How to Design Programs Section 17.