Recursion is an important tool for working with lists. In this lecture module we extend our recursive capabilities with a self-referential data definition for natural numbers, a corresponding template, and examples of problems that we can now solve.

We’ll start with a quick review of lists because our work with natural numbers will have many parallels.
We’ll review how we derived the list template.

;; A (listof X) is one of:
;; * empty
;; * (cons X (listof X))
Rewrite listof-int? without any cond expressions. That is, use only a boolean expression.
Suppose we have a list \( \textit{lst} \).

The test \((\text{empty? } \textit{lst})\) tells us which case applies.

If \((\text{empty? } \textit{lst})\) is \textit{false}, then \textit{lst} is of the form \((\text{cons } f \ r)\).

How do we compute the values \( f \) and \( r \)?

\( f \) is \((\text{first } \textit{lst})\).

\( r \) is \((\text{rest } \textit{lst})\).

Because \( r \) is a list, we recursively apply the function we are constructing to it.
We can repeat this reasoning on a recursive definition of natural numbers to obtain a template.
Logicians use the Peano axioms to define the natural numbers. These include:

- 0 is a natural number.
- 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.

(A handful of other axioms define the rest of the behaviour of natural numbers, but we don’t need to go into them here.)
The successor function $S(n)$ produces the “next” natural number. We will use the Racket function `add1` as the successor function:

- \((\text{add1 } 0) \Rightarrow 1\)
- \((\text{add1 } 1) \Rightarrow 2\)
- \((\text{add1 } 2) \Rightarrow 3\)

With this function, we can translate the logicians’ axioms into a Racket data definition:

- 0 is a natural number.
- For every natural number $n$, $S(n)$ is a natural number.

\[\Rightarrow \quad \text{A Nat is one of:} \]
\[\text{;; } 0 \quad \text{;; } (\text{add1 Nat})\]
Our data definition for a Nat is as follows:

``` ;; A Nat is one of: ;; 0 ;; (add1 Nat)```

The natural numbers start at 0 in computer science and some branches of mathematics (e.g., logic).

We’ll now work out a template for functions that consume a natural number.
Suppose we have a natural number $n$.

The test `(zero? n)` tells us which case applies.

If `(zero? n)` is `false`, then $n$ has the value `(add1 k)` for some $k$.

To compute $k$, we use the inverse function of `add1`, which is `sub1`.

Because the result `(sub1 n)` is a natural number, we recursively apply the function we are constructing to it.

```scheme
;; nat-template: Nat -> Any
(define (nat-template n)
  (cond [(zero? n) ...]
        [else (... n ...
              ... (nat-template (sub1 n)) ...)])
)```
Goal: `countdown`, which consumes a natural number \( n \) and produces a decreasing list of all natural numbers less than or equal to \( n \).

\[
\text{(countdown 0)} \Rightarrow (\text{cons } 0 \text{ empty})
\]

\[
\text{(countdown 1)} \Rightarrow (\text{cons } 1 (\text{cons } 0 \text{ empty}))
\]

\[
\text{(countdown 2)} \Rightarrow (\text{cons } 2 (\text{cons } 1 (\text{cons } 0 \text{ empty})))
\]

With these examples, we proceed by filling in the template.
;; (countdown n) produces a decreasing list of Nats from n to 0
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

;; countdown: Nat -> (listof Nat)
(define (countdown n)
  (cond [((zero? n) ...) ]
    [else (... n ...
      ... (countdown (sub1 n)) ...) ]))

If \( n \) is 0, we produce the list containing 0, and if \( n \) is nonzero, we cons \( n \) onto the countdown list for \( n - 1 \).
;; (countdown n) produces a decreasing list of Nats from n to 0
;; Example:
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

;; countdown: Nat -> (listof Nat)
(define (countdown n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]))
> A condensed trace

(countdown 2)
⇒ (cons 2 (countdown 1))
⇒ (cons 2 (cons 1 (countdown 0)))
⇒ (cons 2 (cons 1 (cons 0 empty)))
Write a recursive function \( \text{sum-to } n \) that consumes a \( \text{Nat} \) and produces the sum of all \( \text{Nat} \) between \( 0 \) and \( n \).

\[
\text{(sum-to 4) } \Rightarrow \ (\ + \ 4 \ (+ \ 3 \ (+ \ 2 \ (+ \ 1 \ 0)))) \Rightarrow 10
\]
The symbol $\mathbb{Z}$ is often used to denote the integers.

We can add subscripts to define subsets of the integers (also known as **intervals**).

For example, $\mathbb{Z}_{\geq 0}$ defines the non-negative integers, also known as the natural numbers.

Other examples: $\mathbb{Z}_{> 4}$, $\mathbb{Z}_{< -8}$, $\mathbb{Z}_{\leq 1}$.
If we change the base case test from `(zero? n)` to `(= n 7)`, we can stop the countdown at 7.

This corresponds to the following definition:

```scheme
;; An integer in \( \mathbb{Z}_{\geq 7} \) is one of:
;;   7
;;   (add1 \( \mathbb{Z}_{\geq 7} \))
```

We use this data definition as a guide when writing functions, but in practice we use a requires section in the contract to capture the new stopping point.
;; (countdown-to-7 n) produces a decreasing list from n to 7
;; Example:
(check-expect (countdown-to-7 9) (cons 9 (cons 8 (cons 7 empty))))

;; countdown-to-7: Nat -> (listof Nat)
;; requires: n >= 7
(define (countdown-to-7 n)
  (cond [ (= n 7) (cons 7 empty)]
    [else (cons n (countdown-to-7 (sub1 n)))]))
We can generalize both \texttt{countdown} and \texttt{countdown-to-7} by providing the base value (e.g., 0 or 7) as a second parameter \texttt{b} (the “base”).

Here, the stopping condition will depend on \texttt{b}.

The parameter \texttt{b} has to \texttt{“go along for the ride”} (be passed unchanged) in the recursion.
;; (countdown-to n base) produces a decreasing list from n to base
;; Example:
(check-expect (countdown-to 4 2) (cons 4 (cons 3 (cons 2 empty))))

;; countdown-to: Int Int -> (listof Int)
;;   requires: n >= base
(define (countdown-to n base)
  (cond [(= n base) (cons base empty)]
        [else (cons n (countdown-to (sub1 n) base))]))
> Another condensed trace

\[(\text{countdown-to } 4 \ 2)\]
⇒ \[(\text{cons} \ 4 \ (\text{countdown-to } 3 \ 2))\]
⇒ \[(\text{cons} \ 4 \ (\text{cons} \ 3 \ (\text{countdown-to } 2 \ 2)))\]
⇒ \[(\text{cons} \ 4 \ (\text{cons} \ 3 \ (\text{cons} \ 2 \ \text{empty})))\]
countdown-to works just fine if we put in negative numbers.
(countdown-to 1 -2)
⇒ (cons 1 (cons 0 (cons -1 (cons -2 empty)))))
Write a recursive function \( \text{sum-between } n \ b \) than consumes two \( \text{Nat} \), with \( n \geq b \), and returns the sum of all \( \text{Nat} \) between \( b \) and \( n \).

\[(\text{sum-between } 5 \ 3) \Rightarrow (+ \ 5 \ (+ \ 4 \ 3)) \Rightarrow 12\]
What if we want an increasing count?

Consider the non-positive integers $\mathbb{Z}_{\leq 0}$.

A integer in $\mathbb{Z}_{\leq 0}$ is one of:

* 0
* (sub1 $\mathbb{Z}_{\leq 0}$)

Examples: $-1$ is (sub1 0), $-2$ is (sub1 (sub1 0)).

Since (add1 (sub1 n)) $\Rightarrow$ n for all integers n, the inverse function we need is add1.

This suggests the following template.
Notice the additional requires section.

;; nonpos-template: Int -> Any
;; requires: n ≤ 0
(define (nonpos-template n)
    (cond [(zero? n) ...]
          [else (... n ...
                  ... (nonpos-template (add1 n)) ...)]))

We can use this to develop a function to produce lists such as

(cons -2 (cons -1 (cons 0 empty))).
(check-expect (countup -2) (cons -2 (cons -1 (cons 0 empty))))

(define (countup n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countup (add1 n)))]))
As before, we can generalize this to counting up to \( b \), by introducing \( b \) as a second parameter in a template.

;; (countup-to n base) produces an increasing list from n to base
;; Example:
;; (check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty))))

;; countup-to: Int Int -> (listof Int)
;; requires: n <= base
(define (countup-to n base)
  (cond [ (= n base) (cons base empty)]
        [else (cons n (countup-to (add1 n) base))])))
Many imperative programming languages offer several language constructs to do repetition:

```plaintext
for i = 1 to 10 do { ... }
```

Racket offers one construct – recursion – that is flexible enough to handle these situations and more.

We will soon see how to use Racket’s abstraction capabilities to abbreviate many common uses of recursion.
When you are learning to use recursion, sometimes you will “get it backwards” and use the countdown pattern when you should be using the countup pattern, or vice-versa.

If you’re building a list and get it backwards, avoid using the built-in list function `reverse` to fix your error. It cannot always save a computation done in the wrong order.

Instead, learn to fix your mistake by using the right pattern.

You may **not** use `reverse` on assignments unless we say otherwise.
You should understand the recursive definition of a natural number, and how it leads to a template for recursive functions that consume natural numbers.

You should understand how subsets of the integers greater than or equal to some bound \( m \), or less than or equal to such a bound, can be defined recursively, and how this leads to a template for recursive functions that “count down” or “count up”. You should be able to write such functions.
In this module we added the following to our toolbox:

`add1` `sub1` `zero?`

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

Write a function \((\text{countdown-by} \ \text{top} \ \text{step})\) that returns a list of \text{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.

\((\text{countdown-by} \ 12 \ 3) \Rightarrow (\text{cons} \ 12 (\text{cons} \ 9 (\text{cons} \ 6 (\text{cons} \ 3 \ \text{empty}))))\)

\((\text{countdown-by} \ 11 \ 3) \Rightarrow (\text{cons} \ 11 (\text{cons} \ 8 (\text{cons} \ 5 (\text{cons} \ 2 \ \text{empty}))))\)

\textbf{Consider: how must you change the base case of the template?}
This exercise recurses on a list and a Nat at the same time.

Complete n-th-item.

;;; (n-th-item L n) Produce the n-th item in L, where (first L) is
;;;   the 0th.
;;; Example:
(check-expect (n-th-item (cons 3 (cons 7 (cons 31 (cons 63 empty)))))) 0) 3)
(check-expect (n-th-item (cons 3 (cons 7 (cons 31 (cons 63 empty)))))) 3) 63)

;;; n-th-item: (listof Any) Nat -> Any
(define (n-th-item lst n) ...)