Natural numbers – recursively

**Readings:** HtDP, sections 11, 12, 13 (Intermezzo 2).

**Topics:**
- Review: data def and templates
- Natural numbers: data def and templates
- Intervals
- Counting up

---

Review: from definition to template

We'll review how we derived the list template.

```scheme
;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))
```

---

**Exercise 1**

Rewrite `listof-int?` without any `cond` expressions. That is, use only a boolean expression.
Suppose we have a list `lst`.

The test `(empty? lst)` tells us which case applies.

If `(empty? lst)` is `false`, then `lst` is of the form `(cons f r)`.

How do we compute the values `f` and `r`?

- `f` is `(first lst)`.
- `r` is `(rest lst)`.

Because `r` is a list, we recursively apply the function we are constructing to it.

```
(listof-X-template lst)
```

We can repeat this reasoning on a recursive definition of natural numbers to obtain a template.

```
(define (listof-X-template lst)
  (cond [(empty? lst) ...]
        [else (... (first lst) ...
                 (listof-X-template (rest lst)) ...)])
```

Here `add1` is the built-in function that adds 1 to its argument.

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.
> nat-template

Suppose we have a natural number \(n\).

The test \((\text{zero? } n)\) tells us which case applies.

If \((\text{zero? } n)\) is false, then \(n\) has the value \((\text{add1 } k)\) for some \(k\).

To compute \(k\), we subtract 1 from \(n\), using the built-in \(\text{sub1}\) function.

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

\[
\text{;; nat-template: Num} \rightarrow \text{Any}
\]
\[
\begin{align*}
\text{(define (nat-template } n) \text{)} \\
\text{ (cond ([} (\text{zero? } n) ... ]} \\
\text{ [else } (... n ... \\
\text{ ... (nat-template (sub1 } n) ...) ... ]))
\end{align*}
\]

---

> Example: a decreasing list

Goal: \(\text{countdown}\), which consumes a natural number \(n\) and produces a decreasing list of all natural numbers less than or equal to \(n\).

\[
\begin{align*}
(\text{countdown } 0) & \Rightarrow (\text{cons } 0 \text{ empty}) \\
(\text{countdown } 1) & \Rightarrow (\text{cons } 1 (\text{cons } 0 \text{ empty})) \\
(\text{countdown } 2) & \Rightarrow (\text{cons } 2 (\text{cons } 1 (\text{cons } 0 \text{ empty})))
\end{align*}
\]

With these examples, we proceed by filling in the template.

\[
\begin{array}{ccc}
0 & 1 & 2 \\
0 & 1 & 0 \end{array}
\]

---

> countdown

\[
\text{;; (countdown } n) \text{ produces a decreasing list of Nats from } n \text{ to } 0 \\
(\text{check-expect (countdown } 0) \text{ ) } (\text{cons } 0 \text{ empty})) \\
(\text{check-expect (countdown } 2) \text{ ) } (\text{cons } 2 (\text{cons } 1 (\text{cons } 0 \text{ empty}))))
\]

\[
\text{;; countdown: Nat} \rightarrow (\text{listof Nat})
\]
\[
\begin{align*}
\text{(define (countdown } n) \text{)} \\
\text{ (cond ([} (\text{zero? } n) ... ]} \\
\text{ [else } (... n ... \\
\text{ ... (countdown (sub1 } n) ...) ... ]))
\end{align*}
\]

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

;; (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)))

Exercise 2

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

(sum-to 4) ⇒ (+ 4 (+ 3 (+ 2 (+ 1 0)))) ⇒ 10
Intervals of the natural numbers

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: $\mathbb{Z}_{> 4}, \mathbb{Z}_{< -8}, \mathbb{Z}_{\leq 1}$.

> Example: $\mathbb{Z}_{\geq 7}$

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

```scheme
;; (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))))
```

```scheme
;; 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)))]))
```
Generalizing `countdown` and `countdown-to-7`

We can generalize both `countdown` and `countdown-to-7` by providing the base value (e.g., 0 or 7) as a second parameter \( b \) (the “base”).

Here, the stopping condition will depend on \( b \).

The parameter \( b \) has to “go along for the ride” (be passed unchanged) in the recursion.

### Another condensed trace

\[
\begin{align*}
(\text{countdown-to } 4 \ 2) & \Rightarrow (\text{cons } 4 \ (\text{countdown-to } 3 \ 2)) \\
& \Rightarrow (\text{cons } 4 \ (\text{cons } 3 \ (\text{countdown-to } 2 \ 2))) \\
& \Rightarrow (\text{cons } 4 \ (\text{cons } 3 \ \text{empty}))
\end{align*}
\]
countdown-to with negative numbers

countdown-to works just fine if we put in negative numbers.

(countdown-to 1 -2)
⇒ (cons 1 (cons 0 (cons -1 (cons -2 empty))))

Exercise 3

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 \( n \).

\( \text{sum-between} \ 5 \ 3 \) ⇒ (+ 5 (+ 4 3)) ⇒ 12

Counting up

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 \( \text{sub1} \ 0 \), \(-2\) is \( \text{sub1} \ (\text{sub1} \ 0) \).

If an integer \( i \) is of the form \( \text{sub1} \ k \), then \( k \) is equal to \( \text{add1} \ i \). 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))).

;; (countup n) produces an increasing list from n to 0
;; Example:
(check-expect (countup -2) (cons -2 (cons -1 (cons 0 empty))))

;; countup: Int → (listof Int)
;; requires: n <= 0
(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))])))
> Repetition in other languages

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.

> reverse

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.

Goals of this module

- 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.
Exercise 4

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

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

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

Consider: how must you change the base case of the template?

Exercise 5

This exercise recurses on a list and a \(\text{Nat}\) at the same time.

Complete \(n\text{-th-item}\).

\[
;; \ (n\text{-th-item } L \ n) \ \text{Produce the } n\text{-th item in } L, \ \text{where (first } L) \ \text{is}
\]

\[
;; \ \text{the } 0\text{th.}
\]

\[
;; \ \text{Example:}
\]

\[
(\text{check-expect } (n\text{-th-item } '(3 \ 7 \ 31 \ 2047 \ 8191) \ 0) \ 3)
\]

\[
(\text{check-expect } (n\text{-th-item } '(3 \ 7 \ 31 \ 2047 \ 8191) \ 3) \ 2047)
\]

\[
;; \ n\text{-th-item: } (\text{listof Any} \ \text{Nat} \rightarrow \text{Any}
\]

\[
\text{(define } (n\text{-th-item } lst \ n) ...)
\]