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,

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

Consider \( s_k = 12 \). This is even, so \( s_{k+1} = s_k / 2 = 12 / 2 = 6 \).

Consider \( s_k = 3 \). This is odd, so \( s_{k+1} = 3s_k + 1 = 3(3) + 1 = 10 \).

Write a function \((\text{collatz-next } sk)\) that consumes a \texttt{Nat} representing an item in a Collatz sequence, and returns the next item in the sequence.

\[
\begin{align*}
\text{(check-expect (collatz-next 3) 10)} \\
\text{(check-expect (collatz-next 12) 6)}
\end{align*}
\]
Collatz Sequences

You might notice:

\[(\text{collatz-next } 1) => 4 \quad (\text{collatz-next } 4) => 2 \quad (\text{collatz-next } 2) => 1\]

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

Numbers seem to eventually reach 1: 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

Assume every starting number will eventually reach 1. (Nobody knows for sure.)

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

\[
(\text{collatz-seq } 13) => (\text{list } 13 \ 40 \ 20 \ 10 \ 5 \ 16 \ 8 \ 4 \ 2 \ 1)
\]

\[
(\text{collatz-seq } 21) => (\text{list } 21 \ 64 \ 32 \ 16 \ 8 \ 4 \ 2 \ 1)
\]

\[
(\text{collatz-seq } 1) => (\text{list } 1)
\]
Collatz Sequences

Clearly this computation is possible, since I can do it by hand. But none of our tools are powerful enough to complete it!

It is not possible to complete this task using and combination of \texttt{map}, \texttt{filter}, and \texttt{foldr}. We need a more powerful tool: \textit{recursion}. 
A definition that refers to itself is said to be **recursive**.

Example: The **Peano axioms** which define the natural numbers include:

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 (+ n 1) to create the successor. (Then (- n 1) 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
;; (+ r 1) where r is a Nat.
```
Consider the function \( n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1 \).

For example:
- \( 3! = 3 \times 2 \times 1 = 6 \)
- \( 4! = 4 \times 3 \times 2 \times 1 = 24 \)

If we add brackets, we see:
\[
4! = 4 \times (3 \times 2 \times 1) \quad \text{But} \quad 3 \times 2 \times 1 = 3! \quad \text{So} \quad 4! = 4 \times 3!
\]

It’s like the function has a “smaller version of itself” inside. “Usually”, \( n! = n \times (n - 1)! \)
A recursive data definition includes two parts:

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

Example:

```plaintext
;; A Nat is either:
;; 0 or
;; (+ r 1) 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:

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

```plaintext
;; (factorial n) return n!
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

Exercise:

Try out `factorial` and see that it works.
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:

```
;; A Nat is either:
;; 0 or
;; (+ r 1) where r is a Nat.
```

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

```
;; (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 (- n 1)) ...)]))
```
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 (- n 1)) ...)])
)

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) ?
   Think: “for what value(s) do I know the answer without doing any work?”
   Test! Make sure it works for the base case!

2. Then fill in the **recursive case**(s).
   The recursive case must be closer to the base case.
   For this template, the (- n 1) brings us closer to the base.
   Test more!
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 (- n 1)) ...)]))

1. Base case(s) first.
2. Then recursive case(s).

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

```scheme
;; A Nat7 is either:
;; 7 or
;; (+ r l) where r is a Nat7.
```

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

```scheme
;; (nat7-template n) a template n down to 7.
;; nat-template: Nat7 -> Any
(define (nat7-template n)
  (cond [(= n 7) ...]
        [else (... n ... (nat-template (- n 1)) ...)])
)```
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
;; (int-b-template n b) a template n down to b.
;; int-b-template: Int Int -> Any
;; Requires: n >= b
(define (int-b-template n b)
  (cond [(= n b) ...
        [else (... n ... (int-b-template (- n 1) b) ...)])))
```

Note here the parameter $b$ is passed to the recursive call, unchanged. Only $n$ changes.
;; (int-b-template n b) a template n down to b.
;; int-b-template: Int Int -> Any
;; Requires: n >= b
(define (int-b-template n b)
  (cond [(= n b) ...
        [else (... n ... (int-b-template (- n 1) b) ...)]]))

Exercise
Write a recursive function (sum-between n b) than consumes two Nat, with n ≥ b, and returns the sum of all Nat between b and n.

(sum-between 5 3) => (+ 5 4 3) => 12

Remember: always fill in the base case first.

Exercise
Complete countdown-to using recursion.

;; (countdown-to n b) return a list of Int from n down to b.
;; countdown-to: Int Int -> (listof Int)
;; Examples:
(check-expect (countdown-to 2 0) (cons 2 (cons 1 (cons 0 '()))))
(check-expect (countdown-to 5 2) (list 5 4 3 2))
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 \((+\ n\ 1)\):

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

(define (nat-upto-template n t)
  (cond [(= n t) ...]
        [else (... n ... (nat-upto-template (+ n 1) t) ...)])
)```
A template for counting up

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

(define (nat-upto-template n t)
  (cond [(= n t) ...
        [else (... n ... (nat-upto-template (+ n 1) t) ...)]]))

Exercise

Use recursion to complete the function list-cubes.

;; (list-cubes b t) return the list of cubes from b*b*b up to t*t*t.
;; list-cubes: Nat Nat -> (listof Nat)
;; Examples:
(check-expect (list-cubes 2 5) (list 8 27 64 125))
We can count up (or down) by numbers other than 1. Simply replace \((+ \ n \ 1)\) with \((+ \ n \ k)\) to count up by \(k\), or replace \((- \ n \ 1)\) with \((- \ n \ k)\) to count down by \(k\).

**Exercise**
Write a function `(countdown-by top step)` that returns a list of Nat so the first is `top`, the next is `step` less, and so on, until the next one would be zero or less.

\[
\text{(countdown-by 15 3) => (list 15 12 9 6 3)}
\]

\[
\text{(countdown-by 14 3) => (list 14 11 8 5 2)}
\]

**Exercise**
Write a recursive function `(step-sqr-sum-between lo hi step)`, that returns the sum of squares of the numbers starting at `lo` and ending before `hi`, spaced by `step`.

That is, duplicate the following function:

\[
\text{(define (step-sqr-sum-between lo hi step)}
\text{  (foldr + 0 (map sqr (range lo hi step))))}
\]
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:

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

But there is no built-in, super-easy way to do it. Why?
Answer: in Racket lists are actually defined recursively.

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

- '(), or
- \((\text{cons} \ v \ L)\) where \(v\) is an \text{Int} and \(L\) is a \((\text{listof} \ \text{Int})\).

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

\[(\text{list} \ 3) \leftrightarrow (\text{cons} \ 3 \ '()) \quad (\text{list} \ 6 \ 7) \leftrightarrow (\text{cons} \ 6 \ (\text{cons} \ 7 \ '()))\]
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)) ...)]))
```
The template previously applied specifically to \((\text{listof Int})\).

A generic template can be used for any type \(x\). You do not need to write a template for each specific type.

\[
\begin{align*}
&\text{;;; (listof-} x \text{-template L) a template on L.} \\
&\text{;;; listof-} x \text{-template: (listof X) -> Any} \\
&(\text{define (listof-} x \text{-template L)}) \\
&(\text{cond [((empty? L) ...)]} \\
&\quad [\text{else} \\
&\quad \quad (\ldots (\text{first L}) \ldots (\text{listof-} x \text{-template (rest L)) ...})])
\end{align*}
\]

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))
Doing work on the way

Our functions can do work on the variables as we go. They can:

- Keep only some values (like \texttt{filter})
- Change values (like \texttt{map})
- Combine value (like \texttt{foldr})
- and more....

;;; (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

\begin{verbatim}
(define (even-squares-between hi lo)
  (cond [(<= hi lo) '()] [(even? hi) (cons (sqr hi) (even-squares-between (- hi 1) lo))] [else (even-squares-between (- hi 1) lo)]))
\end{verbatim}

This does the same thing: \begin{verbatim}
(define (even-squares-btw hi lo)
  (map sqr (filter even? (range hi lo -1))))
\end{verbatim}
Recursion can do anything

Any code we wrote using map, filter, or foldr can we written using only recursion. It will often be harder to write, and almost certainly harder to read. It is particularly important to write the design recipe!
Example: the same thing three ways

Higher order functions:

```
(define (even-squares-btw hi lo)
  (map sqr (filter even? (range hi lo -1))))
```

A single `foldr`:

```
(define (even-squares-foldr hi lo)
  (foldr (lambda (a b)
    (cond [[(even? a) (cons (sqr a) b)]
          [else b]])
    '()
    (range hi lo -1)))
```

Using recursion only:

```
(define (even-squares-between-between hi lo)
  (cond [(<= hi lo) '()] [(even? hi) (cons (sqr hi) (even-squares-between-between (- hi 1) lo))]
                [else (even-squares-between-between (- hi 1) lo)])
```
The same thing three ways

Higher order functions:

```
(define (muddle-down hi lo)
  (map (lambda (x) (* x 2))
    (filter (lambda (y) (= 1 (remainder y 3)))
      (range hi lo -1))))
```

↕

A single foldr:

```
(define (muddle-down-foldr hi lo)
  (foldr (lambda (a b)
    (cond 
      [(= 1 (remainder a 3)) (cons (* a 2) b)]
      [else b]))
    ()
    (range hi lo -1)))
```

↕

Ex. Duplicate the behaviour of muddle-down and muddle-down-foldr using recursion only.
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:

```scheme
;; (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: \( (\text{empty?} \ (\text{rest} \ L)) \) 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 \((\text{listof Int})\) 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:

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

This does not fit into the templates we encountered. Solving it is a challenge, but possible!

Consider what \texttt{collatz-seq sk} need to do when:

1. \( s_k \) is 1?
2. \( s_k \) is an even number?
3. \( s_k \) is an odd number other than 1?

Exercise

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

(\texttt{collatz-seq 13}) \Rightarrow (\texttt{list 13 40 20 10 5 16 8 4 2 1})

(\texttt{collatz-seq 21}) \Rightarrow (\texttt{list 21 64 32 16 8 4 2 1})

(\texttt{collatz-seq 1}) \Rightarrow (\texttt{list 1})
Module Summary

- Be comfortable with the following terms: recursion, base case, recursive case, data definition.
- Understand recursive data definitions for \texttt{Nat} and \texttt{(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.

Further Reading: \textit{How to Design Programs} Sections 9, 10