Natural numbers – recursively

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

Topics:

- Review: data def and templates
- Natural numbers: data def and templates
- Subintervals
- Counting up

Review: from definition to template

We'll review how we derived the list template.

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

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

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

Natural numbers

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

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.

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 subtract 1 from n, using the built-in sub1 function.

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

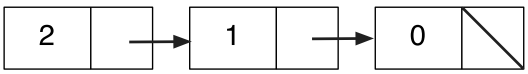
```
(define (nat-template n)
  (cond [(zero? n) ...]
        [else (... n ...
              ... (nat-template (sub1 n)) ...)]))
```

Example: a decreasing list

Goal: countdown, which consumes a natural number n and produces a decreasing list of all natural numbers less than or equal to n.

(countdown 0) ⇒ (cons 0 empty)

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



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


```
;; (countdown n) produces a decreasing list of Nats from n to 0
;; countdown: Nat → (listof Nat)
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

(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
;; countdown: Nat → (listof Nat)
;; Example:
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

(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)))

Subintervals 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.

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:

;; 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 contact to capture the new stopping point.

```
;; (countdown-to-7 n) produces a decreasing list from n to 7
;; countdown-to-7: Nat → (listof Nat)
;; requires: n ≥ 7
;; Example:
(check-expect (countdown-to-7 9) (cons 9 (cons 8 (cons 7 empty))))

(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.


```
;; (countdown-to n b) produces a decreasing list from n to b
;; countdown-to: Int Int → (listof Int)
;; requires: n ≥ b
;; Example:
(check-expect (countdown-to 4 2) (cons 4 (cons 3 (cons 2 empty))))

(define (countdown-to n b)
  (cond [(= n b) (cons b empty)]
        [else (cons n (countdown-to (sub1 n) b))]))
```

Another condensed trace

(countdown-to 4 2)

⇒ (cons 4 (countdown-to 3 2))

⇒ (cons 4 (cons 3 (countdown-to 2 2)))

⇒ (cons 4 (cons 3 (cons 2 empty)))

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))))

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 (sub1 0), -2 is (sub1 (sub1 0)).

If an integer i is of the form (sub1 k), then k is equal to (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
;; countup: Int → (listof Int)
;; requires: n ≤ 0
;; Example:
(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 b) produces an increasing list from n to b
;; countup-to: Int Int → (listof Int)
;; requires: n ≤ b
;; Example:
(check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty))))

(define (countup-to n b)
  (cond [(= n b) (cons b empty)]
        [else (cons n (countup-to (add1 n) b))]))
```

Many imperative programming languages offer several language constructs to do repetition:

```
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.

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.